

ulrich BREYMANN

2. Auflage

DER C++ PROGRAMMIERER

C++ LERNEN -
PROFESSIONELL ANWENDEN -
LÖSUNGEN NUTZEN



EXTRA: Mit kostenlosem E-Book



Auf DVD: Entwicklungsumgebung, Compiler,
weitere Tools, alle Beispiele des Buchs



Topaktuell: Entspricht dem neuen
ISO-C++-Standard

HANSER

Der C++-Programmierer



Bleiben Sie einfach auf dem Laufenden:
www.hanser.de/newsletter

Sofort anmelden und Monat für Monat
die neuesten Infos und Updates erhalten.

Ulrich Breymann



Der C++-Programmierer

**C++ lernen -
Professionell anwenden -
Lösungen nutzen**

2., aktualisierte Auflage

HANSER

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Autor und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2011 Carl Hanser Verlag München (www.hanser.de)

Lektorat: Margarete Metzger

Copy editing: Jürgen Dubau, Freiburg

Herstellung: Irene Weilhart

Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München

Umschlagrealisation: Stephan Rönigk

Datenbelichtung, Druck und Bindung: Kösel, Krugzell

Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702

Printed in Germany

ISBN 978-3-446-42691-7

E-Book-ISBN 978-3-446-42841-6

Inhalt

Vorwort	21
Teil I: Einführung in C++	25
1 Es geht los!	27
1.1 Historisches	27
1.2 Objektorientierte Programmierung	28
1.3 Compiler	31
1.4 Das erste Programm	31
1.4.1 Namenskonventionen	36
1.5 Integrierte Entwicklungsumgebungen	37
1.5.1 Code::Blocks	37
1.5.2 Eclipse	39
1.6 Einfache Datentypen und Operatoren	41
1.6.1 Ausdruck	42
1.6.2 Ganze Zahlen	42
1.6.3 Reelle Zahlen	47
1.6.4 Konstante	50
1.6.5 Zeichen	51
1.6.6 Logischer Datentyp bool	54
1.6.7 Referenzen	55
1.6.8 Regeln zum Bilden von Ausdrücken	56
1.6.9 Standard-Typumwandlungen	57

1.7	Gültigkeitsbereich und Sichtbarkeit	58
1.7.1	Namespace std	60
1.8	Kontrollstrukturen	61
1.8.1	Anweisungen	61
1.8.2	Sequenz (Reihung)	63
1.8.3	Auswahl (Selektion, Verzweigung)	63
1.8.4	Fallunterscheidungen mit switch	67
1.8.5	Wiederholungen	69
1.8.6	Kontrolle mit break und continue	77
1.9	Benutzerdefinierte und zusammengesetzte Datentypen	79
1.9.1	Aufzählungstypen	79
1.9.2	Arrays: Der C++-Standardtyp vector	81
1.9.3	Zeichenketten: Der C++-Standardtyp string	86
1.9.4	Strukturen	88
1.9.5	Typermittlung mit auto	90
1.9.6	Unions und Bitfelder	91
2	Einfache Ein- und Ausgabe	93
2.1	Standardein- und -ausgabe	93
2.2	Ein- und Ausgabe mit Dateien	96
3	Programmstrukturierung	101
3.1	Funktionen	102
3.1.1	Aufbau und Prototypen	102
3.1.2	Gültigkeitsbereiche und Sichtbarkeit in Funktionen	104
3.1.3	Lokale static-Variable: Funktion mit Gedächtnis	105
3.2	Schnittstellen zum Datentransfer	106
3.2.1	Übergabe per Wert	107
3.2.2	Übergabe per Referenz	111
3.2.3	Gefahren bei der Rückgabe von Referenzen	112
3.2.4	Vorgegebene Parameterwerte und variable Parameterzahl	113
3.2.5	Überladen von Funktionen	114
3.2.6	Funktion main()	115
3.2.7	Beispiel Taschenrechnersimulation	116
3.2.8	Spezifikation von Funktionen	121
3.3	Modulare Programmgestaltung	122
3.3.1	Steuerung der Übersetzung nur mit #include	122
3.3.2	Einbinden vorübersetzter Programmteile	123

3.3.3	Dateiübergreifende Gültigkeit und Sichtbarkeit.....	124
3.3.4	Übersetzungseinheit, Deklaration, Definition	126
3.3.5	Compilerdirektiven und Makros	128
3.4	Funktions-Templates	134
3.4.1	Spezialisierung von Templates	137
3.4.2	Einbinden von Templates	138
3.5	inline-Funktionen.....	139
3.6	Namensräume	141
3.7	C++-Header.....	142
3.7.1	Einbinden von C-Funktionen	144
4	Objektorientierung 1	147
4.1	Abstrakte Datentypen	148
4.2	Klassen und Objekte	149
4.2.1	inline-Elementfunktionen.....	152
4.3	Initialisierung und Konstruktoren	154
4.3.1	Standardkonstruktor	154
4.3.2	Allgemeine Konstruktoren	155
4.3.3	Kopierkonstruktor.....	158
4.3.4	Typumwandlungskonstruktor	160
4.4	Beispiel: Rationale Zahlen	162
4.4.1	Aufgabenstellung	162
4.4.2	Entwurf.....	163
4.4.3	Implementation.....	166
4.5	const-Objekte und Methoden.....	170
4.6	Destruktoren	171
4.7	Wie kommt man zu Klassen und Objekten? Ein Beispiel.....	173
4.7.1	Einige Analyse-Überlegungen.....	174
4.7.2	Formulierung des Szenarios in C++	177
4.8	Gegenseitige Abhängigkeit von Klassen	180
4.9	Konstruktor und mehr vorgeben oder verbieten.....	182
4.10	Delegierender Konstruktor.....	182
5	Intermezzo: Zeiger	185
5.1	Zeiger und Adressen.....	186
5.2	C-Arrays.....	189
5.2.1	C-Arrays und sizeof	191
5.2.2	Indexoperator bei C-Arrays.....	191

5.2.3	Initialisierung von C-Arrays	192
5.2.4	Zeigerarithmetik	192
5.3	C-Zeichenketten	193
5.4	Dynamische Datenobjekte	200
5.4.1	Freigeben dynamischer Objekte	203
5.5	Zeiger und Funktionen	205
5.5.1	Parameterübergabe mit Zeigern	205
5.5.2	Parameter des main-Programms	207
5.5.3	Gefahren bei der Rückgabe von Zeigern	208
5.6	this-Zeiger	209
5.7	Mehrdimensionale C-Arrays	209
5.7.1	Statische mehrdimensionale C-Arrays	209
5.7.2	Dynamisch erzeugte mehrdimensionale Arrays	213
5.7.3	Klasse für dynamisches zweidimensionales Array	215
5.8	Binäre Ein-/Ausgabe	220
5.9	Zeiger auf Funktionen	223
5.10	Komplexe Deklarationen lesen	226
5.11	Standard-Typumwandlungen für Zeiger	229
5.12	Zeiger auf Elementfunktionen und -daten	230
5.12.1	Zeiger auf Elementfunktionen	230
5.12.2	Zeiger auf Elementdaten	231
6	Objektorientierung 2	233
6.1	Eine String-Klasse	233
6.1.1	Optimierung der Klasse MeinString	239
6.1.2	friend-Funktionen	241
6.2	Klassenspezifische Daten und Funktionen	242
6.2.1	Klassenspezifische Konstante	245
6.3	Klassen-Templates	246
6.3.1	Ein Stack-Template	246
6.3.2	Stack mit statisch festgelegter Größe	249
6.4	Template-Metaprogrammierung	251
6.5	Variadic Templates: Templates mit variabler Parameterzahl	253
7	Vererbung	257
7.1	Vererbung und Initialisierung	263
7.2	Zugriffsschutz	264
7.3	Typbeziehung zwischen Ober- und Unterklasse	266

7.4	Code-Wiederverwendung	267
7.5	Überschreiben von Funktionen in abgeleiteten Klassen	268
7.6	Polymorphismus	270
7.6.1	Virtuelle Funktionen	270
7.6.2	Abstrakte Klassen	275
7.6.3	Virtueller Destruktor	280
7.7	Probleme der Modellierung mit Vererbung	282
7.8	Mehrfachvererbung	285
7.8.1	Namenskonflikte	288
7.8.2	Virtuelle Basisklassen	289
7.9	Standard-Typumwandlungsoperatoren	292
7.10	Typinformationen zur Laufzeit	295
7.11	Using-Deklaration für Klassen	296
7.12	Private- und Protected-Vererbung	297
8	Fehlerbehandlung	301
8.1	Ausnahmebehandlung	303
8.1.1	Exception-Spezifikation in Deklarationen	306
8.1.2	Exception-Hierarchie in C++	307
8.1.3	Besondere Fehlerbehandlungsfunktionen	308
8.1.4	Erkennen logischer Fehler	309
8.1.5	Arithmetische Fehler / Division durch 0	311
8.2	Speicherbeschaffung mit new	312
8.3	Exception-Sicherheit	315
9	Überladen von Operatoren	317
9.1	Rationale Zahlen – noch einmal	319
9.1.1	Arithmetische Operatoren	319
9.1.2	Ausgabeoperator <<	322
9.2	Eine Klasse für Vektoren	323
9.2.1	Index-Operator []	326
9.2.2	Zuweisungsoperator = ...	328
9.2.3	Mathematische Vektoren	331
9.2.4	Multiplikationsoperator	332
9.3	Inkrement-Operator ++	334
9.4	Typumwandlungsoperator	337
9.5	Smart Pointer: Operatoren -> und *	339
9.5.1	Smart Pointer und die C++-Standardbibliothek	344

9.6	Objekt als Funktion	344
9.6.1	Lambda-Funktionen	346
9.7	new und delete überladen	347
9.7.1	Speichermanagement mit malloc und free	350
9.7.2	Unterscheidung zwischen Heap- und Stack-Objekten	352
9.7.3	Fehlende delete-Anweisung entdecken	353
9.7.4	Eigene Speicherverwaltung	355
9.7.5	Empfehlungen im Umgang mit new und delete	358
9.8	Mehrdimensionale Matrizen	359
9.8.1	Zweidimensionale Matrix als Vektor von Vektoren	360
9.8.2	Dreidimensionale Matrix	363
9.9	Zuweisung bei Vererbung	365
10	Dateien und Ströme	375
10.1	Ausgabe	377
10.1.1	Formatierung der Ausgabe	377
10.2	Eingabe	380
10.3	Manipulatoren	383
10.3.1	Eigene Manipulatoren	386
10.4	Fehlerbehandlung	387
10.5	Typumwandlung von Dateiobjekten nach bool	389
10.6	Arbeit mit Dateien	390
10.6.1	Positionierung in Dateien	391
10.6.2	Lesen und Schreiben in derselben Datei	392
10.7	Umleitung auf Strings	393
10.8	Ergänzungen	395
11	Einführung in die Standard Template Library (STL)	397
11.1	Container, Iteratoren, Algorithmen	398
11.2	Iteratoren im Detail	403
11.3	Beispiel verkettete Liste	404
12	Reguläre Ausdrücke	409
12.1	Elemente regulärer Ausdrücke	410
12.1.1	Greedy oder lazy?	412
12.2	Interaktive Auswertung	413
12.3	Auszug des regex-APIs	416
12.4	Anwendungen	418

13	Threads	419
13.1	Die Klasse thread.....	423
13.2	Synchronisation.....	426
13.2.1	Thread-Group.....	428
13.3	Thread-Steuerung: pausieren, fortsetzen, beenden.....	429
13.4	Interrupt.....	434
13.5	Warten auf Ereignisse	436
13.6	Reader/Writer-Problem.....	441
13.6.1	Wenn Threads verhungern.....	446
13.6.2	Reader/Writer-Varianten	447
13.7	Thread-Sicherheit	448
 Teil II: Bausteine komplexer Anwendungen		449
14	Grafische Benutzungsschnittstellen	451
14.1	Ereignisgesteuerte Programmierung.....	452
14.2	GUI-Programmierung mit Qt.....	453
14.2.1	Meta-Objektsystem	453
14.2.2	Der Programmablauf	454
14.2.3	Speicher sparen und lokal Daten sichern	455
14.3	Signale, Slots und Widgets	457
14.4	Dialog	465
14.5	Qt oder Boost?.....	468
14.5.1	Threads	469
14.5.2	Verzeichnisbaum durchwandern	470
15	Internet-Anbindung	473
15.1	Protokolle	474
15.2	Adressen.....	474
15.3	Socket	478
15.3.1	Bidirektionale Kommunikation.....	481
15.3.2	UDP-Sockets	483
15.3.3	Atomuhr mit UDP abfragen	485
15.4	HTTP.....	488
15.4.1	Verbindung mit GET	489
15.4.2	Verbindung mit POST	494
15.5	Mini-Webserver	494

16	Datenbankanbindung	503
16.1	C++-Interface.....	504
16.2	Anwendungsbeispiel.....	508
 Teil III: Praktische Methoden und Werkzeuge der Softwareentwicklung		515
17	Abläufe automatisieren mit make	517
17.1	Quellen	518
17.2	Wirkungsweise	519
17.3	Variablen und Muster	521
17.4	Universelles Makefile für einfache Projekte	522
18	Unit-Test	525
18.1	Werkzeuge	526
18.2	Test Driven Development	527
18.3	Boost Unit Test Framework.....	528
18.3.1	Beispiel: Testgetriebene Entwicklung einer Operatorfunktion	530
18.3.2	Fixture	534
18.3.3	Testprotokoll und Log-Level	535
18.3.4	Prüf-Makros	536
18.3.5	Kommandozeilen-Optionen.....	540
19	Werkzeuge zur Verwaltung von Projekten	541
19.1	Dokumentation und Strukturanalyse mit doxygen.....	541
19.1.1	Strukturanalyse.....	545
19.2	Versionskontrolle	546
19.2.1	Einrichtung des Servers	548
19.2.2	Exemplarische Benutzung	550
19.3	Projektverwaltung	553
19.3.1	Projektmanagement	553
19.3.2	Wiki für Software-Entwicklungsprojekte	553
 Teil IV: Das C++-Rezeptbuch: Tipps und Lösungen für typische Aufgaben.....		555
20	Sichere Programmentwicklung.....	557
20.1	Regeln zum Design von Methoden	557
20.2	Defensive Programmierung.....	560
20.2.1	double- und float-Werte richtig vergleichen	561

20.2.2	const verwenden	561
20.2.3	Anweisungen nach for/if/while einklammern	561
20.2.4	int und unsigned/size_t nicht mischen	562
20.2.5	size_t oder auto statt unsigned int verwenden.....	562
20.2.6	Postfix++ mit Präfix++ implementieren	562
20.2.7	Ein Destruktor darf keine Exception werfen	563
20.2.8	Typumwandlungsoperatoren vermeiden	564
20.2.9	explicit-Konstruktoren bevorzugen	564
20.2.10	Leere Standardkonstruktoren vermeiden	564
20.2.11	Kopieren und Zuweisung verbieten	564
20.2.12	Vererbung verbieten	566
20.2.13	Defensiv Objekte löschen	566
20.3	Exception-sichere Beschaffung von Ressourcen.....	567
20.3.1	Sichere Verwendung von shared_ptr	567
20.3.2	shared_ptr für Arrays korrekt verwenden	567
20.3.3	unique_ptr für Arrays korrekt verwenden	568
20.3.4	Exception-sichere Funktion	569
20.3.5	Exception-sicherer Konstruktor	570
20.3.6	Exception-sichere Zuweisung	572
20.4	Aussagefähige Fehlermeldung ohne neuen String erzeugen.....	574
20.5	Empfehlungen zur Thread-Programmierung.....	575
20.5.1	Warten auf die Freigabe von Ressourcen	575
20.5.2	Deadlock-Vermeidung.....	576
20.5.3	notify_all oder notify_one?.....	576
20.5.4	Performance mit Threads verbessern?.....	577
21	Von der UML nach C++	579
21.1	Vererbung.....	580
21.2	Interface anbieten und nutzen	580
21.3	Assoziation	582
21.3.1	Aggregation.....	585
21.3.2	Komposition	585
22	Performance, Wert- und Referenzsemantik	587
22.1	Performanceproblem Wertsemantik	589
22.1.1	Auslassen der Kopie	589
22.1.2	Temporäre Objekte bei der Zuweisung	590
22.2	Optimierung durch Referenzsemantik für R-Werte.....	591

22.2.1	Bewegender Konstruktor	594
22.2.2	Bewegender Zuweisungsoperator	595
22.3	Ein effizienter binärer Plusoperator	596
22.3.1	Kopien temporärer Objekte eliminieren	597
22.3.2	Verbesserung durch verzögerte Auswertung	597
23	Effektive Programmerzeugung	601
23.1	Automatische Ermittlung von Abhängigkeiten	602
23.1.1	Getrennte Verzeichnisse: src, obj, bin	603
23.2	Makefile für Verzeichnisbäume	605
23.2.1	Rekursive Make-Aufrufe	606
23.2.2	Ein Makefile für alles	608
23.3	Automatische Erzeugung von Makefiles	609
23.3.1	Makefile für rekursive Aufrufe erzeugen	610
23.4	Erzeugen von Bibliotheken	611
23.4.1	Statische Bibliotheksmodule	612
23.4.2	Dynamische Bibliotheksmodule	613
23.5	GNU Autotools	616
23.6	CMake	619
23.7	Code Bloat bei der Instanziierung von Templates vermeiden	619
23.7.1	extern-Template	620
23.7.2	Aufspaltung in Schnittstelle und Implementation	622
24	Algorithmen für verschiedene Aufgaben	623
24.1	Algorithmen mit Strings	624
24.1.1	String splitten	624
24.1.2	String in Zahl umwandeln	625
24.1.3	Zahl in String umwandeln	629
24.1.4	Strings sprachlich richtig sortieren	629
24.1.5	Umwandlung in Klein- bzw. Großschreibung	631
24.1.6	Strings sprachlich richtig vergleichen	633
24.1.7	Von der Groß-/Kleinschreibung unabhängiger Zeichenvergleich	634
24.1.8	Von der Groß-/Kleinschreibung unabhängige Suche	635
24.2	Textverarbeitung	636
24.2.1	Datei durchsuchen	636
24.2.2	Ersetzungen in einer Datei	638
24.2.3	Code-Formatierer	640
24.2.4	Lines of Code (LOC) ermitteln	641

24.2.5	Zeilen, Wörter und Zeichen einer Datei zählen	643
24.2.6	CSV-Datei lesen	643
24.2.7	Kreuzreferenzliste	645
24.3	Operationen auf Folgen	647
24.3.1	Container anzeigen	648
24.3.2	Folge mit gleichen Werten initialisieren	648
24.3.3	Folge mit Werten eines Generators initialisieren	648
24.3.4	Folge mit fortlaufenden Werten initialisieren	649
24.3.5	Summe und Produkt	650
24.3.6	Mittelwert und Standardabweichung	651
24.3.7	Skalarprodukt	651
24.3.8	Folge der Teilsummen oder -produkte	653
24.3.9	Folge der Differenzen	653
24.3.10	Minimum und Maximum	655
24.3.11	Elemente rotieren	656
24.3.12	Elemente verwürfeln	657
24.3.13	Dubletten entfernen	658
24.3.14	Reihenfolge umdrehen	660
24.3.15	Anzahl der Elemente, die einer Bedingung genügen	661
24.3.16	Gilt X für alle, keins oder wenigstens ein Element einer Folge?	662
24.3.17	Permutationen	663
24.3.18	Lexikografischer Vergleich	665
24.4	Sortieren und Verwandtes	666
24.4.1	Partitionieren	666
24.4.2	Sortieren	667
24.4.3	Stabiles Sortieren	667
24.4.4	Partielles Sortieren	669
24.4.5	Das n.-größte oder n.-kleinste Element finden	669
24.4.6	Verschmelzen (merge)	671
24.5	Suchen und Finden	674
24.5.1	Element finden	674
24.5.2	Element einer Menge in der Folge finden	675
24.5.3	Teilfolge finden	677
24.5.4	Bestimmte benachbarte Elemente finden	679
24.5.5	Bestimmte aufeinanderfolgende Werte finden	680
24.5.6	Binäre Suche	681
24.6	Mengenoperationen auf sortierten Strukturen	684

24.6.1	Teilmengenrelation	684
24.6.2	Vereinigung	685
24.6.3	Schnittmenge	686
24.6.4	Differenz	686
24.6.5	Symmetrische Differenz	687
24.7	Heap-Algorithmen	688
24.7.1	pop_heap	689
24.7.2	push_heap	690
24.7.3	make_heap	691
24.7.4	sort_heap	691
24.7.5	is_heap	692
24.8	Vergleich von Containern auch ungleichen Typs	692
24.8.1	Unterschiedliche Elemente finden	692
24.8.2	Prüfung auf gleiche Inhalte	694
24.9	Rechnen mit komplexen Zahlen: Der C++-Standardtyp complex	695
24.10	Schnelle zweidimensionale Matrix	697
24.10.1	Optimierung mathematischer Array-Operationen	701
24.11	Singleton	705
24.11.1	Implementierung mit einem Zeiger	706
24.11.2	Implementierung mit einer Referenz	706
24.11.3	Meyers' Singleton	707
24.12	Vermischtes	710
24.12.1	Erkennung eines Datums	710
24.12.2	Erkennung einer IP-Adresse	712
24.12.3	Erzeugen von Zufallszahlen	712
24.12.4	for_each – Auf jedem Element eine Funktion ausführen	716
24.12.5	Verschiedene Möglichkeiten, Container-Bereiche zu kopieren	717
24.12.6	Vertauschen von Elementen, Bereichen und Containern	719
24.12.7	Elemente transformieren	720
24.12.8	Ersetzen und Varianten	722
24.12.9	Elemente herausfiltern	723
24.12.10	Grenzwerte von Zahltypen	725
24.12.11	Minimum und Maximum	726
25	Ein- und Ausgabe	727
25.1	Datei- und Verzeichnisoperationen	727
25.1.1	Datei oder Verzeichnis löschen	728

25.1.2	Datei oder Verzeichnis umbenennen	729
25.1.3	Verzeichnis anlegen.....	730
25.1.4	Verzeichnis anzeigen	731
25.1.5	Verzeichnisbaum anzeigen.....	732
25.2	Tabelle formatiert ausgeben	734
25.3	Formatierte Daten lesen	735
25.3.1	Eingabe benutzerdefinierter Typen	735
25.4	Array als Block lesen oder schreiben.....	737
Teil V: Die C++-Standardbibliothek		739
26	Aufbau und Übersicht	741
26.1	Auslassungen.....	743
26.2	Beispiele des Buchs und die C++-Standardbibliothek	745
27	Hilfsfunktionen und -klassen	747
27.1	Relationale Operatoren	747
27.2	Unterstützung der Referenzsemantik für R-Werte.....	748
27.3	Paare.....	750
27.4	Tupel.....	752
27.5	Funktionsobjekte.....	753
27.5.1	Arithmetische, vergleichende und logische Operationen	753
27.5.2	Funktionsobjekte zum Negieren logischer Prädikate.....	753
27.5.3	Binden von Argumentwerten.....	754
27.5.4	Funktionen in Objekte umwandeln.....	756
27.6	Templates für rationale Zahlen.....	758
27.7	Zeit und Dauer	760
27.8	Hüllklasse für Referenzen.....	761
28	Container	763
28.1	Gemeinsame Eigenschaften.....	765
28.1.1	Initialisierungslisten	767
28.1.2	Konstruktion an Ort und Stelle.....	768
28.1.3	Reversible Container	768
28.2	Sequenzen	769
28.2.1	vector	770
28.2.2	vector<bool>	771
28.2.3	list.....	772

28.2.4	deque	775
28.2.5	stack	776
28.2.6	queue	777
28.2.7	priority_queue	778
28.2.8	array	780
28.3	Sortierte assoziative Container	782
28.3.1	map	782
28.3.2	multimap	787
28.3.3	set	787
28.3.4	multiset	791
28.4	Hash-Container	791
28.4.1	unordered_map	793
28.4.2	unordered_multimap	798
28.4.3	unordered_set	798
28.4.4	unordered_multiset	801
28.5	bitset	801
29	Iteratoren	805
29.1	Iterator-Kategorien	806
29.1.1	Anwendung von Traits	808
29.2	distance() und advance()	810
29.3	Reverse-Iteratoren	811
29.4	Insert-Iteratoren	812
29.5	Stream-Iteratoren	813
30	Algorithmen	815
30.1	Algorithmen mit Prädikat	816
30.1.1	Algorithmen mit binärem Prädikat	816
30.2	Übersicht	817
31	Nationale Besonderheiten	821
31.1	Sprachumgebungen festlegen und ändern	822
31.1.1	Die locale-Funktionen	823
31.2	Zeichensätze und -codierung	825
31.3	Zeichenklassifizierung und -umwandlung	829
31.4	Kategorien	829
31.4.1	collate	830
31.4.2	ctype	831

31.4.3	numeric	832
31.4.4	monetary	833
31.4.5	time	836
31.4.6	messages	839
31.5	Konstruktion eigener Facetten	839
32	String	841
33	Speichermanagement	849
33.1	Smart Pointer unique_ptr, shared_ptr, weak_ptr	849
33.2	new mit vorgegebenem Speicherort	854
33.3	Hilfsfunktionen	855
34	Optimierte numerische Arrays (valarray)	857
34.1	Konstruktoren	858
34.2	Elementfunktionen	858
34.3	Binäre Valarray-Operatoren	861
34.4	Mathematische Funktionen	863
34.5	slice und slice_array	864
34.6	gslice und gslice_array	866
34.7	mask_array	869
34.8	indirect_array	870
35	C-Header	873
35.1	<cassert>	874
35.2	<cctype>	874
35.3	<cerrno>	875
35.4	<cmath>	875
35.5	<cstdarg>	876
35.6	<cstddef>	877
35.7	<cstdio>	877
35.8	<cstdlib>	877
35.9	<cstring>	879
35.10	<ctime>	881
A	Anhang	883
A.1	Programmierhinweise	883
A.2	C++-Schlüsselwörter	887
A.3	ASCII-Tabelle	887

A.4	Rangfolge der Operatoren	890
A.5	Compilerbefehle	891
A.6	Lösungen zu den Übungsaufgaben	892
A.7	Installation der DVD-Software für Windows.....	937
A.7.1	Installation des Compilers und der Entwicklungsumgebung.....	937
A.7.2	Installation der Boost-Bibliothek	937
A.7.3	Installation von Qt	938
A.7.4	Codeblocks einrichten	938
A.7.5	Integration von Qt in ein Code::Blocks-Projekt.....	940
A.7.6	Bei Verzicht auf die automatische Installation.....	941
A.8	Installation der DVD-Software für Linux.....	942
A.8.1	Installation des Compilers	942
A.8.2	Installation von Boost	943
A.8.3	Installation von Code::Blocks	944
A.8.4	Code::Blocks einrichten	945
A.8.5	Beispieldateien entpacken	946
A.8.6	Installation von Qt4	946
A.8.7	Integration von Qt in ein Code::Blocks-Projekt.....	947
Glossar		949
Literaturverzeichnis		959
Register		963

Vorwort

Die zweite Auflage dieses Buchs unterscheidet sich im Wesentlichen durch Aktualisierungen im Verlauf der Entwicklung des ISO C++-Standards. Das Buch ist konform zum neuen C++-Standard, ohne den Anspruch auf Vollständigkeit zu erheben – das Dokument [\[ISOC++\]](#) umfasst mehr als 1300 Seiten. Das Buch gibt eine kompetente Einführung in die Sprache und im Teil »Das C++ Rezeptbuch« zahlreiche Tipps und Lösungen für typische Aufgaben, die in der täglichen Praxis anfallen. Es gibt konkrete, sofort umsetzbare Lösungsvorschläge zur defensiven Programmierung, zur exception-sicheren Programmierung, zur Vermeidung von Memory-Problemen, zur Performance-Verbesserung und zur automatisierten Programm- und Bibliothekserzeugung. Zahlreiche Algorithmen für praxisnahe Problemstellungen helfen bei der täglichen Arbeit. Auf größtmögliche Portabilität wird geachtet: Die Beispiele funktionieren unter Linux genau so wie unter Windows. Die problembezogene Orientierung lässt die in die Sprache einführenden Teile kürzer werden. Damit wird das Lernen erleichtert, und die Qualität des Buchs als Nachschlagewerk bleibt erhalten.

Für wen ist dieses Buch geschrieben?

Dieses Buch ist für alle geschrieben, die einen kompakten und gleichzeitig gründlichen Einstieg in die Konzepte und Programmierung mit C++ suchen. Es ist für Anfänger¹ gedacht, die noch keine Programmiererfahrung haben, aber auch für Programmierer, die diese Programmiersprache kennen lernen möchten. Beiden Gruppen und auch C++-Erfahrenen dient das Buch als ausführliches Nachschlagewerk.

Die ersten 11 Kapitel führen in die Sprache ein. Es wird sehr schnell ein Verständnis des objektorientierten Ansatzes entwickelt. Die sofortige praktische Umsetzung des Gelernten steht im Vordergrund. C++ wird als Programmiersprache unabhängig von speziellen Produkten beschrieben; C-Kenntnisse werden nicht vorausgesetzt. Das Buch eignet sich zum Selbststudium und als Begleitbuch zu einer Vorlesung oder zu Kursen. Die vielen Beispiele sind leicht nachzuvollziehen und praxisnah umsetzbar. Klassen und Objekte,

¹ Geschlechtsbezogene Formen meinen hier und im Folgenden stets Männer *und* Frauen.

Templates, STL und Exceptions sind Ihnen bald keine Fremdworte mehr. Es gibt mehr als 85 Übungsaufgaben – mit Musterlösungen im Anhang. Durch das Studium dieser Kapitel werden aus Anfängern bald Fortgeschrittene.

Diesen und anderen Fortgeschrittenen und Profis bietet das Buch eine Einführung in die Themen Thread-Programmierung, Netzwerk-Programmierung mit Sockets einschließlich eines kleinen Webservers, Datenbankanbindung, grafische Benutzungsoberflächen und mehr. Dabei wird durch Einsatz der Boost-Library und des Qt-Frameworks größtmögliche Portabilität erreicht.

Softwareentwicklung ist nicht nur Programmierung: Einführend werden anhand von Beispielen unter anderem die Automatisierung der Programmerzeugung mit Make, die Dokumentationserstellung mit Doxygen und die Versionskontrolle mit Subversion behandelt. Das Programmdesign wird durch konkrete Umsetzungen von UML-Mustern nach C++ unterstützt. Das integrierte »C++-Rezeptbuch« mit mehr als 150 praktischen Lösungen, das sehr umfangreiche Register und das detaillierte Inhaltsverzeichnis machen das Buch zu einem praktischen Nachschlagewerk für alle, die sich mit der Softwareentwicklung in C++ beschäftigen.

Übersicht

Schwerpunkt von Teil I ist die Einführung in die Programmiersprache. Die anschließenden Teile gehen darüber hinaus und konzentrieren sich auf die verschiedenen Probleme der täglichen Praxis.

Teil I - Einführung in C++

Das *Kapitel 1* vermittelt zunächst die Grundlagen, wie ein Programm geschrieben und zum Laufen gebracht wird. Es folgen einfache Datentypen und Anweisungen zur Kontrolle des Programmablaufs. Die Einführung der C++-Datentypen `vector` und `string` beendet das Kapitel. *Kapitel 2* beschäftigt sich mit der einfachen Ein- und Ausgabe, auch mit Dateien. Das *Kapitel 3* zeigt Ihnen, wie Sie Funktionen schreiben. Makros, Templates für Funktionen und die modulare Gestaltung von Programmen folgen.

Objektorientierung ist der Schwerpunkt von *Kapitel 4*. Dabei geht es nicht nur um die Konstruktion von Objekten, sondern auch um den Weg von der Problemstellung zu Klassen und Objekten. Zeiger, einfache Arrays (C-Arrays) und Zeichenketten sowie die Erzeugung von Objekten zur Laufzeit sind Inhalt von *Kapitel 5*. Dazu kommen mehrdimensionale C-Arrays und das Schreiben und Lesen von Binärdaten in bzw. aus Dateien. Auf der Basis von Zeigern führt *Kapitel 6* das Thema Objektorientierung fort. Dabei lernen Sie kennen, wie eine String-Klasse funktioniert, und wie Sie Klassen-Templates und Templates mit einer variablen Anzahl von Parametern konstruieren. Das *Kapitel 7* zeigt Ihnen das Mittel objektorientierter Sprachen, um Generalisierungs- und Spezialisierungsbeziehungen auszudrücken: die Vererbung mit ihren Möglichkeiten. Strategien zur Fehlerbehandlung mit Exceptions finden Sie in *Kapitel 8*. Das *Kapitel 9* zeigt, wie Sie Operatorsymbolen wie `+` und `-` eigene Bedeutungen zuweisen können und in welchem Zusammenhang das sinnvoll ist. Sie lernen, »intelligente« Zeiger (Smart Pointer) zu konstruieren und Objekte als Funktionen einzusetzen. *Kapitel 10* beschreibt ausführlich die Ein- und Ausgabemöglichkeiten, die in Kapitel 2 nur einführend gestreift werden, einschließlich der Fehlerbehandlung und der Formatierung der Ausgabe. Eine Einführung

in die Standard Template Library (STL) bietet *Kapitel 11*. Es zeigt, wie die Komponenten (Container, Iteratoren und Algorithmen) zusammenwirken. Die STL und ihre Wirkungsweise bilden die Grundlage eines sehr großen Teils der C++-Standardbibliothek.

Die Kapitel 1 bis 11 sind für ein gutes Verständnis von C++ unverzichtbar. Reguläre Ausdrücke (*Kapitel 12*) und die Programmierung paralleler Abläufe mit Threads (*Kapitel 13*) sind dazu nicht notwendig – hier handelt es sich um Ergänzungen, wie sie von vielen Programmiersprachen angeboten werden.

Teil II - Bausteine komplexer Anwendungen

Ein Programm benötigt eine Möglichkeit, mit der Außenwelt zu kommunizieren. Tastatur und Konsole allein reichen für komplexe Anwendungen in der Regel nicht aus. Mausbedienung und Bildschirmgrafik sind heute Standard bei Desktop-Anwendungen. Das *Kapitel 14* zeigt, wie grafische Benutzungsschnittstellen konstruiert werden. Wie ein Programm die Verbindung mit dem Internet aufnehmen kann, dokumentiert das *Kapitel 15*. Und wohin mit den ganzen Daten, die bei Programmende nicht verloren gehen sollen? In *Kapitel 16* lernen Sie, wie ein Programm an eine Datenbank angebunden wird. Die genannten Themen sind so umfangreich, dass sie selbst Bücher füllen. Dieser Teil bietet Ihnen daher nur einen Einstieg.

Teil III - Praktische Methoden und Werkzeuge der Softwareentwicklung

Die Entwicklung von Programmen besteht nicht nur im Schreiben von Code. Die Compilation eines Projekts mit vielen Programmdateien und Abhängigkeiten kann schnell ein komplexer Vorgang werden. Die Automatisierung dieses Prozesses mit dem Tool *make* ist Thema von *Kapitel 17*. Programme sind nicht auf Anhieb fehlerfrei. Sie müssen getestet werden: *Kapitel 18* stellt ein Werkzeug für den Unit-Test vor und zeigt den praktischen Einsatz. *Kapitel 19* demonstriert ein Werkzeug zur automatischen Dokumentationserstellung, zeigt, wie eine Versionsverwaltung eingerichtet wird, und geht kurz auf die Projektverwaltung ein.

Teil IV - Das C++-Rezeptbuch: Tipps und Lösungen für typische Aufgaben

Sichere Programmentwicklung ist die Überschrift des *Kapitels 20*. Sie finden dort Regeln zum Design von Methoden und mehrere Tipps zur defensiven Programmierung, die die Risiken falscher Algorithmen oder falscher Benutzung vermindern. Auch gibt es Tipps zur exception-sicheren Beschaffung von Speicher und zur Thread-Programmierung. *Kapitel 21* zeigt Rezepte, wie Sie bestimmte UML-Muster in C++-Konstruktionen umwandeln können. *Kapitel 22* erklärt den Unterschied zwischen Wert- und Referenzsemantik und die Auswirkung auf die Geschwindigkeit von C++-Programmen. Es werden Empfehlungen gegeben und am Beispiel konkretisiert, wie die Performanz deutlich verbessert werden kann. *Kapitel 23* erweitert die Grundlagen des Kapitels 17 um praktische Rezepte zur automatischen Ermittlung von Abhängigkeiten zwischen Programmdateien, Makefiles für Verzeichnisbäume, die automatische Erzeugung von Makefiles und statischer und dynamischer Bibliotheken. Algorithmen für viele verschiedene Aufgaben finden Sie in *Kapitel 24*. Wegen der Vielzahl empfiehlt sich ein Blick in das Inhaltsverzeichnis, um einen Überblick zu gewinnen. Der C++-Standard bietet für viele Datei- und Verzeichnisoperationen keine Unterstützung an. *Kapitel 25* enthält fertige Rezepte zum Anlegen,

Löschen und Lesen von Verzeichnissen und mehr auf der Basis der Boost-Library. Ergänzt wird dies durch das formatierte Lesen und Schreiben von Daten und die Abspeicherung binärer Daten als Block.

Teil V - Die C++-Standardbibliothek

In mehreren Kapiteln wird die C++-Standardbibliothek in Kürze beschrieben. Die Inhalte dieses Teils sind: Hilfsfunktionen und -klassen, Container, Iteratoren, Algorithmen, Einstellung nationaler Besonderheiten, String, Speicherverwaltung, Funktionen der Programmiersprache C.

Anhang

Der Anhang enthält unter anderem Empfehlungen zur Programmierung, verschiedene hilfreiche Tabellen und die Lösungen der Übungsaufgaben.



Wo finden Sie was?

Bei der Programmentwicklung wird häufig das Problem auftauchen, etwas nachschlagen zu müssen. Es gibt die folgenden Hilfen:

Erklärungen zu Begriffen sind im *Glossar* ab Seite 949 aufgeführt.

Es gibt ein recht umfangreiches *Stichwortverzeichnis* ab Seite 963 und ein sehr detailliertes *Inhaltsverzeichnis*.

Auf der Webseite <http://www.cppbuch.de/> finden Sie weiteres Material, Hinweise, Errata und nützliche Links.



Zu guter Letzt

Allen Menschen, die dieses Buch durch Hinweise und Anregungen verbessern halfen, sei an dieser Stelle herzlich gedankt. Frau Margarete Metzger und Frau Irene Weihart vom Hanser Verlag danke ich für die sehr gute Zusammenarbeit.

Bremen, im Juni 2011

Ulrich Breymann



Teil I: Einführung in C++

1

Es geht los!

Dieses Kapitel behandelt die folgenden Themen:

- Entstehung und Entwicklung der Programmiersprache C++
- Objektorientierte Programmierung - Erste Grundlagen
- Wie schreibe ich ein Programm und bringe es zum Laufen?
- Einfache Datentypen und Operationen
- Ablauf innerhalb eines Programms steuern
- Erste Definition eigener Datentypen
- Standarddatentypen vector und string

1.1 Historisches

C++ wurde etwa ab 1980 von Bjarne Stroustrup als Sprache, die Objektorientierung stark unterstützt, entwickelt und 1998 von der ISO (International Standards Organisation) und der IEC (International Electrotechnical Commission) standardisiert. Diesem Standard haben sich nationale Standardisierungsgremien wie ANSI (USA) und DIN (Deutschland) angeschlossen. Mittlerweile ist der C++-Standard, zu dem es 2003 einige Korrekturen gab, in die Jahre gekommen. Die Anforderungen an C++ sind gewachsen, auch zeigte sich, dass manches im Standard fehlte und anderes überflüssig oder fehlerhaft war. Das C++-Standardkomitee hat kontinuierlich an der Entwicklung des nächsten C++-Standards gearbeitet. Der Arbeitsname war C++0x, weil es anfangs die feste Absicht des Standardkomitees war, den neuen C++-Standard spätestens 2009 zu verabschieden. Der Standardisierungsprozess dauerte dann doch zwei Jahre länger als geplant.

2006 wurde beschlossen, den TR1 (Technical Report 1 [TR1]) mit der Ausnahme spezieller mathematischer Funktionen in den Standard aufzunehmen. Der TR1 besteht aus teilweise erheblichen Erweiterungen der C++-Standardbibliothek. Der TR1 wurde von Boost [boost] entwickelt, einer Community, die von Mitgliedern der »C++ Standards Committee Library Working Group« gegründet wurde und an der sich heute Tausende von Entwicklern beteiligen. Bereits vor der Integration in den C++-Standard haben die Boost-Bibliotheken vielfachen Einsatz gefunden. Weitere Erweiterungen sind in Arbeit. Im März 2010 wurde der »Final Committee Draft« veröffentlicht und zur Abstimmung gestellt. Im April 2011 wurde der »Final Draft International Standard (FDIS)« [ISOC++] publiziert, der Ende 2011 in den endgültigen Standard mündet. ISO-Standards sind kostenpflichtig. Wegen des jahrelangen Abstimmungsprozesses wird sich der endgültige Standard nur minimal vom FDIS unterscheiden. Deshalb verweise ich in diesem Buch auf das per Internet zugängliche Dokument [ISOC++].

1.2 Objektorientierte Programmierung

Nach üblicher Auffassung heißt Programmieren, einem Rechner mitzuteilen, *was* er tun soll und *wie* es zu tun ist. Ein Programm ist ein in einer Programmiersprache formulierter Algorithmus oder, anders ausgedrückt, eine Folge von Anweisungen, die der Reihe nach auszuführen sind, ähnlich einem Kochrezept, geschrieben in einer besonderen Sprache, die der Rechner »versteht«. Der Schwerpunkt dieser Betrachtungsweise liegt auf den einzelnen Schritten oder Anweisungen an den Rechner, die zur Lösung einer Aufgabe abzuarbeiten sind.

Was fehlt hier beziehungsweise wird bei dieser Sicht eher stiefmütterlich behandelt? Der Rechner muss »wissen«, *womit* er etwas tun soll. Zum Beispiel soll er

- eine bestimmte Summe Geld von einem Konto auf ein anderes transferieren;
- eine Ampelanlage steuern;
- ein Rechteck auf dem Bildschirm zeichnen.

Häufig, wie in den ersten beiden Fällen, werden Objekte der realen Welt (Konten, Ampelanlage ...) *simuliert*, das heißt im Rechner abgebildet. Die abgebildeten Objekte haben eine *Identität*. Das *Was* und das *Womit* gehören stets zusammen. Beide sind also Eigenschaften eines Objekts und sollten besser nicht getrennt werden. Ein Konto kann schließlich nicht auf Gelb geschaltet werden, und eine Überweisung an eine Ampel ist nicht vorstellbar.

Ein *objektorientiertes Programm* kann man sich als Abbildung von Objekten der realen Welt in Software vorstellen. Die Abbildungen werden selbst wieder Objekte genannt. Klassen sind Beschreibungen von Objekten. Die *objektorientierte Programmierung* berücksichtigt besonders die Kapselung von Daten und den darauf ausführbaren Funktionen sowie die Wiederverwendbarkeit von Software und die Übertragung von Eigenschaften von Klassen auf andere Klassen, Vererbung genannt. Auf die einzelnen Begriffe wird noch eingegangen.

Das Motiv hinter der objektorientierten Programmierung ist die rationelle und vor allem ingenieurmäßige Softwareentwicklung. Unter »Softwarekrise« wird das Phänomen verstanden, dass viele Softwareprojekte nicht im geplanten Zeitraum fertig werden, dass das finanzielle Budget überschritten wird oder die Projekte ganz abgebrochen werden. Die objektorientierte Programmierung wird als ein Mittel angesehen, das zur Lösung der »Softwarekrise« beitragen kann.

Wiederverwendung heißt, Zeit und Geld zu sparen, indem bekannte Klassen wiederverwendet werden. Das Leitprinzip ist hier, das Rad nicht mehrfach neu zu erfinden! Unter anderem durch den Vererbungsmechanismus kann man Eigenschaften von bekannten Objekten ausnutzen. Zum Beispiel sei Konto eine bekannte Objektbeschreibung mit den »Eigenschaften« Inhaber, Kontonummer, Betrag, Dispo-Zinssatz und so weiter. In einem Programm für eine Bank kann nun eine Klasse Waehrungskonto entworfen werden, für die alle Eigenschaften von Konto übernommen (= geerbt) werden könnten. Zusätzlich wäre nur noch die Eigenschaft »Währung« hinzuzufügen.

Wie Computer können auch Objekte Anweisungen ausführen. Wir müssen ihnen nur »erzählen«, was sie tun sollen, indem wir ihnen eine *Aufforderung* oder *Anweisung* senden, die in einem Programm formuliert wird. Anstelle der Begriffe »Aufforderung« oder »Anweisung« wird in der Literatur manchmal *Botschaft* (englisch *message*) verwendet, was jedoch den Aufforderungscharakter nicht zur Geltung bringt. Eine gängige Notation (= Schreibweise) für solche Aufforderungen ist *Objektname.Anweisung* (gegebenenfalls *Daten*). Beispiele:

```
dieAmpel.blinken(gelb);  
dieAmpel.ausschalten(); // keine Daten notwendig!  
dieAmpel.einschalten(gruen);  
dasRechteck.zeichnen(position, hoehe, breite);  
dasRechteck.verschieben(5.0); // Daten in cm
```

Die Beispiele geben schon einen Hinweis, dass die Objektorientierung uns ein Hilfsmittel zur Modellierung der realen Welt in die Hand gibt.

Klassen

Es muss unterschieden werden zwischen der *Beschreibung* von Objekten und den *Objekten selbst*. Die Beschreibung besteht aus Attributen und Operationen. Attribute bestehen aus einem Namen und Angaben zum Datenformat der Attributwerte. Eine Kontobeschreibung könnte so aussehen:

Attribute:

Inhaber: Folge von Buchstaben
Kontonummer: Zahl
Betrag: Zahl
Dispo-Zinssatz in %: Zahl

Operationen:

überweisen (Ziel-Kontonummer, Betrag)
abheben(Betrag)
einzahlen(Betrag)

Eine Aufforderung ist nichts anderes als der Aufruf einer Operation, die auch Methode genannt wird. Ein *tatsächliches* Konto k1 enthält *konkrete* Daten, also Attributwerte, deren Format mit dem der Beschreibung übereinstimmen muss. Die Tabelle zeigt k1 und ein weiteres Konto k2.

Tabelle 1.1: Attribute und Werte zweier Konten

Attribut	Wert für Konto k1	Wert für Konto k2
Inhaber	Roberts, Julia	Depp, Johnny
Kontonummer	12573001	54688490
Betrag	-200,30 €	1222,88 €
Dispo-Zinssatz	13,75 %	13,75 %

Julia will Johnny 1000 € überweisen. Dem Objekt k1 wird also der Auftrag mit den benötigten Daten mitgeteilt:

k1.überweisen(54688490, 1000.00).

Johnny will 22 € abheben. Die Aufforderung wird an k2 gesendet:

k2.abheben(22).

Es scheint natürlich etwas merkwürdig, wenn einem Konto ein Auftrag gegeben wird. In der objektorientierten Programmierung werden Objekte als Handelnde aufgefasst, die auf Anforderung selbstständig einen Auftrag ausführen, entfernt vergleichbar einem Sachbearbeiter in einer Firma, der seine eigenen Daten verwaltet und mit anderen Sachbearbeitern kommuniziert, um eine Aufgabe zu lösen.

- Die *Beschreibung* eines tatsächlichen Objekts gibt seine innere *Datenstruktur* und die möglichen *Operationen* oder *Methoden* an, die auf die inneren Daten anwendbar sind.
- Zu *einer* Beschreibung kann es kein, ein oder beliebig viele Objekte geben.

Die Beschreibung eines Objekts in der objektorientierten Programmierung heißt *Klasse*. Die tatsächlichen Objekte heißen auch *Instanzen* einer Klasse.

Auf die inneren Daten eines Objekts nur mithilfe der vordefinierten Methoden zuzugreifen, dient der Sicherheit der Daten und ist ein allgemein anerkanntes Prinzip. Das Prinzip wird *Datenabstraktion* oder Geheimnisprinzip genannt. Der Softwareentwickler, der die Methoden konstruiert hat, weiß ja, wie die Daten konsistent (das heißt widerspruchsfrei) bleiben und welche Aktivitäten mit Datenänderungen verbunden sein müssen. Zum Beispiel muss eine Erhöhung des Kontostands mit einer Gutschrift oder einer Einzahlung verbunden sein. Außerdem wird jeder Buchungsvorgang protokolliert. Es darf nicht möglich sein, dass jemand anders die Methoden umgeht und direkt und ohne Protokoll seinen eigenen Kontostand erhöht. Wenn Sie die Unterlagen eines Kollegen haben möchten, greifen Sie auch nicht einfach in seinen Schreibtisch, sondern Sie bitten ihn darum (= Sie senden ihm eine Aufforderung), dass er sie Ihnen gibt.

Die hier verwendete Definition einer Klasse als Beschreibung der Eigenschaften einer Menge von Objekten wird im Folgenden beibehalten. Gelegentlich findet man in der Literatur andere Definitionen, auf die hier nicht weiter eingegangen wird. Weitere Informationen zur Objektorientierung sind in Kapitel 4 und in der Literatur zu finden, zum Beispiel in [Bal].

1.3 Compiler

Compiler sind die Programme, die Ihren Programmtext in eine für den Computer verarbeitbare Form übersetzen. Von Menschen geschriebener und für Menschen lesbarer Programmtext kann nicht vom Computer »verstanden« werden. Das vom Compiler erzeugte Ergebnis der Übersetzung kann der Computer aber ausführen. Das Erlernen einer Programmiersprache ohne eigenes praktisches Ausprobieren ist kaum sinnvoll. Deshalb sollten Sie die Dienste des Compilers möglichst bald anhand der Beispiele nutzen – wie, zeigt Ihnen der Abschnitt direkt nach der Vorstellung des ersten Programms. Falls Sie nicht schon einen C++-Compiler oder ein C++-Entwicklungssystem haben, um die Beispiele korrekt zu übersetzen, bietet sich die Benutzung der in Abschnitt 1.5 beschriebenen Entwicklungsumgebungen an. Ein viel verwendeter Compiler ist der GNU¹ C++-Compiler [GCC]. Entwicklungsumgebung und Compiler sind kostenlos erhältlich. Ein Installationsprogramm dafür finden Sie auf der DVD zum Buch.

1.4 Das erste Programm

Sie lernen hier die Entwicklung eines ganz einfachen Programms kennen. Dabei wird Ihnen zunächst das Programm vorgestellt, und wenige Seiten später erfahren Sie, wie Sie es eingeben und zum Laufen bringen können. Der erste Schritt besteht in der Formulierung der Aufgabe. Sie lautet: »Lies zwei Zahlen a und b von der Tastatur ein. Berechne die Summe beider Zahlen und zeige das Ergebnis auf dem Bildschirm an.« Die Aufgabe ist so einfach, wie sie sich anhört! Im zweiten Schritt wird die Aufgabe in die Teilaufgaben »Eingabe«, »Berechnung« und »Ausgabe« zerlegt:

```
int main() {           // Noch tut dieses Programm nichts!
    // Lies die Zahlen a und b ein
    /* Berechne die Summe beider
       Zahlen */
    // Zeige das Ergebnis auf dem Bildschirm an
}
```

Hier sehen Sie schon ein einfaches C++-Programm. Es bedeuten:

int	ganze Zahl zur Rückgabe
main	Schlüsselwort für Hauptprogramm
()	Innerhalb dieser Klammern können dem Hauptprogramm Informationen mitgegeben werden.
{ }	Block
/* ... */	Kommentar, der über mehrere Zeilen gehen kann
// ...	Kommentar bis Zeilenende

¹ Siehe Glossar Seite 952

Ein durch { und } begrenzter *Block* enthält die Anweisungen an den Rechner. Der *Compiler* übersetzt den Programmtext in eine rechnerverständliche Form. Im obigen Programm sind lediglich *Kommentare* enthalten und noch keine Anweisungen an den Computer, so dass unser Programm (noch) nichts tut.

Kommentare werden einschließlich der Kennungen vom Compiler vollständig ignoriert. Ein Kommentar, der mit /* beginnt, ist erst mit */ beendet, auch wenn er sich über mehrere Zeilen erstreckt. Ein mit // beginnender Kommentar endet am Ende der Zeile. Auch wenn Kommentare vom Compiler ignoriert werden, sind sie doch sinnvoll für den menschlichen Leser eines Programms, um ihm die Anweisungen zu erläutern, zum Beispiel für den Programmierer, der Ihr Nachfolger wird, weil Sie befördert worden sind oder die Firma verlassen haben.

Kommentare sind auch wichtig für den Autor eines Programms, der nach einem halben Jahr nicht mehr weiß, warum er gerade diese oder jene komplizierte Anweisung geschrieben hat. Sie sehen:

Ein Programm ist »nur« ein Text!

- Der Text hat eine Struktur entsprechend den C++-Sprachregeln: Es gibt Wörter wie hier das Schlüsselwort `main` (in C++ werden alle Schlüsselwörter kleingeschrieben). Es gibt weiterhin Zeilen, Satzzeichen und Kommentare.
- Die Bedeutung des Textes wird durch die Zeilenstruktur nicht beeinflusst. Mit \ und nachfolgendem `ENTER` ist eine Worttrennung am Zeilenende möglich. Das Zeichen \ wird »Backslash« genannt. Mit dem Symbol `ENTER` ist hier und im Folgenden die Betätigung der großen Taste  rechts auf der Tastatur gemeint.
- Groß- und Kleinschreibung werden unterschieden! `main()` ist nicht dasselbe wie `Main()`.

Weil die Zeilenstruktur für den Rechner keine Rolle spielt, kann der Programmtext nach Gesichtspunkten der Lesbarkeit gestaltet werden. Im dritten Schritt müssen »nur« noch die Inhalte der Kommentare als C++-Anweisungen formuliert werden. Dabei bleiben die Kommentare zur Dokumentation stehen, wie im Beispielprogramm auf der nächsten Seite zu sehen ist. Es sind einige neue Worte dazugekommen, die hier kurz erklärt werden. Machen Sie sich keine Sorgen, wenn Sie nicht alles auf Anhieb verstehen! Alles wird im Verlauf des Buchs wieder aufgegriffen und vertieft. Wie das Programm zum Laufen gebracht wird, erfahren Sie nur wenige Seiten später (Seite 36).

`#include<iostream>` Einbindung der Ein-/Ausgabefunktionen. Diese Zeile muss in jedem Programm stehen, das Eingaben von der Tastatur erwartet oder Ausgaben auf den Bildschirm bringt. Man kann sich vorstellen, dass der Compiler beim Übersetzen des Programms an dieser Stelle erst alle zur Ein- und Ausgabe notwendigen Informationen liest. Details folgen in Abschnitt 3.3.5.

`using namespace std;` Der Namensraum `std` wird benutzt. Schreiben Sie es einfach in jedes Programm an diese Stelle und haben Sie Geduld: Eine genauere Erklärung folgt später (Seiten 60 und 141).

Listing 1.1: Summe zweier Zahlen berechnen

```
// cppbuch/k1/summe.cpp
// Hinweis: Alle Programmbeispiele sind von der Internet-Seite zum Buch herunterladbar
// (http://www.cppbuch.de/).
// Die erste Zeile in den Programmbeispielen gibt den zugehörigen Dateinamen an.
#include<iostream>
using namespace std;

// Programm zur Berechnung der Summe zweier Zahlen
int main() {
    int summe;
    int a;
    int b;

    // Lies die Zahlen a und b ein
    cout << "a und b eingeben:";
    cin >> a >> b;

    /* Berechne die Summe beider Zahlen */
    summe = a + b;

    // Zeige das Ergebnis auf dem Bildschirm an
    cout << "Summe=" << summe;
    return 0;
}
```

```
int main()
```

`main()` ist das Hauptprogramm (es gibt auch Unterprogramme). Der zu `main()` gehörende Programmcode wird durch die geschweiften Klammern `{` und `}` eingeschlossen. Ein mit `{` und `}` begrenzter Bereich heißt *Block*. Mit `int` ist gemeint, dass das Programm `main()` nach Beendigung eine Zahl vom Typ `int` (= ganze Zahl) an das Betriebssystem zurückgibt. Dazu dient die unten beschriebene `return`-Anweisung. Normalerweise – das heißt bei ordnungsgemäßem Programmablauf – wird die Zahl 0 zurückgegeben. Andere Zahlen könnten verwendet werden, um über das Betriebssystem einem nachfolgenden Programm einen Fehler zu signalisieren.

```
int summe;
int a;
int b;
```

Deklaration von Objekten: Mitteilung an den Compiler, der entsprechend Speicherplatz bereitstellt und ab jetzt die Namen `summe`, `a` und `b` innerhalb des Blocks `{ }` kennt. Es gibt verschiedene Zahlentypen in C++. Mit `int` sind ganze Zahlen gemeint: `summe`, `a`, `b` sind ganze Zahlen.

```
;
```

Ein Semikolon beendet jede Deklaration und jede Anweisung (aber keine Verbundanweisung, siehe später).

```
cout
```

Ausgabe: `cout` (Abkürzung für *character out*) ist die Standardausgabe. Der Doppelpfeil deutet an, dass alles, was rechts davon steht, zur Ausgabe `cout` gesendet wird, zum Beispiel `cout << a;`. Wenn mehrere Dinge ausgegeben werden sollen, sind sie durch `<<` zu trennen.

<code>cin</code>	Eingabe: Der Doppelpfeil zeigt hier in Richtung des Objekts, das ja von der Tastatur einen neuen Wert aufnehmen soll. Die Information fließt von der Eingabe <code>cin</code> zum Objekt <code>a</code> beziehungsweise zum Objekt <code>b</code> .
<code>=</code>	Zuweisung: Der Variablen auf der linken Seite des Gleichheitszeichens wird das Ergebnis des Ausdrucks auf der rechten Seite zugewiesen.
<code>"Text"</code>	beliebige Zeichenkette, die die Anführungszeichen selbst nicht enthalten darf, weil sie als Anfangs- beziehungsweise Endemarkierung einer Zeichenfolge dienen. Wenn die Zeichenfolge die Anführungszeichen enthalten soll, sind diese als <code>\</code> zu schreiben: <code>cout << "\"C++\" ist der Nachfolger von \"C\"!";</code> erzeugt die Bildschirmausgabe <i>"C++" ist der Nachfolger von "C"!</i> .
<code>return 0;</code>	Unser Programm läuft einwandfrei; es gibt daher 0 zurück. Diese Anweisung kann fehlen, dann wird automatisch 0 zurückgegeben.

`<iostream>` ist ein Header. Dieser aus dem Englischen stammende Begriff (*head* = dt. Kopf) drückt aus, dass Zeilen dieser Art am Anfang eines Programmtextes stehen. Der Begriff wird im Folgenden verwendet, weil es zurzeit keine gängige deutsche Entsprechung gibt. Einen Header mit einem Dateinamen gleichzusetzen, ist meistens richtig, nach dem C++-Standard aber nicht zwingend.

`a`, `b` und `summe` sind veränderliche Daten und heißen Variablen. Sie sind Objekte eines vordefinierten Grunddatentyps für ganze Zahlen (`int`), mit denen die üblichen Ganzzahloperationen wie `+`, `-` und `=` durchgeführt werden können. Der Begriff »Variable« wird für ein veränderliches Objekt gebraucht. Für Variablen gilt:

- Objekte müssen deklariert werden. `int summe;` ist eine Deklaration, wobei `int` der *Datentyp* des Objekts `summe` ist, der die Eigenschaften beschreibt. Entsprechendes gilt für `a` und `b`. Die Objektnamen sind frei wählbar im Rahmen der unten angegebenen Konventionen. Unter *Deklaration* wird verstanden, dass der Name dem Compiler bekannt gemacht wird. Wenn dieser Name später im Programm versehentlich falsch geschrieben wird, z. B. `sume = a + b;` im Programm auf Seite 33, kennt der Compiler den falschen Namen `sume` nicht und gibt eine Fehlermeldung aus. Damit dienen Deklarationen der Programmsicherheit.
- Objektnamen bezeichnen Bereiche im Speicher des Computers, deren Inhalte verändert werden können. Die Namen sind symbolische Adressen, unter denen der Wert gefunden wird. Über den Namen kann dann auf den aktuellen Wert zugegriffen werden (siehe Abbildung 1.1).

Der Speicherplatz wird vom Compiler reserviert. Man spricht dann von der *Definition* der Objekte. Definition und Deklaration werden unterschieden, weil es auch Deklarationen ohne gleichzeitige Definition gibt, doch davon später. Zunächst sind die Deklarationen zugleich Definitionen. Abbildung 1.2 zeigt den Ablauf der Erzeugung eines lauffähigen Programms. Ein Programm ist ein Text, von Menschenhand geschrieben (über Programmgeneratoren soll hier nicht gesprochen werden) und dem Rechner unverständlich. Um dieses Programm auszuführen, muss es erst vom Compiler in eine für den Computer verständliche Form übersetzt werden.

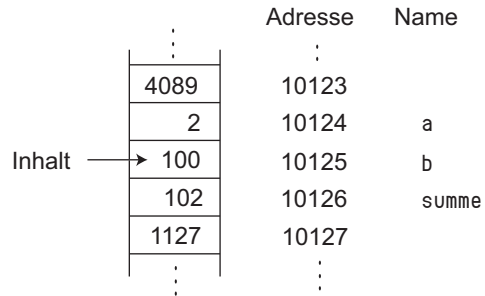


Abbildung 1.1: Speicherbereiche mit Adressen

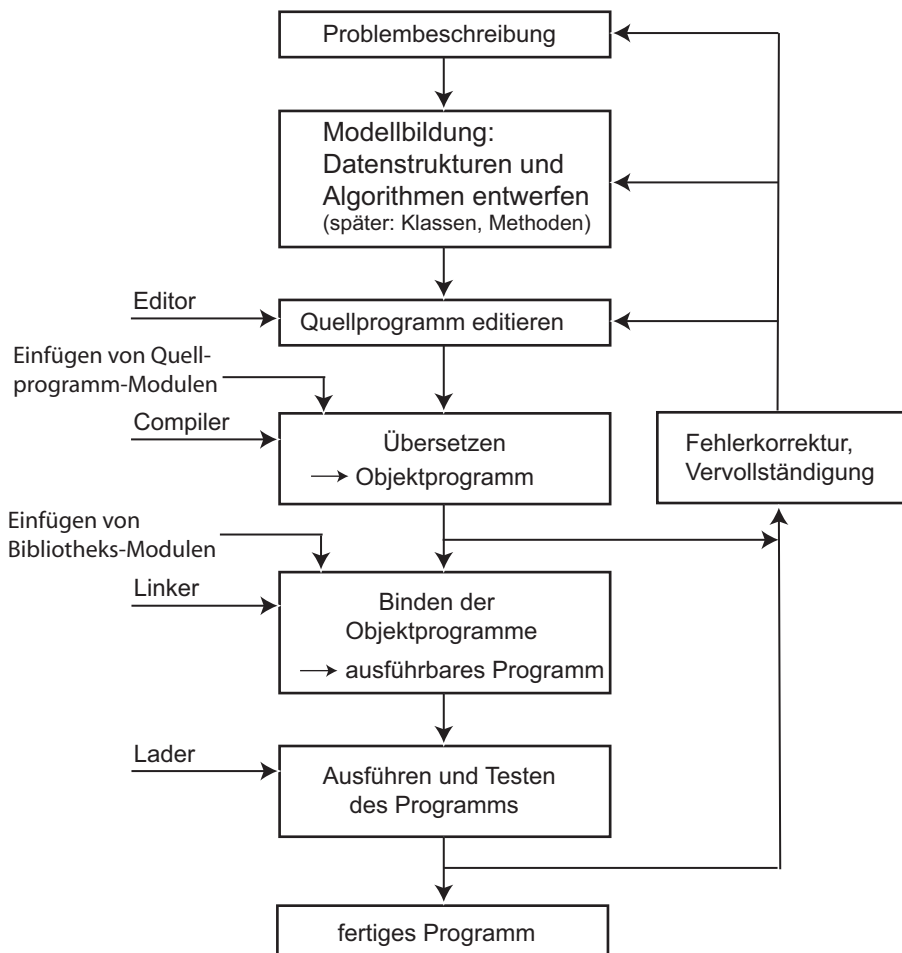


Abbildung 1.2: Erzeugung eines lauffähigen Programms

Der Compiler ist selbst ein Programm, das bereits in maschinenverständlicher Form vorliegt und speziell für diese Übersetzung zuständig ist. Nach Eingabe des Programmtextes mit dem Editor können Sie den Compiler starten und anschließend das Programm binden oder linken (eine Erklärung folgt bald) und ausführen. Ein Programmtext wird auch »Quelltext« (englisch *source code*) genannt.

Der Compiler erzeugt den Objektcode, der noch nicht ausführbar ist. Hinter den einfachen Anweisungen `cin >> ...` und `cout << ...` verbergen sich eine Reihe von Aktivitäten wie die Abfrage der Tastatur und die Ansteuerung des Bildschirms, die nicht speziell programmiert werden müssen, weil sie schon in vorübersetzter Form in Bibliotheksdateien vorliegen. Die Aufrufe dieser Aktivitäten im Programm müssen mit den dafür vorgesehenen Algorithmen in den Bibliotheksdateien zusammengebunden werden, eine Aufgabe, die der *Linker* übernimmt, auch *Binder* genannt. Der Linker bindet Ihren Objektcode mit dem Objektcode der Bibliotheksdateien zusammen und erzeugt daraus ein ausführbares Programm, das nun gestartet werden kann. Der Aufruf des Programms bewirkt, dass der *Lader* eine Funktion des Betriebssystems, das Programm in den Rechner Speicher lädt und startet. Diese Schritte werden stets ausgeführt, auch wenn sie in den Programmumweltumgebungen verborgen ablaufen. Bibliotheksmodule können auch während der Programmausführung geladen werden (nicht im Bild dargestellt). Weitere Details werden in Abschnitt 3.3 erläutert.

Wie bekomme ich ein Programm zum Laufen?

Der erste Schritt ist das Schreiben mit einem Textsystem, Editor genannt. Der Text sollte keine Sonderzeichen zur Formatierung enthalten, weswegen nicht alle Editoren geeignet sind. Integrierte Entwicklungsumgebungen (englisch *Integrated Development Environment*, *IDE*) haben einen speziell auf Programmierzwecke zugeschnittenen Editor, der darüber hinaus auf Tastendruck oder Mausklick die Übersetzung anstößt. Alternativ besteht die Möglichkeit, Compiler und Linker per Kommandozeile in einer Linux-Shell oder einem Windows-Eingabeaufforderungs-Fenster zu starten. Beispiel für das Programm *summe.cpp* und den Open Source C++-Compiler g++[GCC]:

```
g++ -c summe.cpp    compilieren (summe.o wird erzeugt)
                    (bzw. summe.obj bei anderen Compilern)

g++ -o summe.exe summe.o    linken. Beide Schritte lassen sich zusammenfassen:
g++ -o summe.exe summe.cpp
```

Das Programm wird durch Eintippen von *summe.exe* gestartet. Dabei wird vorausgesetzt, dass der g++-Compiler im Pfad ist – sonst wird er nicht gefunden. Wie man den Pfad um ein Verzeichnis erweitert, lesen Sie im Anhang, Abschnitt A.7. Weitere Einzelheiten zur Bedienung von Compiler und Linker finden Sie in Abschnitt A.5 auf Seite 891 oder in den Hilfedateien Ihres C++-Systems. Der folgenden Abschnitt 1.5 zeigt Installation und Bedienung einer kostenlosen² Integrierten Entwicklungsumgebung.

1.4.1 Namenskonventionen

Funktions-, Variablen- und andere Namen unterliegen der folgenden Konvention:

² Bitte Lizenzbestimmungen beachten!

- Ein Name ist eine Folge von Zeichen, bestehend aus Buchstaben, Ziffern und Unterstrich (_).
- Ein Name beginnt stets mit einem Buchstaben oder einem Unterstrich. Am Anfang eines Namens sollten Unterstriche jedoch vermieden werden, ebenso Namen, die zwei Unterstriche (__) direkt nacheinander enthalten. Solche Namen werden systemintern benutzt.
- Selbsterfundene Namen dürfen nicht mit den vordefinierten Schlüsselwörtern übereinstimmen (zum Beispiel `for`, `int`, `main` ...). Eine Tabelle der Schlüsselwörter ist im Anhang auf Seite 887 zu finden.
- Ein Name kann prinzipiell beliebig lang sein. In den Compilern ist die Länge jedoch begrenzt, zum Beispiel auf 31 oder 255 Zeichen.

Die hier aufgelisteten Konventionen zeigen die Regeln für die *Struktur* eines Namens, auch *Syntax* oder *Grammatik* genannt. Ein Name darf niemals ein Leerzeichen enthalten! Wenn eine Worttrennung aus Gründen der Lesbarkeit gewünscht ist, kann man den Unterstrich oder einen Wechsel in der Groß- und Kleinschreibung benutzen. Beispiele für Deklarationen:

<code>int 1_Zeile;</code>	falsch! (Ziffer am Anfang)
<code>int Anzahl der Zeilen;</code>	falsch! (Name enthält Leerzeichen)
<code>int AnzahlDerZeilen;</code>	richtig! andere Möglichkeit:
<code>int Anzahl_der_Zeilen;</code>	richtig!

Zur Abkürzung können Variablen des gleichen Datentyps aufgelistet werden, sofern sie durch Kommas getrennt werden. `int a; int b; int c;` ist gleichwertig mit `int a,b,c;`.

1.5 Integrierte Entwicklungsumgebungen

Im Folgenden stelle ich Ihnen kurz zwei ausgewählte Integrierte Entwicklungsumgebungen, abgekürzt IDE, vor: Code::Blocks und Eclipse. Beide gibt es für Windows und Linux.



Tipp

Anfängern empfehle ich die IDE Code::Blocks, sei es unter Linux oder Windows, weil sie am einfachsten zu installieren und zu bedienen ist.

1.5.1 Code::Blocks

Code::Blocks [CB] ist eine Open Source-IDE für C++ für Windows und Linux, die einfach installier- und bedienbar und für unsere Zwecke sehr gut geeignet ist. Weil in diesem Buch nur sehr kurz auf die Bedienung eingegangen werden kann, empfehle ich Ihnen die Bedienungsanleitung von Code::Blocks, die Sie auf der DVD zu diesem Buch oder der Internetseite [CB] finden.

**Code::Blocks auf DVD**

Code::Blocks ist auf der DVD vorhanden. Die Installationsanleitung für die Software (die auch Code::Blocks enthält) für Windows finden Sie in Abschnitt [A.7](#) (Seite 937), die für Linux in Abschnitt [A.8.3](#) (Seite 944).

Programm eingeben, übersetzen und starten

Um jetzt ein Programm einzugeben, starten Sie Code::Blocks (Windows: Start → Alle Programme → CodeBlocks). Nach dem Start legen Sie ein neues Projekt an, indem Sie auf »Create a new Project« im Startfenster klicken. Alternativ ist der Weg über den Menübalken möglich (File → New → Project). Es wird Ihnen eine Auswahl verschiedenster Anwendungstypen angeboten. Fürs Erste wählen Sie bitte die einfachste Art der Anwendung, die »Console Application«. Im dann erscheinenden Fenster müssen noch einige Angaben eingetragen werden. Vorschlag:

Project title: *Projekt1*

Folder to create project in: *cpp_projekt*

Die anderen Einstellungen werden belassen. Nun »Next« und »Finish« anklicken. Im linken Teil klicken Sie bitte auf Sources und dort auf *main.cpp*. Bitte ändern Sie das angezeigte Programm so, dass Sie damit die Summe zweier Zahlen berechnen können (Programm von Seite 33). Um einen Fehler zu provozieren und seine Reparatur zu zeigen, wird noch eine nichtexistierende Variable *c* addiert. Ein Klick auf das Diskettensymbol oben sichert die Datei.

Ein Klick auf das Build-Icon oder Drücken der Tastenkombination Strg-F9 startet den Übersetzungsprozess. Der mit Absicht erzeugte Fehler wird nun im Programm mit einem roten Balken markiert; Erklärungen dazu finden sich im Fenster unter dem Programmcode, siehe Abbildung 1.3. Es empfiehlt sich, die Hinweise auf Fehler genau zu lesen!

Im Bild und in diesem Buch wird übrigens für die Positionierung der geschweiften Klammern der Kernighan & Ritchie³-Stil gewählt (Settings → Editor, links unten »Source formatter« anklicken und dann rechts K&R wählen).

Jetzt korrigieren Sie bitte das Programm, indem die fehlerhafte Addition der Variablen *c* entfernt wird. Ein weiterer Klick auf das Build-Icon wird nun von Erfolg gekrönt: 0 errors, 0 warnings! Ein Klick auf das Run-Icon (oder die Tastenkombination Strg-F10) führt zur Ausführung des Programms. Im erscheinenden Fenster geben Sie einfach zwei Zahlen ein und drücken `[ENTER]`. Das Ergebnis wird dann angezeigt. Mit einem weiteren Tastendruck wird das Fenster geschlossen. Man kann eine weitere Pause erzwingen, falls das Fenster sich vorher zu schnell schließt. Dazu werden am Programmende die Zeilen

```
cin.ignore(1000, '\n'); // genaue Erklärung folgt in Kap. 10
cin.get();
```

hinzugefügt, wie im Beispiel auf der DVD gezeigt (Datei *cppbuch/k1/summe.cpp*).

³ Ritchie hat unter der Mitwirkung von Kernighan die Programmiersprache C entwickelt.

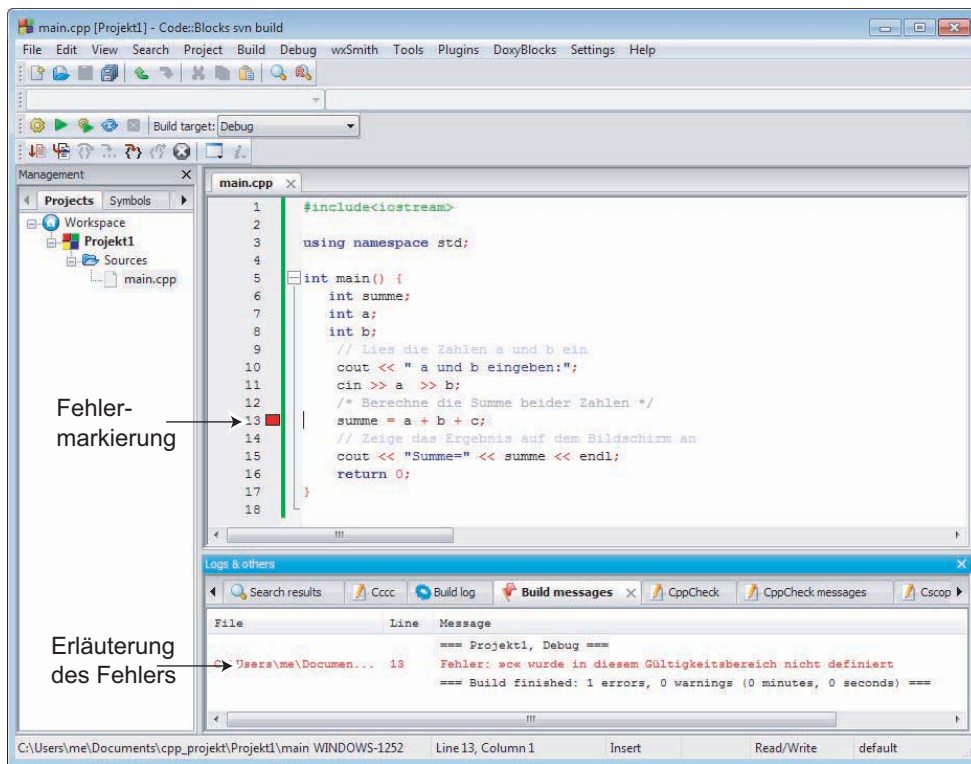


Abbildung 1.3: Die Entwicklungsumgebung Code::Blocks zeigt einen Fehler an.



Tipp

Man kann das Programm direkt aus einem bestehenden Fenster, das dann weiter bestehen bleibt, starten, hier gezeigt für das Betriebssystem Windows. Dazu rufen Sie in Windows Start → Alle Programme → Zubehör → Eingabeaufforderung auf und gehen durch Anwendung des Befehls `cd` in das Verzeichnis, in dem das Programm abgelegt worden ist, zum Beispiel `cpp_projekt/Projekt1/bin/Debug`. In diesem Verzeichnis befindet sich die vom Compiler erzeugte ausführbare Datei `Projekt1.exe`, wie der Befehl `dir` zeigt. Wenn Sie nun `Projekt1.exe` eintippen, wird das Programm ausgeführt.

1.5.2 Eclipse

Eclipse (<http://www.eclipse.org>) ist eine mächtige Entwicklungsumgebung, die es für viele Betriebssysteme gibt, darunter Linux, Mac OS X und Windows. Sie setzt eine Java-Installation voraus. Für erfahrene Softwareentwickler, die von Java auf C++ umsteigen, ist Eclipse sehr gut geeignet. Eclipse gibt es fertig für C/C++ konfiguriert zum Herunterladen. Ein Hinweis: Bitte loggen Sie sich für alle Installationsvorgänge als »Administrator« (Windows) bzw. »root« (Linux) ein, wenn die Installation für alle Benutzer des Rechners gelten soll. Eclipse kann aber auch lokal installiert werden. Der C++-Compiler ist nicht

Bestandteil von Eclipse. Er muss daher vorher installiert worden sein und sich im Pfad befinden. Dann findet Eclipse ihn automatisch. In Abbildung 1.4 wird der gefundene Compiler rechts angezeigt.

Anlegen eines neuen C++-Projekts

Die Folge File → New → C++ Project ergibt das in Abbildung 1.4 gezeigte Fenster.

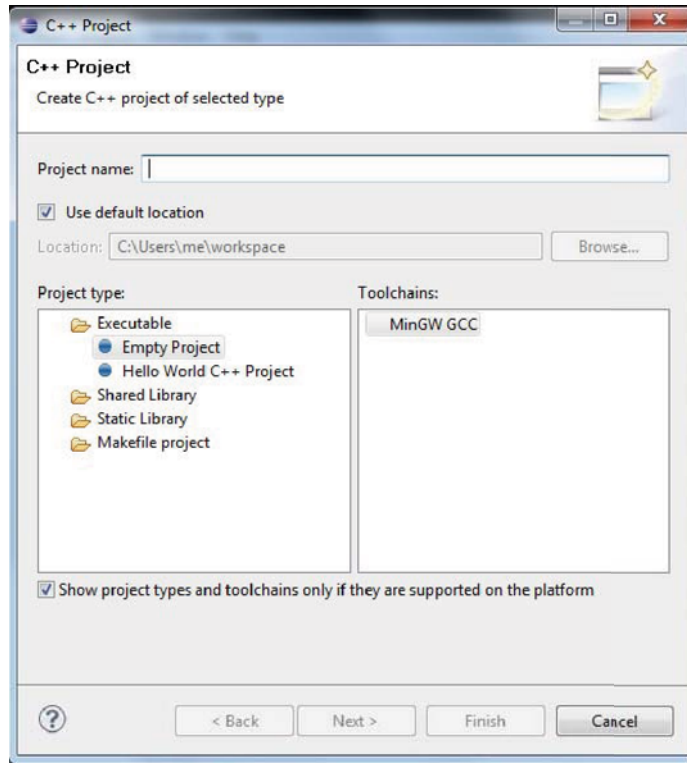


Abbildung 1.4: Neues Projekt anlegen

Dort tragen Sie oben den Projektnamen ein, zum Beispiel »Mein erstes C++-Projekt« und klicken auf Finish. Nun wird als Beispiel das einfache Programm zur Berechnung einer Summe erzeugt. Dazu klicken Sie mit der *rechten* Maustaste auf den Projektnamen links im Fenster und wählen New → Source File. Als Name können Sie zum Beispiel *sum-me.cpp* eingeben. Mit Finish gelangen Sie in das Editorfenster, in das Sie das Programm nun eingeben können. Wenn Sie die rot markierten Worte in Kommentaren stören: Entweder installieren Sie das deutsche Wörterbuch (im Internet nach BabelLanguagePack-eclipse suchen), oder Sie schalten die Rechtschreibprüfung ab (unter Window → Preferences → General – Editors – Text Editors – Spelling).

Der Compilervorgang wird durch Anklicken des Hammer-Symbols oben gestartet. Zum Vergleich wurde derselbe Fehler wie oben eingebaut, siehe Abbildung 1.5:

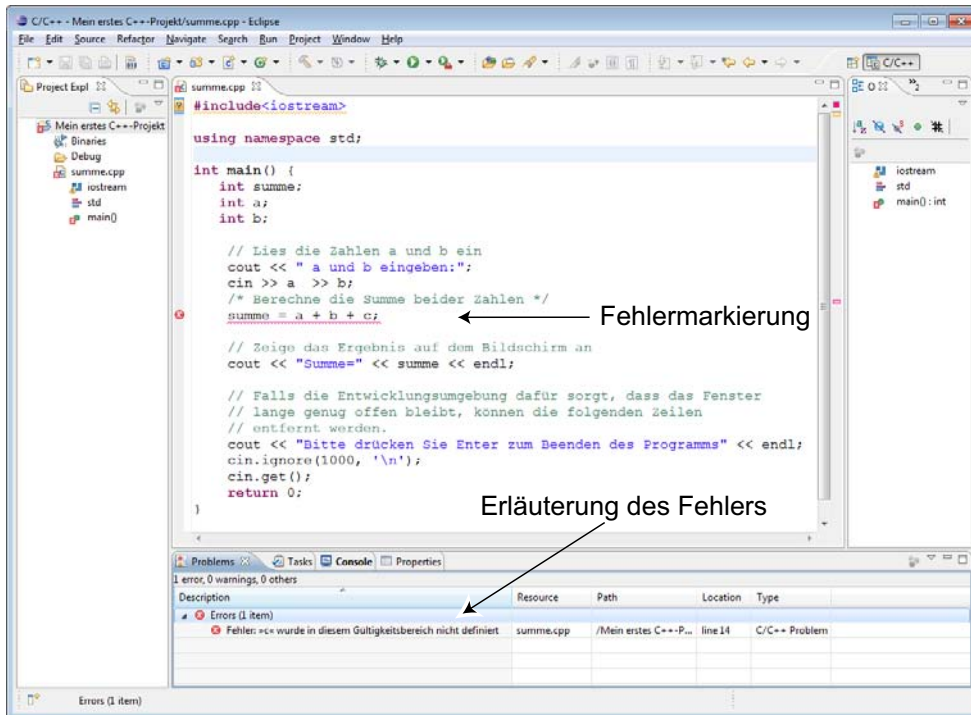


Abbildung 1.5: Die Eclipse-IDE zeigt einen Fehler an.

Nach Korrektur wird das Programm gesichert und noch einmal compiliert. Durch Klicken auf das Run-Symbol oben (helles Dreieck im grünen Kreis) bringen Sie das Programm zum Laufen. Im Konsolen-Fenster von Eclipse (unten im Eclipse-Fenster) können Sie nun zwei Zahlen eingeben. Die berechnete Summe wird dort angezeigt.

1.6 Einfache Datentypen und Operatoren

Sie haben schon flüchtig die Datentypen `int` für ganze Zahlen und `float` für Gleitkommazahlen kennengelernt. Es gibt darüber hinaus noch eine Menge anderer Datentypen. Hier wird näher auf die *Grunddatentypen* eingegangen. Sie sind definiert durch ihren Wertebereich sowie die mit diesen Werten möglichen Operationen. Nicht-veränderliche Daten sind für alle Grunddatentypen möglich; sie werden in Abschnitt 1.6.4 erläutert.

1.6.1 Ausdruck

Ein Ausdruck besteht aus einem oder mehreren Operanden, die miteinander durch Operatoren verknüpft sind. Die Auswertung eines Ausdrucks resultiert in einem Wert, der an die Stelle des Ausdrucks tritt. Der einfachste Ausdruck besteht aus einer einzigen Konstante, Variable oder einem Literal. Die Operatoren müssen zu den Operanden passen, die zu bestimmte Datentypen gehören. Beispiele: 17, 1+ 17, a = 1 + 17, »Textliteral«.

a = 1 + 17 ist ein zusammengesetzter Ausdruck. Die Operanden 1 und 17 sind Zahl-Literale, die hier ganze Zahlen repräsentieren und die durch den +-Operator verknüpft werden. Der resultierende Wert 18 wird dem Objekt a zugewiesen. Der Wert des gesamten Ausdrucks ist der resultierende Wert von a.

1.6.2 Ganze Zahlen

Es gibt verschiedene rechnerinterne Darstellungen von ganzen Zahlen, die sich durch die bereitgestellte Anzahl von Bits pro Zahl unterscheiden. Die verschiedenen Darstellungen werden durch die Datentypen short, int und long repräsentiert, wobei gilt $\text{Bits}(\text{short}) \leq \text{Bits}(\text{int}) \leq \text{Bits}(\text{long})$. Die tatsächlich verwendete Anzahl von Bits variiert je nach Rechnersystem. Typische Werte sind:

short (oder short int)	16 Bits
int	32 Bits (oder 16 Bits)
long (oder long int)	64 Bits (oder 32 Bits)
long long (oder long long int)	mindestens soviel wie long

Die eingeklammerten Bitzahlen sind entsprechend dem C++-Standard mindestens erforderlich. Ein Bit wird für das Vorzeichen reserviert. In den folgenden ausgewählten Bitkombinationen für 16-Bit-int-Zahlen repräsentiert das links stehende Bit das Vorzeichen:

binär	dezimal
0111 1111 1111 1111	32767
0000 0000 0000 0000	0
1111 1111 1111 1111	-1
1111 1111 1111 1110	-2
1000 0000 0000 0000	-32768

Negative Zahlen werden üblicherweise im Zweierkomplement dargestellt. Das Zweierkomplement wird gebildet, indem alle Bits invertiert werden und dann auf das Ergebnis 1 addiert wird. Durch das Schlüsselwort unsigned werden Zahlen ohne Vorzeichen definiert, zum Beispiel unsigned int, unsigned long (Langform: unsigned long int). Durch das damit gewonnene zusätzliche Bit teilt sich der Zahlenbereich anders auf:

signed	16 Bits	$-2^{15} \dots 2^{15}-1$	=	-32 768 ...	32 767
unsigned	16 Bits	$0 \dots 2^{16}-1$	=	0 ...	65 535
signed	32 Bits	$-2^{31} \dots 2^{31}-1$	=	-2 147 483 648 ...	2 147 483 647
unsigned	32 Bits	$0 \dots 2^{32}-1$	=	0 ...	4 294 967 295
signed	64 Bits	$-2^{63} \dots 2^{63}-1$	=	-9 223 372 036 854 775 808	...
				...	9 223 372 036 854 775 807
unsigned	64 Bits	$0 \dots 2^{64}-1$	=	0 ...	18 446 744 073 709 551 615

Die in Ihrem C++-System zutreffenden Zahlenbereiche finden Sie im Header `<climits>` unter den Namen `INT_MAX` usw. Das folgende Programm gibt die Grenzwerte und den benötigten Speicherplatz für die Ganzzahl-Typen aus.

Listing 1.2: Grenzwerte und Speicherplatzbedarf

```
// cppbuch/k1/limits.cpp
#include<iostream>
#include<climits>          // hier sind die Bereichsinformationen
using namespace std;
int main() {
    cout << "Grenzwerte für Ganzzahl-Typen:" << endl; // = neue Zeile (newline)
    cout << "INT_MIN =" << INT_MIN << endl;
    cout << "INT_MAX =" << INT_MAX << endl;
    cout << "LONG_MIN =" << LONG_MIN << endl;
    cout << "LONG_MAX =" << LONG_MAX << endl;
    cout << "LLONG_MIN =" << LLONG_MIN << endl;
    cout << "LLONG_MAX =" << LLONG_MAX << endl;
    cout << "unsigned-Grenzwerte:" << endl;
    cout << "UINT_MAX =" << UINT_MAX << endl;
    cout << "ULONG_MAX =" << ULONG_MAX << endl;
    cout << "ULLONG_MAX =" << ULLONG_MAX << endl;
    cout << "Anzahl der Bytes für:" << endl;
    cout << "int      " << sizeof(int) << endl;
    cout << "long     " << sizeof(long) << endl;
    cout << "long long " << sizeof(long long) << endl;
}
```

Statt `INT_MAX` können Sie `numeric_limits<int>::max()` schreiben usw. Diese Alternative wird im Listing auf Seite 47 am Beispiel der Rechnerdarstellung für reelle Zahlen gezeigt. Beim Rechnen mit ganzen Zahlen ist der *begrenzte Wertebereich* zu beachten! Aufgrund der Tatsache, dass nur eine begrenzte Anzahl von Bits für die rechnerinterne Repräsentation einer Zahl zur Verfügung steht, ergibt sich, dass der von `int` abgedeckte Zahlenbereich nur eine *Untermenge* der ganzen Zahlen darstellt, wie das Programmbeispiel »Arithmetischer Überlauf« zeigt.

Listing 1.3: Arithmetischer Überlauf

```
// cppbuch/k1/overflow.cpp
#include <iostream>
using namespace std;

int main() {
    int ai = 50000;
    int bi = 1000000;
    int ci = ai * bi;
    cout << "int-Zahlen haben auf Ihrem System "
         << 8*sizeof(int) << " Bits" << endl;
    cout << "Rechnung mit int: ";
    cout << ai << " * " << bi << " = " << ci << endl;
    // Ausgabe -1539607552 statt 50000000000 bei 32 Bit-int
    long al = 50000; long bl = 1000000; long cl = al*bl;
    cout << "long-Zahlen haben auf Ihrem System "
```



```

    << 8*sizeof(long) << " Bits" << endl;
    cout << "Rechnung mit long: "
        << a1 << "*" << b1 << "=" << c1 << endl;

    // Falls Ihr Compiler long long unterstützt:
    long long a11 = 50000; long long b11 = 1000000;
    long long c11 = a11 * b11;
    cout << "long long-Zahlen haben auf Ihrem System "
        << 8*sizeof(long long) << " Bits" << endl;
    cout << "Rechnung mit long long: ";
    cout << a11 << "*" << b11 << "=" << c11 << endl;
}

```

Daraus folgt, dass die Regeln der Mathematik in der Nähe der Grenzen des Zahlenintervalls nur noch eingeschränkt gelten. Das Ergebnis einer Folge arithmetischer Operationen ist nur dann korrekt, wenn kein Zwischenergebnis den durch den Datentyp vorgegebenen maximalen Zahlenbereich überschreitet. 50000000000 liegt außerhalb des durch 32 Bits darstellbaren Zahlenbereichs. Wird das Resultat einer Operation betragsmäßig zu groß, liegt ein *Überlauf* (englisch *overflow*) vor, der durch den Computer *nicht* gemeldet wird. Der Ersatz von `int` durch `long` führt im obigen Beispiel nur dann zu einem Programm, das das korrekte Ergebnis ausgibt, falls `long` mehr Bits als `int` hat. Dies ist oft nicht der Fall. Das Problem ist nicht grundsätzlich lösbar; es wird bei Ganzzahl-Datentypen mit mehr Bits pro Zahl nur in Richtung größerer Zahlen verschoben.

Beim Schreiben eines Programms muss man sich also Gedanken über die möglichen vorkommenden Zahlenwerte machen. »Sicherheitshalber« immer den größtmöglichen Datentyp zu wählen, ist nicht sinnvoll, weil Variablen dieses Typs mehr Speicherplatz benötigen und weil Rechenoperationen mit ihnen länger dauern. Ganze Zahlen können auf dreierlei Arten dargestellt werden:

1. Wenn eine Zahl mit einer 0 beginnt, wird sie als *Oktalzahl* interpretiert, zum Beispiel $0377 = 377_8 = 3 \cdot 8^2 + 7 \cdot 8^1 + 7 \cdot 8^0 = 255_{10}$ (dezimal).
2. Wenn eine Zahl mit 0x oder 0X beginnt, wird sie als *Hexadezimalzahl* interpretiert, zum Beispiel `0xAFFE` = 45054 dezimal.
3. Dezimal wie üblich. Ein Suffix (l, ll, L, LL) kennzeichnet `long`- oder `long long`-Zahlen, zum Beispiel `2147483647L` oder `9223372036854775806LL`. Ein Suffix `u` oder `U` kennzeichnet `unsigned`-Zahlen.

Operatoren für ganze Zahlen

Die Tabelle 1.2 zeigt die für ganze Zahlen möglichen Operatoren, also für die Datentypen `short`, `int` und `long`. Die zusammengesetzten Operatoren wie `+=` heißen Kurzform-Operatoren. Auf Daten eines bestimmten Typs kann man nur bestimmte Operationen durchführen. Eine Zeichenkette kann man zum Beispiel nicht durch eine andere dividieren. Man kann jedoch ganze Zahlen `a` und `b` addieren. Daraus folgt: Ein Datum und die zugehörigen Operationen gehören zusammen! Einige der Operatoren der Tabelle 1.2 werden durch Beispiele erläutert. Manche Operatoren setzen jedoch hier noch nicht besprochene Dinge voraus, weshalb gelegentlich auf spätere Abschnitte verwiesen wird. Beispiele für Operatoren:

Tabelle 1.2: Operatoren für Ganzzahlen

Operator	Beispiel	Bedeutung
Arithmetische Operatoren:		
+	+ i	unäres Plus (kann weggelassen werden)
-	- i	unäres Minus
++	++ i	vorherige Inkrementierung um eins
	i ++	nachfolgende Inkrementierung um eins
--	-- i	vorherige Dekrementierung um eins
	i --	nachfolgende Dekrementierung um eins
+	i + 2	binäres Plus
-	i - 5	binäres Minus
*	5 * i	Multiplikation
/	i / 6	Division
%	i % 4	Modulo (Rest mit Vorzeichen von i)
=	i = 3 + j	Zuweisung
*=	i *= 3	i = i * 3
/=	i /= 3	i = i / 3
%=	i %= 3	i = i % 3
+=	i += 3	i = i + 3
-=	i -= 3	i = i - 3
relationale Operatoren:		
<	i < j	kleiner als
>	i > j	größer als
<=	i <= j	kleiner gleich
>=	i >= j	größer gleich
==	i == j	gleich
!=	i != j	ungleich
Bit-Operatoren:		
<<	i << 2	Linksschieben (Multiplikation mit 2er-Potenzen)
>>	i >> 1	Rechtsschieben (Division durch 2er-Potenzen)
&	i & 7	bitweises UND
^	i ^ 7	bitweises XOR (Exklusives Oder)
	i 7	bitweises ODER
~	~ i	bitweise Negation
<<=	i <<= 3	i = i << 3
>>=	i >>= 3	i = i >> 3
&=	i &= 3	i = i & 3
=	i = 3	i = i 3

```
int a;
int i = 5;
a = ++i;    // vorangestelltes ++
```

i wird *erst* um eins *inkrementiert*, *dann benutzt*. Unter Inkrementierung wird die Addition von 1 verstanden, unter Dekrementierung die Subtraktion von 1.

`++` ist der Inkrementierungsoperator, der je nach Stellung eine Variable vor oder nach Benutzung des Werts hochzählt. Nach dieser Anweisung haben sowohl `a` als auch `i` den Wert 6.

```
int j = 2;
int b = j++;    // nachgestelltes ++
```

`j` wird *erst benutzt, dann inkrementiert*. Nach dieser Anweisung hat `b` den Wert 2 und `j` den Wert 3.

```
j = j + 4;
j += 5;
```

Hier wird 4 zu `j` addiert. Ergebnis: `j = 7`. Anschließend wird 5 mit dem Kurzformoperator hinzugefügt. `j` hat danach den Wert 12.

Ganzzahlige Division

Wenn bei der Division nur ganze Zahlen beteiligt sind, ist das Ergebnis auch ganzzahlig! Der Rest wird verworfen und bei Bedarf mit dem Modulo-Operator `%` ermittelt:

```
int m = 9, n = 5;
int ergebnis = m / n; // 1
int rest = m % n;     // 4
```

Bit-Operatoren

Weil ganze Zahlen auch als Bitvektoren aufgefasst werden können, sind zusätzlich Bit-Operationen möglich. Es folgen Beispiele für zwei `int`-Zahlen `c` und `k`. Die Zahl `k` soll den Wert 5 repräsentieren. Die binäre Darstellung der Zahl 5 ist für 16-Bit-`int`-Zahlen `0000 0000 0101`. Die Anweisung `c = k << 2;` bewirkt eine Bitverschiebung um 2 Stellen nach links und Zuweisung des Ergebnisses an `c`; das entspricht der Multiplikation mit 2^2 , also 4.

```
0000 0000 0000 0101  k ist gleich 5
0000 0000 0001 0100  c= k << 2, d.h. 2 Stellen verschoben. c hat danach den Wert 20.
```

Bei der Verschiebung nach links werden von rechts Nullen nachgezogen. Wenn nach rechts verschoben wird, werden links Nullen eingefügt, falls der Operand vom Typ `unsigned` ist. Bei vorzeichenbehafteten Typen wird links entweder das Vorzeichenbit kopiert oder es werden Nullbits eingefügt – je nach C++-System. Die Anweisung `c = c & k;` bewirkt die bitweise UND-Verknüpfung:

```
0000 0000 0000 0101  k ist gleich 5
0000 0000 0001 0100  c ist gleich 20
0000 0000 0000 0100  Das Ergebnis von c & k ist 4.
```

Die Anweisung `c = ~k;` bewirkt die bitweise Negation:

```
0000 0000 0000 0101  k ist gleich 5
1111 1111 1111 1010  c = ~k (also -6)
```

Würde man auf `c` noch 1 addieren, erhielte man das Zweierkomplement, also die Darstellung der negativen Zahl -5.

size_t

Der Datentyp `size_t` ist ein vorzeichenloser Ganzzahl-Typ für Größenangaben, die nicht negativ werden können, wie zum Beispiel die Anzahl der Einträge in einer Tabelle. Er ist groß genug, die Größe eines beliebigen Objekts in Bytes abzubilden. Die Definition steht im Header `<stddef>`, der von vielen Standard-Headern bereits eingeschlossen wird.

1.6.3 Reelle Zahlen

Reelle Zahlen, auch Gleitkommazahlen genannt, sind wegen der beschränkten Bit-Anzahl in der Regel nicht beliebig genau darstellbar. Sie werden in C++ wie folgt geschrieben:

Vorzeichen (optional), Vorkommastellen, Dezimalpunkt, Nachkommastellen, `e` oder `E` und Ganzzahl-Exponent (optional), Suffix `f`, `F` oder `l`, `L` (optional, Zahlen ohne Suffix sind `double`).

Es können wegfallen:

- entweder Vorkommastellen oder Nachkommastellen (aber nicht beide), oder
- entweder Dezimalpunkt oder `e` (oder `E`) mit Exponent (aber nicht beide).

Einige Beispiele für Gleitkommazahlen sind:

-123.789e6f 1.8E6 88.009 1e-03 1.8L

Der Exponent meint Zehnerpotenzen. $1.8e6$ ist dasselbe wie 1.8×10^6 oder 1800000. Reelle Zahlen werden durch die drei Datentypen der Tabelle 1.3 (ungenau) dargestellt.

Tabelle 1.3: Datentypen und Bereiche für reelle Zahlen

Typ	Bits	Zahlenbereich		Genauigkeit in Dezimalstellen
<code>float</code>	32	$\pm 1.1 \cdot 10^{-38}$	$\dots \pm 3.4 \cdot 10^{38}$	ca. 7
<code>double</code>	64	$\pm 2.2 \cdot 10^{-308}$	$\dots \pm 1.8 \cdot 10^{308}$	ca. 16
<code>long double</code>	96	$\pm 3.4 \cdot 10^{-4932}$	$\dots \pm 1.2 \cdot 10^{4932}$	ca. 24

Anstelle des Kommas tritt ein Dezimalpunkt, wie im angelsächsischen Sprachraum üblich. Die Festlegungen Ihres C++-Systems sind im Header `<limits>` oder in der C-Datei `float.h` zu finden. Anders als im Programm auf Seite 43 können die `numeric_limits<float>`- bzw. `numeric_limits<double>`-Funktionen Auskunft geben, die es auch für `int` usw. gibt.

Listing 1.4: Grenzwerte von float-Zahlen

```
// cppbuch/k1/floatlimits.cpp
#include<iostream>
#include<limits>           // hier sind die Bereichsinformationen
using namespace std;

int main() {
    cout << "Grenzwerte für Float-Zahl-Typen:" << endl;
    cout << "Float-Min: "
         << numeric_limits<float>::min() << endl;
    cout << "Float-Max: "
         << numeric_limits<float>::max() << endl;

    cout << "Double-Min: "
         << numeric_limits<double>::min() << endl;
    cout << "Double-Max: "
```

```

    << numeric_limits<double>::max() << endl;

    cout << "Long-Double-Min: "
         << numeric_limits<long double>::min() << endl;
    cout << "Long-Double-Max: "
         << numeric_limits<long double>::max() << endl;
}

```

Intern werden Mantisse und Exponent jeweils durch Binärzahlen einer bestimmten Bitbreite verkörpert. Die hier angegebenen für Mantisse und Exponent aufsummierten Bitbreiten und damit Zahlenbereiche und Rechengenauigkeiten sind *implementationsabhängig* und dienen *nur als Beispiel*. Die Tabelle 1.4 zeigt ein Beispiel der Repräsentation reeller Zahlen im Rechner.

Tabelle 1.4: Anzahl der Bits (Beispiel)

	float	double	long double
Vorzeichen	1	1	1
Mantisse	23	52	80
Exp-Vorzeichen	1	1	1
Exponent	7	10	14
Gesamtanzahl Bits	32	64	96

In C++ ist festgelegt, dass die Genauigkeit von `double`-Zahlen nicht schlechter sein darf als die von `float`-Zahlen, und die von `long double`-Zahlen darf nicht schlechter sein als die von `double`-Zahlen. Die Genauigkeit hängt von der Anzahl der Bits ab, die für die Mantisse verwendet werden. Der Zahlenbereich wird wesentlich durch die Anzahl der Bits für den Exponenten bestimmt, der Einfluss der Mantisse ist minimal. Es sei angenommen, dass für den Typ `double` die oben angegebenen Bitzahlen gelten. Da 2^{52} ca. $4.5 \cdot 10^{15}$ ist, ergibt sich eine etwa 15-stellige Genauigkeit. Der Zahlenbereich leitet sich aus der Bitanzahl für den Exponenten ab: 2^{10} ist 1024. Der Exponent kann damit im Bereich von 0 bis $2^{10} - 1$ liegen. Übertragen auf das Dezimalsystem ergibt sich mithilfe der Schulmathematik ein maximaler Exponent von $1024 \log 2 / \log 10$, also etwa 308. Der darstellbare Bereich geht also bis ca. 10^{308} . Insgesamt ergibt sich, dass eine beliebige Genauigkeit nicht für alle Zahlen möglich ist. Falls 32 Bits für die Darstellung einer reellen Zahl verwendet werden, existieren nur $2^{32} = 4\,294\,967\,296$ verschiedene Möglichkeiten, eine Zahl zu bilden. Mit dem mathematischen Begriff eines reellen Zahlenkontinuums hat das nur näherungsweise zu tun, und alle Illusionen von der computertypischen Genauigkeit und Korrektheit sind dahin, wie das folgende Programm demonstriert:

Listing 1.5: Genauigkeit von float-Zahlen

```

// cppbuch/k1/genau.cpp
#include<iostream>
using namespace std;
int main() {
    float a = 1.234567E-7;
    float b = 1.000000;
}

```

```

float c = -b;
float s1 = a + b;
s1 += c;           // entspricht s1 = s1 + c;
float s2 = a;
s2 += b + c;
cout << "Ungenauigkeit bei float-Arithmetik:" << endl;
cout << "(a+b)+c= " << s1 << '\n';    // 1.19209e-7
cout << "a+(b+c)= " << s2 << '\n';    // 1.23457e-7
}

```

Folgen der nicht exakten Darstellung können sein:

- Bei der Subtraktion zweier fast gleich großer Werte heben sich die signifikanten Ziffern auf. Die Differenz wird damit ungenau. Dieser Effekt ist unter dem Namen *numerische Auslöschung* bekannt.
- Die Division durch betragsmäßig zu kleine Werte ergibt einen *Überlauf* (englisch *overflow*).
- Eine *Unterschreitung* (englisch *underflow*) tritt auf, wenn der Betrag des Ergebnisses zu klein ist, um mit dem gegebenen Datentyp darstellbar zu sein. Das Resultat wird dann gleich 0 gesetzt.
- Ergebnisse können von der Reihenfolge der Berechnungen abhängen. Im Programmbeispiel wird $a+b+c$ auf zwei verschiedene Arten berechnet: Um $s1$ zu berechnen, werden erst a und b addiert und danach c , während zur Berechnung von $s2$ zu a die Summe von b und c addiert wird. Rein mathematisch betrachtet müsste das gleiche Ergebnis herauskommen, die Computerarithmetik liefert jedoch abweichende Ergebnisse. Daher gehört zu kritischen Rechnungen immer eine Genauigkeitsbetrachtung.

Die Ausgabe von `\n` im Beispielprogramm bedeutet, dass nach den Zahlen $s1$ und $s2$ jeweils eine neue Zeile auf dem Bildschirm begonnen wird (siehe Abschnitt 1.6.5).

Es gibt verschiedene Methoden, den Fehler bei ungenauen Rechnungen zu minimieren. Zum Beispiel könnte man bei der Addition einer großen Menge verschiedener Zahlen so vorgehen, dass zunächst die Zahlen sortiert werden und erst danach die Addition vorgenommen wird, beginnend mit der kleinsten Zahl. Für die »reellen« Zahlentypen `float`, `double` und `long double` stehen mit Ausnahme des Modulo-Operators `%` alle arithmetischen und relationalen Operatoren der Tabelle 1.2 zur Verfügung. Tabelle 1.5 auf der nächsten Seite zeigt einige Beispiele zur Umsetzung mathematischer Ausdrücke in die Programmiersprache C++. Einige mathematische Funktionen wie `sqrt()`, `exp()` u.a. sind vordefiniert. Die Deklaration der meisten dieser Funktionen befindet sich im Header `<cmath>`.

Aus historischen Gründen befindet sich die Funktion `abs()` für `int`-Zahlen jedoch im Header `<cstdlib>`. Zum Nachschlagen der verschiedenen Möglichkeiten bieten sich die Tabellen 35.3 und 35.5 an (ab Seite 876). Um dem Compiler die Deklarationen bekannt zu machen, genügt es, im Programm am Anfang der Programmdatei, die Zeile `#include<cmath>` einzufügen. Die Ausgabe von `endl` bewirkt genau wie `\n` den Sprung in eine neue Zeile:

Tabelle 1.5: Umsetzen mathematischer Ausdrücke

Mathematische Schreibweise	C++- Notation
$a - \sqrt{1 + x^2}$	<code>a - sqrt(1+x*x)</code>
$\frac{a-b}{1+\frac{x}{y}}$	<code>(a-b)/(1+x/y)</code>
$ x $	<code>abs(x)</code> <code>int</code> <code>labs(x)</code> <code>long int</code> <code>fabs(x)</code> <code>float, double, long double</code>
$\frac{x}{y}$ $\frac{y}{z}$	<code>x/y/z</code> klarer: <code>(x/y)/z</code>
$e^{-\delta t} \sin(\omega t + \varphi)$	<code>exp(-delta*t)*sin(omega*t+phi)</code>

Listing 1.6: Mathematische Funktionen

```
// cppbuch/k1/mathexpr.cpp : Berechnung mathematischer Ausdrücke
#include<iostream>
#include<cmath>
using namespace std;

int main() {
    float x;
    cout << "x eingeben:";
    cin >> x;
    cout << "x      = " << x      << endl;
    cout << "fabs(x) = " << fabs(x) << endl;
    cout << "sqrt(x) = " << sqrt(x) << endl;
    cout << "sin(x) = " << sin(x) << endl; // Argument von sin() im Bogenmaß!
    cout << "exp(x) = " << exp(x) << endl;
    cout << "log(x) = " << log(x) << endl; // log() ist der natürliche Logarithmus!
}
```



Übung

1.1 Probieren Sie das vorstehende Programm aus! Wie verhält sich Ihr Rechner bei Eingabe von 0 oder einer negativen Zahl? (Dies ist eine von etwa sechs Übungsaufgaben, die mathematisch mehr als die Kenntnis der vier Grundrechenarten voraussetzen – ein Anteil von unter 10% bei 86 Aufgaben insgesamt.)

1.6.4 Konstante

In einem Programm kommen häufig Zahlen oder andere Datenstrukturen vor, die im Programmablauf nicht verändert werden dürfen. Sie heißen *Konstanten*. Zum Beispiel könnte man »umfang = 3.1415926 * durchmesser;« schreiben. Besser ist

```
const float PI = 3.1415926; // noch besser: vordefinierte Konstante M_PI nehmen
umfang = PI * durchmesser;
```

weil bei anschließendem häufigerem Gebrauch der Zahl `PI` Schreibfehler leicht ausgeschlossen werden können, und Änderungen oder Korrekturen einer Konstante nur an einer Stelle vorgenommen werden müssen. `PI` ist nur einmal als Konstante zu vereinbaren (= zu deklarieren). `float` bedeutet, dass die Konstante eine Gleitkommazahl ist. Eine ganzzahlige Konstante würde zum Beispiel mit `const int GROESSE = 1000;` vereinbart.

- Eine Konstante besteht aus einem Namen und dem zugeordneten Wert, der *nicht veränderbar* ist. Der Name wird üblicherweise großgeschrieben.
- Konstanten müssen wie Variablen deklariert werden. Das Schlüsselwort `const` leitet die Deklaration ein, Arithmetik ist erlaubt, z.B. `const int MAX_INDEX = GROESSE-1;`

Eine Zahl auf der rechten Seite zur Initialisierung einer Konstanten heißt »Zahlliteral«. Ein Literal für Zahlen muss den dafür erlaubten Regeln entsprechen. Ein Literal steht im Programmtext und ist daher nicht veränderbar. Die Bedeutung besteht nur in seinem Wert. Ein Literal für ganze Zahlen besteht nur aus einem optionalen Vorzeichen, gefolgt von Ziffern und möglicherweise einem Suffix, um den Typ zu spezifizieren, zum Beispiel `l` oder `L` für long-Zahlen.

Unveränderliche Größen sollten *stets* als `const` deklariert werden! Begründung: Der Compiler wird damit in die Lage versetzt, fehlerhafte Zuweisungen zu finden:

```
const float PI = 3.1415926;
// weiterer Programmtext
PI = 17.5; // ergibt eine Fehlermeldung des Compilers!
```

1.6.5 Zeichen

Zeichen sind in diesem Zusammenhang Buchstaben wie A, b, c, D, Ziffernzeichen wie 1, 2, 3 und Sonderzeichen wie ; , . ! , und andere. Dabei werden Zeichenkonstanten (= Zeichenlitterale) immer in Hochkommata eingeschlossen, also zum Beispiel `'a'`, `'1'`, `'?'` usw. Für Zeichen wird der Datentyp `char` bereitgestellt (Beispieldeklaration siehe einige Zeilen weiter unten). Ein Zeichen ist in diesem Sinne *stets* auch *nur* ein Zeichen, insbesondere sind Ziffernzeichen etwas anderes als die Ziffern selbst, das heißt `'1'` ist nicht `1`! Ein Zeichen wird intern als 1-Byte-Ganzzahl interpretiert (0 ... 255 `unsigned char` beziehungsweise -128 ... +127 `signed char`). Hier und im Folgenden wird angenommen, dass ein Byte aus 8 Bits besteht⁴. Die ASCII-Tabelle definiert die ersten 7 Bits eines Bytes, also 128 Zeichen (siehe Abschnitt A.3). Es gibt drei verschiedene `char`-Datentypen:

```
signed char
unsigned char
char // bedeutet systemabhängig unsigned oder signed
```

Zusätzlich gibt es »lange«-Zeichen (englisch *wide characters*), die den Typ `wchar_t` haben. »Wide characters« sind für Zeichensätze gedacht, bei denen ein Byte nicht zur Darstellung eines Zeichens ausreicht, zum Beispiel japanische Zeichen. Ein Zeichenliteral vom Typ `wchar_t` beginnt mit einem `L`, zum Beispiel `L'??'`.



Mehr über `wchar_t` und Zeichenlitterale lesen Sie in Abschnitt 31.2.

⁴ Dies muss nicht für jedes System gelten, ist aber verbreitet.

Alle Basisfunktionen der Standardbibliothek (Kapitel 26) gelten ebenso für `wchar_t` wie für `char`. Falls nicht ausdrücklich anders erwähnt, wird für `char` im Folgenden stets `signed char` angenommen. Die Art der Voreinstellung variiert von Compiler zu Compiler. Beispiele für Deklarationen und Zuweisungen:

```
const char STERN = '*';
char a;
a = 'a';
```

`a` und `'a'` haben hier eine verschiedene Bedeutung. `a` ist hier eine Variable, die nur dank der Zuweisung den Wert `'a'` hat. Ein anderer Wert wäre ebenso möglich:

```
a = 'x';
a = STERN;
```

Es gibt besondere Zeichenkonstanten der ASCII-Tabelle, die nicht direkt im Druck oder in der Anzeige sichtbar sind. Um sie darstellen zu können, werden sie als Folge zweier Zeichen geschrieben, nehmen aber dennoch ebenfalls nur ein Byte in Anspruch. Diese Zeichen heißen auch *Escape-Sequenzen*, weil `\` als Escape-Zeichen dient, um der normalen Interpretation als einzelnes Zeichen zu entkommen (englisch *to escape*). Außerhalb der ASCII-Tabelle gibt es ANSI-Escape-Sequenzen zur Bildschirmansteuerung, die mehrere Bytes umfassen können. Eine Hilfsdatei namens `ansi_esc.h` finden Sie im Verzeichnis `cppbuch/include` der Beispiele. Lesen Sie vor Benutzung den Anfang der Datei.

Tabelle 1.6 zeigt einige Beispiele. Das `endl` auf Seite 49 ist allerdings nicht als Zeichenkonstante zu verstehen, weil es zusätzlich für das sofortige Erscheinen der Zeile auf dem Bildschirm sorgt, was bei `'\n'` durch die gepufferte Ausgabe nicht immer so sein muss (siehe Kapitel 10, Stichwort `unibuf`).

Tabelle 1.6: Besondere Zeichenkonstanten (Escape-Sequenzen)

Zeichen	Bedeutung	ASCII-Name
<code>\a</code>	Signalton	BEL
<code>\b</code>	Backspace	BS
<code>\f</code>	Seitenvorschub	FF
<code>\n</code>	neue Zeile	LF
<code>\r</code>	Zeilenrücklauf	CR
<code>\t</code>	Tabulator	HT
<code>\v</code>	Zeilensprung	VT
<code>\\</code>	Backslash	
<code>\'</code>	'	
<code>\"</code>	"	
<code>\o</code>	◊ = Platzhalter: ◊ = Folge von Oktalziffern, Beispiel: <code>\377</code>	
<code>\0</code>	Spezialfall davon (Nullbyte)	NUL
<code>\x◊, \X◊</code>	◊ = Zeichenfolge aus Hex-Ziffern, Beispiel: <code>\xDB</code> oder <code>\x1f</code>	

Da ein ASCII-Zeichen genau *ein* Byte beansprucht, ist ein Zeichen der Oktaldarstellung \777 nicht erlaubt ($377 = 255_{10}$)⁵. Die Zuordnung von Zeichen und Zahl geschieht über die ASCII-Tabelle, über die eine Ordnungsrelation definiert ist: ...'0' < '1' < .. < '9' < .. < 'A' < .. < 'Z' < .. < 'a' < .. < 'z'... Ein Zeichen hat eine eindeutige Position innerhalb der ASCII-Tabelle (siehe Seite 887).

Genau genommen definiert die ASCII-Tabelle nur alle Zeichen mit 7 Bits, insgesamt 128. Der Datentyp `char` stellt 8 Bits, also 1 Byte zur Verfügung, sodass 256 Zeichen darstellbar sind. Die 128 zusätzlichen Zeichen sind nicht genormt, sondern unterscheiden sich für verschiedene Rechner- und Betriebssystemtypen. Meistens werden in diesen 128 Zusatzzeichen Blockgrafiksymbole und nationale Sonderzeichen wie ä, ö, ß untergebracht.



Mehr über das Thema Zeichensatz lesen Sie in Abschnitt 31.2.

Die Position eines Zeichens in der erweiterten Tabelle kann über die Umwandlung in eine `int`-Zahl bestimmt werden, ebenso wie aus einer Position über die Umwandlung in `char` das zugehörige Zeichen ermittelt werden kann. Die Umwandlung geschieht einfach über den `static_cast`-Operator mit Angabe des gewünschten Datentyps in spitzen und Angabe der Variable in runden Klammern. Die Typumwandlung heißt in der englischsprachigen Literatur *type cast* oder einfach *cast*. Der `static_cast`-Operator verlangt bestimmte, in [ISOC++] festgelegte Verträglichkeiten zwischen den zu wandelnden Typen.

```
char c;
int i;
i = static_cast<int>(c); // Typumwandlung char → int
```

bedeutet, dass der Wert der Variablen `i` nun eine `int`-Repräsentation der `char`-Variablen `c` ist. Andere, einfachere Schreibweisen sind ebenfalls möglich:

```
i = int(c);           // oder
i = (int) c;
```

Weil ein `char` vom Compiler wie eine 1-Byte-`int`-Zahl interpretiert wird, ist auch eine implizite Typumwandlung möglich:

```
i = c;
```

Die einfacheren Schreibweisen werden jedoch nicht empfohlen, weil sie in bestimmten Zusammenhängen unsicherer sind. Eine Begründung wird in Abschnitt 7.9 gegeben. Die Schreibweise `int(c)` erinnert an die später zu besprechenden Funktionen. Die aus der Sprache C übernommene traditionelle Schreibweise `(int) c` bewirkt genau dasselbe und ist im Gegensatz zur Funktionsschreibweise auch für komplexere Datentypen möglich. Hier wird von `int` nach `char` und zurück gewandelt:

```
i = 66;
c = static_cast<char>(i); // Typumwandlung int -> char
cout << c;               // 'B'
c = 'I';                 // Das Ziffernzeichen '1' hat die Position
i = static_cast<int>(c); // 49 innerhalb der ASCII-Tabelle:
cout << i;               // 49
```

⁵ Es sind beliebig viele Oktal- oder Hexadezimalziffern erlaubt, wenn der Zeichentyp (z.B. `wchar_t`) sie darstellen kann.

Es gelten die Identitäten

```
c == static_cast<char>( static_cast<int>(c)); und
i == static_cast<int>( static_cast<char>(i));,
```

falls $-128 \leq i \leq 127$ ist (beziehungsweise $0 \leq i \leq 255$ bei unsigned char). Falls *i* außerhalb dieses Bereichs liegt, gibt es einen Datenverlust, weil die überzähligen Bits bei der Umwandlung nicht berücksichtigt werden können. Mehr zu Standard-Typumwandlungen erfahren Sie auf Seite 57. Wie kann man aus einem Ziffernzeichen die repräsentierte Ziffer erhalten? Da die Folge der Ziffernzeichen in der ASCII-Tabelle mit '0' beginnt, genügt es, '0' abzuziehen:

```
char c = '5';
int ziffer = c - '0';    // implizite Typumwandlung!
cout << ziffer << endl; // 5
ziffer = static_cast<int>(c) - static_cast<int>('0'); // explizite Typumwandlung!
```

Weil ein char vom Compiler wie eine 1-Byte-int-Zahl interpretiert wird, ist das Rechnen ohne explizite Typumwandlung möglich. Zum Vergleich sind beide Möglichkeiten angegeben.

Operatoren für Zeichen

Da der Datentyp char intern als 1-Byte-Ganzzahl dargestellt wird, sind eigentlich alle Ganzzahl-Operatoren (siehe Tabelle 1.2 auf Seite 45) möglich, aber im Sinne der Bedeutung von *Zeichen* sind nur die Operatoren aus Tabelle 1.7 sinnvoll.

Tabelle 1.7: Operatoren für char

Operator	Beispiel	Bedeutung
=	d = 'A'	Zuweisung
<	d < f	kleiner als
>	d > f	größer als
<=	d <= f	kleiner gleich
>=	d >= f	größer gleich
==	d == f	gleich
!=	d != f	ungleich

1.6.6 Logischer Datentyp bool

Ein logischer Datentyp wird zur Erinnerung an den englischen Mathematiker George Boole (1815–1864) mit bool bezeichnet. Boole hat die später nach ihm benannte Boolesche Algebra entwickelt. Variablen eines logischen Datentyps können nur die Wahrheitswerte *wahr* (englisch *true*) beziehungsweise *falsch* (englisch *false*) annehmen. Falls notwendig, wird der Datentyp bool zu int gewandelt, wobei false der Wert 0 ist und true der Wert 1. Das folgende Programmstück gibt eine 1 aus, falls das eingelesene Zeichen ein Großbuchstabe war, ansonsten eine 0.

```
bool istGrossBuchstabe;
char c;
cin >> c;
istGrossBuchstabe = (c >= 'A') && (c <= 'Z');
```

```
cout << istGrossBuchstabe;    // Wandlung in int
```

Die Ausgabe als Text *true* bzw. *false* kann eingestellt werden:

```
cout.setf(ios_base::boolalpha); // Textformat einschalten
cout << istGrossBuchstabe;    // Wandlung in Text
```

Zunächst werden die Klammern ausgewertet, die jeweils für sich Wahrheitswerte von *false* oder *true* ergeben. Die Relationen *>=* und *<=* beziehen sich dabei auf die ASCII-Tabelle. Es wird also geprüft, ob das Zeichen *c* gleich dem Zeichen 'A' ist oder in der Tabelle nach ihm folgt, und ob es gleich dem Zeichen 'Z' ist oder in der Tabelle vor dem 'Z' liegt. Die Wahrheitswerte werden dann durch das logische UND (*&&*) verbunden, nicht zu verwechseln mit dem bitweisen UND (*&*) der Tabelle 1.2 auf Seite 45. Das Ergebnis wird der Variablen *istGrossBuchstabe* zugewiesen. Die Klammern sind hier nur zur Verdeutlichung angegeben. Sie können entfallen, weil die relationalen Operatoren eine höhere Priorität als die logischen haben. Tabelle 1.8 zeigt Operatoren für logische Datentypen. Der Datentyp *bool* wird an allen Stellen, die nicht ausdrücklich *bool* verlangen, nach *int* gewandelt (siehe obiges Beispiel). Dabei wird *true* zu 1 und *false* zu 0. Die umgekehrte Wandlung von *int* nach *bool* ergibt *false* für 0 und *true* für alle anderen *int*-Werte. Hier ist die Wirkungsweise der *logischen Negation* zu sehen:

Tabelle 1.8: Operatoren für logische Datentypen

Operator	Beispiel	Bedeutung
!	!i	logische Negation
&&	a && b	logisches UND
	a b	logisches ODER
==	a == b	Vergleich
!=	a != b	Vergleich
=	a = a && b	Zuweisung

```
bool wahrheitswert = true;
wahrheitswert = !wahrheitswert;    // Negation
cout << wahrheitswert << endl;    // 0, d.h. false
// Beispiel mit int-Zahlen: Aus 0 wird 1 und aus einer Zahl ungleich 0
// wird durch die Negation eine 0:
int i = 17;
int j = !i;    // 0 (implizite Typumwandlung)
i = !j;    // 1 (implizite Typumwandlung)
// Typumwandlung von int nach bool
wahrheitswert = 99;    // true
wahrheitswert = 0;    // false
```

1.6.7 Referenzen

Eine *Referenz* ist ein Datentyp, der einen *Verweis* auf ein Objekt liefert. Referenzen werden in C++ häufig zur Parameterübergabe benutzt; nähere Erklärungen und umfangreichere Beispiele finden sich erst einige Kapitel später (ab Kapitel 3). Eine Referenz bildet einen *Alias-Namen* für ein Objekt, über den es ansprechbar ist. Ein Objekt hat damit zwei Namen! Der Compiler »weiß« aufgrund der Deklaration, dass es sich um eine Referenz

handelt, und nicht etwa um ein neues Objekt. Um eine Variable als Referenz zu deklarieren, wird das &-Zeichen benutzt, das neben dem bitweisen UND und dem (noch nicht benutzten) Adressoperator nun die dritte Bedeutung hat. Beispiele:

```
int i = 2;
int j = 9;
int& r = i;    // Referenz auf i (r ist ein Alias für i)
r = 10;        // ändert i
r = j;        // Wirkung: i = j;
```

Wo das &-Zeichen zwischen `int` und `r` steht, ist unerheblich. In diesem Buch wird die Schreibweise `int& r` bevorzugt, um zu verdeutlichen, dass die Referenzeigenschaft zum Typ gehört. Eine Referenz wird genau wie eine Variable benutzt, der Compiler weiß, dass sie ein Alias-Name ist. Referenzen müssen bei der Deklaration initialisiert werden. Es ist nicht möglich, eine Referenz nach der Initialisierung so zu ändern, dass sie ein Alias-Name für eine andere Variable als die erstzugewiesene wird. Die Deklaration `int& s = r;` könnte vordergründig so interpretiert werden, dass die Referenz `s` eine Referenz auf `r` wäre, weil sie ja mit `r` initialisiert wird. Der Compiler setzt aber, wie oben beschrieben, auf der rechten Seite für `r` das referenzierte Objekt `i` ein. `s` ist daher nur ein weiterer Alias-Name für `i`. Mit anderen Worten, wenn nach den obigen Deklarationen einer der Namen `i`, `r` oder `s` benutzt wird, könnte man ihn durch einen der anderen beiden ersetzen, ohne dass ein Programm in seiner Bedeutung geändert wird. Zusammengefasst:

- Auf Objekte wird nur über symbolische Namen (Bezeichner) oder Zeiger zugegriffen. Zeiger (in Kapitel 5 beschrieben) seien hier ausgeklammert.
- Die Bezeichner (Namen) von Referenzen sind nichts anderes als Alias-Namen. Für ein Objekt kann es keinen oder beliebig viele Alias-Namen geben, die wie andere Bezeichner auch verwendet werden.
- Alle Bezeichner für dasselbe Objekt sind in der Verwendung semantisch gleichwertig. Die obige Deklaration `int& s = r;` hat daher dieselbe Wirkung wie `int& s = i;`.

1.6.8 Regeln zum Bilden von Ausdrücken

Es gelten im Allgemeinen Vorrangregeln der Algebra beim Auswerten eines Ausdrucks inklusive der Klammerregeln. Die Tabelle 1.9 zeigt die Rangfolge einiger ausgewählter Operatoren. Einige der Operatoren werden erst in folgenden Kapiteln erklärt. Eine ausführliche Auflistung der Operatorenrangfolge finden Sie im Anhang A.4, Seite 890.

Auf gleicher Prioritätsstufe wird ein Ausdruck von links nach rechts abgearbeitet (linksassoziativ), mit Ausnahme der Ränge 2, 14 und 15 der Tabelle, die von rechts abgearbeitet werden (rechtsassoziativ). Zuerst werden jedoch die Klammern ausgewertet. Beispiel:

```
a = b + d + c;    ist gleich mit:  a = ((b + d) + c);    → linksassoziativ
a = b = d = c;    ist gleich mit:  a = (b = (d = c));    → rechtsassoziativ
```

Die Reihenfolge der Auswertung von Unterausdrücken untereinander, also auch Klammerausdrücken, ist jedoch undefiniert. Daher sollten Ausdrücke vermieden werden, die einen Wert sowohl verändern als auch benutzen. Zwei Beispiele:

```
int total = 0;
int sum = (total = 3) + (++total); // Fehler!
int i = 2;
i = 3 * i++;                       // Fehler !
```

Tabelle 1.9: Präzedenz ausgewählter Operatoren

Rang	Operatoren
1	. [] f () (Funktionsaufruf) ++ -- (postfix)
2	sizeof ++ -- (präfix) ~ ! + - (unär) (Typ) Ausdruck (C-Stil-Typumwandlung)
4	* / %
5	+ - (binär)
6	<< >>
7	< > <= >=
8	== !=
9	& (bitweises UND)
10	^ (bitweises exklusiv-ODER)
11	(bitweises ODER)
12	&& (logisches UND)
13	(logisches ODER)
14	? : (Bedingungsoperator)
15	alle Zuweisungsoperatoren =, +=, <<= usw.

Die Variable `sum` kann hier den Wert 4 oder den Wert 7 annehmen, abhängig von der Reihenfolge, in der die Unterausdrücke in den Klammern berechnet werden. Der Wert von `i` ist undefiniert, weil die Reihenfolge der Auswertung nicht feststeht. Es gibt zwei Möglichkeiten:

1. `3*i` wird berechnet und ergibt 6. Dieser Wert wird `i` zugewiesen. Erst anschließend wird `i` inkrementiert, sodass zum Schluss `i` gleich 7 gilt.
2. `3*i` wird berechnet und ergibt 6. Gleich nach der Berechnung wird `i` von 2 auf 3 inkrementiert. Erst dann erfolgt die Zuweisung des berechneten Ergebnisses an `i`, sodass zum Schluss `i` gleich 6 gilt.

1.6.9 Standard-Typumwandlungen

Standard-Typumwandlungen sind implizite Typumwandlungen für eingebaute Typen. Implizit heißt, dass eine Typumwandlung nicht ausdrücklich hingeschrieben wird und der Compiler dennoch keine Fehlermeldung bei der Typumwandlung gibt. Es kann auch eine Folge von hintereinandergeschalteten Standard-Typumwandlungen geben. In diesem Abschnitt werden die wichtigsten Möglichkeiten beschrieben. Im Einzelfall kann ein Informationsverlust auftreten. Mögliche Ursachen:

- Genauigkeitsverlust, weil zum Beispiel eine `float`-Zahl nicht so viele Mantissen-Bits wie eine `double`-Zahl hat (siehe Aufstellung Seite 47).
- Überschreitung des Grenzbereichs, weil der mögliche Exponent einer `float`-Zahl kleiner als der einer `double`-Zahl ist.
- Verlust des Nachkommateils bei der Umwandlung z.B. `double` nach `int`.
- Umwandlung z.B. einer zu großen `double`- oder `float`-Zahl in einen integralen Typ.
- Die Bitzahl reicht nicht, etwa wenn eine den `int`-Bereich überschreitende `long`-Zahl in eine `int`-Zahl umgewandelt wird. Dasselbe gilt für die Umwandlung einer Zahl, die größer als 127 ist, in den Typ `signed char` (bzw. `> 255` in `unsigned char`).

- Vorzeichenverlust und gleichzeitige Wertänderung, wenn zum Beispiel eine negative `int`-Zahl in eine `unsigned int`-Zahl umgewandelt wird.

Typumwandlung integraler Typen nach `int`

Jeder R-Wert (Rechts-Wert, (englisch *rvalue*), vgl. Seite 62) des Typs `char`, `signed char`, `unsigned char`, `short int` oder `unsigned short int` kann ohne Informationsverlust in einen `int`-Wert umgewandelt werden (englisch *integral promotion*).

Typumwandlung integraler Typen untereinander

Integrale Typen sind untereinander konvertierbar. Im C++-Standard werden alle Fälle, die nicht zur obigen »integral promotion« gehören, *integral conversion* genannt. Sonderfälle sind die Typen `bool` (siehe Seite 55) und `enum` (siehe Seite 80).

float-Typumwandlungen

Jeder R-Wert des Typs `float` kann in einen `double`-Wert oder einen Integer-Wert umgewandelt werden und umgekehrt. Für `bool` gilt ähnlich wie im `int`-Fall: `true` wird 1.0, `false` wird 0.0, 0.0 wird `false`, und ein `float`- bzw. `double`-Ausdruck ungleich 0 wird `true`.

1.7 Gültigkeitsbereich und Sichtbarkeit

In C++ gelten Gültigkeits- und Sichtbarkeitsregeln für Namen. Es gibt folgende Regeln:

- Namen sind nur *nach der Deklaration* und nur *innerhalb des Blocks* gültig, in dem sie deklariert wurden. Sie sind *lokal* bezüglich des Blocks. Zur Erinnerung: Ein Block ist ein Programmbereich, der durch ein Paar geschweifeter Klammern `{ }` eingeschlossen wird. Blöcke können verschachtelt sein, also selbst wieder Blöcke enthalten.
- Namen von Variablen sind auch gültig für innerhalb des Blocks neu angelegte innere Blöcke.
- Die Sichtbarkeit (englisch *visibility*) zum Beispiel von Variablen wird eingeschränkt durch die Deklaration von Variablen gleichen Namens. Für den Sichtbarkeitsbereich der inneren Variablen ist die äußere unsichtbar.

Der Datenbereich für lokale Daten wird bei Betreten des Gültigkeitsbereichs auf einem besonderen Speicherbereich mit dem Namen *Stack* angelegt und am Ende des Gültigkeitsbereichs, also am Blockende, wieder freigegeben. Der Stack (deutsch: »Stapel«, auch Kellerspeicher) ist ein Bereich mit der Eigenschaft, dass die zuletzt darauf abgelegten Elemente zuerst wieder freigegeben werden (*last in, first out*). Damit lässt sich das beschriebene Anlegen von Variablen bei Blockbeginn und ihre Freigabe bei Blockende gut verwalten, ohne dass *wir* uns darum kümmern müssen.

Auf verschiedene Arten der Sichtbarkeitsbereiche (Funktionen, Dateien, Klassen) wird später eingegangen. Das folgende Programm zeigt Beispiele für verschiedene Gültigkeits-

bereiche (englisch *scope*). Der im Programm verwendete Operator `::` bewirkt den Zugriff auf Variablen, die *global* sichtbar sind, also nicht Zugriff auf den nächsten äußeren Block.

Listing 1.7: Beispielprogramm: Variablen und Blöcke

```
// cppbuch/k1/bloেকে.cpp
#include<iostream>
using namespace std;
// a und b werden außerhalb eines jeden Blocks deklariert. Sie sind damit innerhalb
// eines jeden anderen Blocks gültig und heißen daher globale Variablen.
int a = 1, b = 2;

int main() { // Ein neuer Block beginnt.
    cout << "globales a= " << a << endl; // Ausgabe von a
    // Innerhalb des Blocks wird eine Variable a deklariert. Ab jetzt ist das globale a noch
    // gültig, aber nicht mehr unter dem Namen a sichtbar, wie die Folgezeile zeigt.
    int a = 10;
    // Der Wert des lokalen a wird ausgegeben:
    cout << "lokales a= " << a << endl;
    // Das globale a lässt sich nach der Deklaration des lokalen a nur noch mithilfe des
    // Bereichsoperators :: (englisch scope operator) ansprechen. Ausgabe von ::a :
    cout << "globales ::a= " << ::a << endl;
    { // Ein neuer Block innerhalb des bestehenden beginnt.
        int b = 20;
        // Variable b wird innerhalb dieses Blocks deklariert. Damit
        // wird das globale b zwar nicht ungültig, aber unsichtbar.
        int c = 30; // c wird innerhalb dieses Blocks deklariert.
        // Die Werte von b und c werden ausgegeben.
        cout << "lokales b = " << b << endl;
        cout << "lokales c = " << c << endl;
        // Wie oben beschrieben, ist das globale b nur über den
        // Scope-Operator ansprechbar. Ausgabe von ::b:
        cout << "globales ::b = " << ::b << endl;
    } // Der innere Block wird geschlossen. Damit ist das globale b
    // auch ohne Scope-Operator wieder sichtbar:
    cout << "globales b wieder sichtbar: b = " << b << endl;
    // cout << "c = " << c << endl; // Fehler, siehe Text
} // Ende des äußeren Blocks
```

Das Programm wird im Kommentar zeilenweise erklärt. Es zeigt, dass Gültigkeit und Sichtbarkeit nicht das Gleiche sind, und erzeugt folgende Ausgabe:

```
globales a= 1
lokales a= 10
globales ::a= 1
lokales b = 20
lokales c = 30
globales ::b = 2
globales b wieder sichtbar: b = 2
```

Die Kommentarzeichen `//` in der letzten Programmzeile sind erforderlich, weil der Compiler diese Zeile sonst als fehlerhaft bemängeln würde. Grund: Durch Schließen des inneren Blocks ist der Gültigkeitsbereich aller in diesem Block deklarierten Variablen beendet, also ist `c` außerhalb des Blocks unbekannt.

Vermeiden Sie lokale Objekte mit Namen, die Objekte in einem äußeren Gültigkeitsbereich verdecken! Die Verständlichkeit eines Programms wird durch verschiedene Objekte mit demselben Namen erschwert, wie das Beispiel hoffentlich zeigt.

1.7.1 Namespace std

Eine weitere Möglichkeit zur Schaffung von Sichtbarkeitsbereichen sind *Namensräume* (englisch *namespaces*). Bisher wurde der zur C++-Standardbibliothek gehörende Namensraum `std` benutzt, wie Sie an den Zeilen `using namespace std;` in den Beispielen gesehen haben. Namensräume spielen bei der Benutzung verschiedener Bibliotheken eine Rolle. Einzelheiten werden weiter unten in Abschnitt 3.6 (ab Seite 141) erklärt. Hier soll nur vorab darauf hingewiesen werden, dass man auch ohne pauschale Nutzung der Standardbibliothek auskommt, wenn die betreffenden Elemente mit einem sogenannten qualifizierten Namen angesprochen werden, der den Namensraum angibt. Bezogen auf den Standard-Namensraum `std` gibt es im Wesentlichen drei Möglichkeiten:

```
// 1. Pauschale Nutzung
using namespace std; // macht alles aus std ab jetzt bekannt
// ... ggf. weiterer Programmtext
cout << "Ende" << endl;
```

oder

```
// 2. Nutzung von cout und endl aus std mit qualifizierten Namen:
// using namespace std; sei nicht deklariert.
std::cout << "Ende" << std::endl; // richtig
cout << "Ende" << endl; // Fehlermeldung des Compilers!
```

oder

```
// 3. Deklaration ausgewählter Teile; using namespace std; sei nicht deklariert.
using std::cout;
using std::endl;
cout << "Ende" << endl;
```

Die erste Möglichkeit wird in den `main()`-Programmen dieses Buchs bevorzugt, weil sie Schreibarbeit spart. Die zweite oder dritte Möglichkeit wird in allen anderen Fällen empfohlen. Sie werden in den Beispielen daher alle drei Varianten antreffen.



Übung

1.2 Schreiben Sie ein Programm, das die größtmögliche `unsigned int`-Zahl (`int`, `long`, `unsigned long`) ausgibt, *ohne* dass die Kenntnis der systemintern verwendeten Bitanzahl für jeden Datentyp benutzt wird. Hinweis: Studieren Sie die möglichen Operatoren für ganze Zahlen und die Datei `limits.h`.

1.8 Kontrollstrukturen

Kontrollstrukturen dienen dazu, den Programmfluss zu steuern. Im einfachsten Fall werden Anweisungen eines Programms eine nach der anderen in derselben Reihenfolge ausgeführt, wie sie hingeschrieben worden sind. Dies ist nicht immer erwünscht. Manchmal ist es notwendig, dass der Programmfluss sich in Abhängigkeit von den Daten ändern soll, oder es müssen Teile des Programms wiederholt durchlaufen werden. Erst mit Kontrollstrukturen lassen sich überhaupt Programme von einiger Komplexität schreiben.

1.8.1 Anweisungen

In den folgenden Abschnitten wird des Öfteren der Begriff »Anweisung« gebraucht, der deswegen an dieser Stelle erläutert werden soll. Eine Anweisung kann unter anderem⁶ sein:

- eine Deklarationsanweisung
- eine Ausdrucksanweisung
- eine Schleifenanweisung
- eine Auswahlanweisung
- eine Verbundanweisung, auch Block genannt.

Deklarationsanweisung

Eine Deklarationsanweisung führt einen Namen in das Programm ein. Sie kann in verschiedenen Formen vorkommen, unter denen die einfachen Deklarationen wie zum Beispiel `int x;` die häufigsten sind. Nach dieser Deklaration ist das Objekt `x` in einem Programm bekannt und kann benutzt werden. Eine einfache Deklaration wird stets mit einem Semikolon `;` abgeschlossen.

Ausdrucksanweisung

Eine Ausdrucksanweisung ist ein Ausdruck (siehe Seite 42), gefolgt von einem Semikolon. Ein Ausdruck repräsentiert nach der Auswertung einen Wert. Zum Beispiel kann der Ausdruck `x == 1` den Wert `true` oder `false` annehmen. In C++ ist mit einer Ausdrucksanweisung in der Regel eine Aktivität verbunden, zum Beispiel eine Zuweisung (siehe unten) wie `x = 3;`. Eine Zuweisung hat einen Wert, weswegen verkettete Zuweisungen möglich sind:

```
a = b = c;
```

meint, dass `b` (= der Wert der Zuweisung `b = c`) der Variablen `a` zugewiesen wird. Der Wert der letzten Zuweisung `a = b` wird nicht mehr verwendet. Selbst eine Ausgabe auf den Bildschirm ist ein Ausdruck. Der Wert ist das Objekt `cout` selbst, das die Ausgabe bewerkstelligt.

```
cout << a << b;
```

⁶ Eine vollständige Auflistung ist nicht beabsichtigt.

bedeutet dasselbe wie

```
(cout << a) << b;
```

Das Ergebnis des Ausdrucks in den runden Klammern ist `cout`, weswegen der zweite Teil der Ausgabe als `cout << b`, gelesen werden kann. Ein alleinstehendes Semikolon ist eine Leeranweisung.

Zuweisung

Eine Zuweisung ist ein Spezialfall einer Ausdrucksanweisung. Ein Zuweisungsausdruck, zum Beispiel `a = b`, besteht aus drei Teilen:

- Einem linken Teil, auch *L-Wert* (englisch *lvalue*) genannt, was eine Abkürzung für *Links-Wert* ist.
- Dem Zuweisungsoperator `=`.
- Einem rechten Teil, auch *R-Wert* (englisch *rvalue*) genannt, was eine Abkürzung für *Rechts-Wert* ist.

Dabei wird der L-Wert⁷ als (symbolische) *Adresse*, der R-Wert als *Wert* interpretiert. Die Bedeutung einer Zuweisung `a = b`; ist also: Der Wert der Variablen `b` wird an die Adresse der Variablen `a` kopiert, sodass danach Variable `a` denselben Wert hat.

Schleifenanweisung

Diese Anweisungen sind mit den Schlüsselwörtern `for`, `while` und `do ... while` verbunden. Einzelheiten folgen in den nächsten Abschnitten.

Auswahlanweisung

Diese Anweisungen sind mit den Schlüsselwörtern `if` und `switch` verbunden. Einzelheiten folgen in den nächsten Abschnitten.

Verbundanweisung, Block

Eine Verbundanweisung, auch Block genannt, ist ein Paar geschweifte Klammern, die eine Folge von Anweisungen enthalten. Die Folge kann leer sein. Die enthaltenen Anweisungen können selbst wieder Verbundanweisungen sein.

```
{ }           // leerer Block

{             // Block mit einer Anweisung
    Anweisung
}

{             // Block mit zwei Anweisungen
    Anweisung1
    Anweisung2
}
```

⁷ Mehr dazu im Glossar auf Seite [953](#).

1.8.2 Sequenz (Reihung)

Im einfachsten Fall werden die Anweisungen der Reihe nach durchlaufen:

```
a = b + 1;
a += a;
cout << "\n Ergebnis =" << a;
```

Nebenbei sehen wir, dass das Zeichen '\n' in eine Zeichenkette eingebaut werden kann, sodass vor der Ausgabe von *Ergebnis* eine neue Zeile auf dem Bildschirm begonnen wird.

1.8.3 Auswahl (Selektion, Verzweigung)

Häufig hängt die Ausführung von Anweisungen von einer Bedingung ab. C++ stellt für solche Zwecke die `if`-Anweisung bereit.

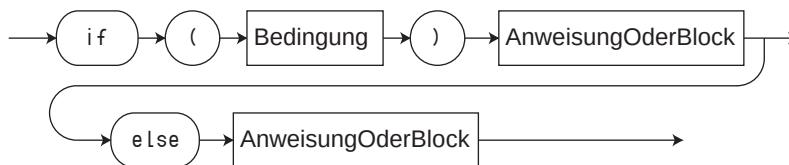
```
if (Bedingung)
    Anweisung1
```

bedeutet, dass *Anweisung1* nur dann ausgeführt wird, wenn die *Bedingung* wahr ist, das heißt zu einem Ausdruck mit dem Wert `true` (oder ungleich 0) ausgewertet wird. Die Bedingung kann ein arithmetisches Ergebnis haben. Alternativ kann eine zweite Anweisung angegeben werden, sodass *Anweisung1* ausgeführt wird, falls die *Bedingung* wahr ist, und andernfalls *Anweisung2* (`else`-Zweig):

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

Zu einem `if`- oder `else`-Zweig gehört stets nur genau *eine* Anweisung! Diese kann natürlich eine Verbundanweisung (Block) sein, also ein Paar geschweifter Klammern, die beliebig viele Anweisungen umschließen können, also auch keine oder nur eine. Abbildung 1.6 zeigt das Syntaxdiagramm einer `if`-Anweisung.

if-Anweisung:



AnweisungOderBlock:

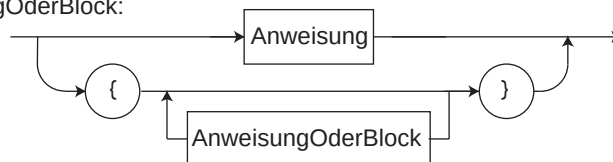


Abbildung 1.6: Syntaxdiagramm einer `if`-Anweisung

Ein Semikolon nach einem Block bedeutet eine zweite (leere) Anweisung. Diese ist immer unnütz und im Fall eines folgenden `else` sogar falsch. Beispiele für `if`-Anweisungen:

```
if (x < 100)
    cout << "x < 100";           // nur eine Anweisung
else
    cout << "x >= 100";

if (a > b) {
    x = a - b;                   // zwei Anweisungen im Block
    y = 2*a;
}

if ((c >= 'A') && (c <= 'Z')) { // if-Anweisung mit else
    istGrossBuchstabe = true;
}
else {
    istGrossBuchstabe = false;
}
```

Die letzte `if`-Anweisung ist gleichwertig mit: `istGrossBuchstabe = c >= 'A' && c <= 'Z'`,. Der Bedingungsausdruck muss vom Typ `bool` sein oder in `bool` umgewandelt werden können. Relationale Ausdrücke wie `(a < b)` werden zu `true` ausgewertet, falls `a < b` ist, ansonsten zu `false`. `if(a == 0)` kann daher auch als `if(!a)` geschrieben werden: `a` ist eine Zahl vom Typ `int`, die in einen Wahrheitswert umgewandelt wird. Dabei wird ein Wert, der gleich 0 ist, in `false` und ein Wert ungleich 0 in `true` umgewandelt.

Auf dieses Ergebnis wird der Negationsoperator `!` angewendet, womit sich das gewünschte Verhalten ergibt. Bedingungsausdrücke werden von links nach rechts ausgewertet. Dabei werden unnötige Berechnungen übersprungen. Damit ist gemeint, dass in einer Bedingung, die aus mehreren ODER-Verknüpfungen besteht, die Berechnung nach dem ersten Ergebnis abgebrochen werden kann, das den Wahrheitswert *wahr* liefert. Grund ist, dass sich das Ergebnis nach weiteren Berechnungen nicht ändern kann. Umgekehrt braucht man bei UND-Verknüpfungen nicht weiterzurechnen, sobald auch nur ein Teilergebnis *falsch* liefert. Diese Art der Auswertung wird *Kurzschlussauswertung* genannt (englisch *short circuit evaluation*).

Mit Teilbedingungen verbundene *Seiteneffekte* werden daher nur bei Auswertung der jeweiligen Teilbedingung ausgeführt. Seiteneffekte sind Ergebnisse, die zusätzlich zum eigentlichen Zweck, gewissermaßen nebenbei, entstehen. In den folgenden Beispielen ist die Hauptsache die Auswertung der Bedingung. Als Seiteneffekt werden *nach* Auswertung der Bedingung, aber noch *vor* Ausführung des nachfolgenden Programmcodes, `i` beziehungsweise `j` durch den Operator `++` modifiziert, sofern es nötig ist, die Teilbedingung zu berechnen. Vollziehen Sie die Beispiele nach! Das Ergebnis ist im Kommentar aufgeführt. Das Beispiel ist nur zur Übung gedacht. Im Allgemeinen sind Seiteneffekte zu vermeiden!

```
int i = 0, j = 2; // stimmt's oder nicht?
if (i++ || j++) i++; // i == 2 und j == 3
```

```
int i = 1, j = 2;
if (i++ || j++) i++; // i == 3 und j == 2
```

```
int i = 0, j = 2;
if(i++ && j++) i++; // i == 1 und j == 2
```

```
int i = 1, j = 2;
if(i++ && j++) i++; // i == 3 und j == 3
```

if-Anweisungen können beliebig tief geschachtelt werden. Das Beispielprogramm hat die Aufgabe, ein Zeichen einzulesen und zu prüfen, ob es einer römischen Ziffer entspricht. Falls ja, soll die zugehörige arabische Zahl angezeigt werden, falls nein, eine passende Meldung.

Listing 1.8: Umwandlung römischer Ziffern mit if / else

```
// cppbuch/k1/roemzif1.cpp
#include<iostream>
using namespace std;

int main( ) {
    int a = 0;
    char c;
    cout << "Zeichen ?";
    cin >> c;
    if(c == 'I')    a = 1;
    else if(c == 'V') a = 5;
    else if(c == 'X') a = 10;
    else if(c == 'L') a = 50;
    else if(c == 'C') a = 100;
    else if(c == 'D') a = 500;
    else if(c == 'M') a = 1000;
    if(a == 0) {
        cout << "keine römische Ziffer!\n";
    }
    else {
        cout << a << endl;
    }
}
```

Achtung, Falle: Fehler in Verbindung mit if

Ein häufiger Fehler ist die versehentlich falsche Schreibweise des Gleichheitsoperators, sodass sich unfreiwillig der Zuweisungsoperator ergibt.

```
if(a = b) {           // Vorsicht! Vermutlich anders gemeint!
    cout << "a ist gleich b";
}
```

bewirkt, dass *a ist gleich b* immer dann ausgegeben wird, wenn b ungleich 0 ist. Die richtige Schreibweise ist:

```
if(a == b) {
    cout << "a ist gleich b";
}
```

Die Verwendung von `=` statt `==` hat die Wirkung einer *Zuweisung*. Zunächst erhält `a` den Wert von `b`. Das *Ergebnis* dieses Ausdrucks, nämlich `a`, wird dann als logische Bedingung interpretiert. Die falsche Schreibweise führt also nicht nur zu einem falschen Ergebnis für die Bedingung, sondern auch zur nicht beabsichtigten Änderung des Wertes von `a`. Freundliche Compiler geben an solchen Stellen eine Warnung aus, damit man sich noch einmal überlegen kann, ob man wirklich eine Zuweisung gemeint hat.

Eine weitere Gefahr sind Mehrdeutigkeiten durch falschen Schreibstil, das heißt falsche, wenn auch richtig gemeinte Einrückungen. Ohne Klammerung gehört ein `else` immer zum letzten `if`, dem kein `else` zugeordnet ist:

```
if(x == 1)
    if(y == 1)
        cout << "x == y == 1 !";
else
    cout << "x != 1";           // falsch
```

Trotz der augenfälligen Übereinstimmung des Zeilenanfangs der untersten Zeile mit dem Zeilenanfang von `if(x == 1)` gehört das `else` syntaktisch zur `if(y == 1)`-Zeile! Richtig ist:

```
if(x == 1) {
    if(y == 1) {
        cout << "x == 1 und y == 1 !";
    }
}
else {
    cout << "x != 1";           // nun korrekt
}
```

Auch einzelne Anweisungen nach `if` bzw. `else` sollten daher immer geklammert werden. Ein weiterer gelegentlicher Fehler aus der Praxis ist ein überflüssiges Semikolon (also eine Leeranweisung) nach der Bedingung, das beim Lesen leicht übersehen wird.

```
if(a == b);                     // Fehler!
    cout << "a ist gleich b";
```

Auf diese Art wird die `if`-Abfrage zwar durchgeführt, aber ohne Folgen, da sie bereits beim ersten Semikolon endet. Die anschließende Ausgabe wird also in jedem Fall durchgeführt, da die Ausgabe nun eine eigenständige Anweisung ist und somit nicht mehr zur `if`-Anweisung gehört. Solche Fälle akzeptiert der Compiler widerspruchsfrei!

Abschließend die Empfehlung, `int` und `unsigned` nicht zu mischen. Eine `unsigned`-Zahl kann nicht negativ sein. Wenn so ein Wert unüberlegt mit einem `int`-Wert verglichen wird, kann es schwer zu entdeckende Fehler geben:

```
int i = -1;
unsigned int u = 0;

if(u < i) {
    cout << u << ' ' << i << endl;
}
```

Da diese Art der Verwendung prinzipiell zulässig ist, wird ein Compiler sie akzeptieren und allenfalls eine Warnung ausgeben. Ganz klar ist $0 > -1$, dennoch wird $0 < -1$ ausgegeben! Der Grund besteht darin, dass der Compiler im Fall verschiedener Datentypen eine Typumwandlung des zweiten Bedingungsoperanden vornimmt, um den Vergleich durchführen zu können. Bei der Umwandlung von -1 in eine `unsigned`-Zahl wird das Bitmuster beibehalten. Damit ist das Ergebnis die größtmögliche `unsigned`-Zahl, die natürlich größer als 0 ist.

Bedingungsoperator ?:

Dieser Operator ist der einzige in C++, der drei Operanden benötigt. Abbildung 1.7 zeigt die Syntax.

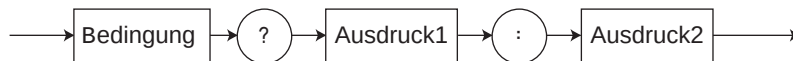


Abbildung 1.7: Syntaxdiagramm des Bedingungsoperators

Falls die *Bedingung* zutrifft, ist der Wert des gesamten Ausdrucks der Wert von *Ausdruck1*, ansonsten der Wert von *Ausdruck2*. Ein Ausdruck mit dem Bedingungsoperator kann lesbarer durch eine `if`-Anweisung ersetzt werden, wird aber wegen seiner Kürze geschätzt. Die Berechnung des Maximums zweier Zahlen lautet:

```
max = a > b ? a : b;
```

Das `if`-Äquivalent dazu ist:

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



Übungen

1.3 Schreiben Sie ein Programm, das drei ganze Zahlen als Eingabe verlangt. Die erste ist als Anfang eines Zahlenbereichs, die zweite als Ende des Bereichs gemeint. Das Programm soll prüfen, ob die dritte Zahl innerhalb des Bereichs einschließlich der Grenzen liegt und eine entsprechende Meldung ausgeben. Geben Sie eine Fehlermeldung aus, wenn die Zahl für den Anfang größer als die Zahl für das Ende ist.

1.4 Schreiben Sie ein Programm, das drei ganze Zahlen als Eingabe verlangt und dann die größte der Zahlen ausgibt.

1.8.4 Fallunterscheidungen mit `switch`

Eine `if`-Anweisung erlaubt nur zwei Möglichkeiten. Erst durch die Verschachtelung konnte die Auswahl unter mehreren Möglichkeiten getroffen werden. Die Gefahr besteht jedoch, dass die gesamte Anweisung bei größerer Schachtelungstiefe unübersichtlich wird und Änderungen nur umständlich nachzutragen sind. Einfacher und übersicht-

licher ist daher die Auswahl unter vielen Anweisungen mit `switch`. Abbildung 1.8 zeigt die Syntax, ein Beispiel folgt unten.

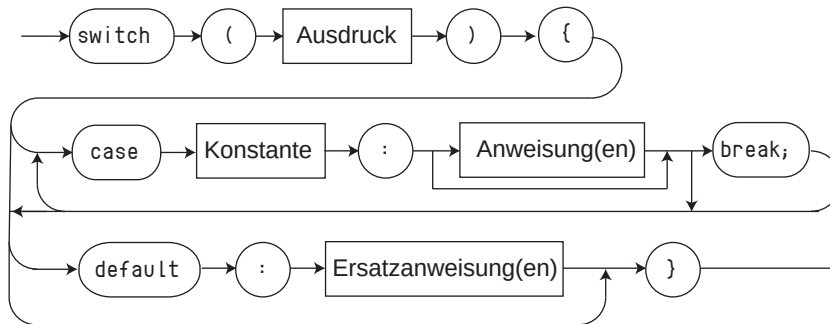


Abbildung 1.8: Syntaxdiagramm einer `switch`-Anweisung

Der *Ausdruck* wird ausgewertet und muss ein Ergebnis vom Typ `int` haben oder nach `int` konvertierbar sein wie zum Beispiel `char`. Dieses Ergebnis wird mit den `case`-Konstanten `const1`, `const2` ... verglichen, die zum Einsprung an die richtige Stelle dienen. Bei Übereinstimmung werden die zur passenden Konstante gehörigen Anweisungen ausgeführt. Nach Ausführung wird nicht automatisch aus der `switch`-Anweisung herausgesprungen. Erst `break` führt zum Verlassen der `switch`-Anweisung und sollte stets verwendet werden, um unnötige Tests auf die Folgewerte zu vermeiden. Die `case`-Konstanten `const1`, `const2` ... müssen eindeutig und auf `int` abbildbar sein. Zeichen (Typ `char`) sind erlaubt, `float`-Zahlen nicht.

Die nach `default` stehenden Anweisungen (meistens nur eine) werden immer dann ausgeführt, wenn der `switch`-Ausdruck einen Wert liefert, der mit keiner der `case`-Konstanten übereinstimmt. `default` ist optional, doch ist es sinnvoll, `default` mit anzugeben. Insbesondere fängt man an dieser Stelle nicht vorgesehene Werte des `switch`-Ausdrucks ab oder Daten, die nicht berücksichtigt werden sollen, wie zum Beispiel fehlerhafte Tastatureingaben. Die Aufgabe, römische Ziffern zu erkennen, kann übersichtlicher mit der Fallunterscheidung durch `switch` als mit verschachtelten `if`-Anweisungen wie auf Seite 65 gelöst werden, wie das folgende Programm zeigt.

Listing 1.9: Umwandlung römischer Ziffern mit `switch`

```

// cppbuch/k1/roemzif2.cpp
#include<iostream>
using namespace std;

int main() {
    int a = -1;
    char c;
    cout << "Zeichen ?";
    cin >> c;
    switch(c) {
        case 'I' : a = 1; break;
        case 'V' : a = 5; break;
    }
}

```

```

    case 'X' : a = 10; break;
    case 'L' : a = 50; break;
    case 'C' : a = 100; break;
    case 'D' : a = 500; break;
    case 'M' : a = 1000; break;
    default : a = 0;
}
if (a > 0) {
    cout << "a = " << a << endl;
}
else {
    cout << "keine römische Ziffer!" << endl;
}
}

```

Wenn eine case-Konstante mit der switch-Variablen übereinstimmt, werden *alle nachfolgenden Anweisungen bis zum ersten break* ausgeführt. Mit einem fehlenden break und einer fehlenden Anweisung nach einer case-Konstante lässt sich eine ODER-Verknüpfung realisieren. Bei der Umwandlung römischer Ziffern lässt sich damit die Auswertung von Kleinbuchstaben bewerkstelligen:

```

switch(c) {
    case 'i' :
    case 'I' : a = 1; break;
    case 'v' : case 'V' : a = 5; break; // andere Schreibweise
    // Rest weggelassen
}

```

Ein fehlendes break sollte kommentiert werden, sofern sich die Absicht nicht klar aus dem Programm ergibt. Alle interessierenden case-Konstanten müssen einzeln aufgelistet werden. Es ist in C++ *nicht* möglich, *Bereiche* anzugeben, etwa der Art case 7..11 : Anweisung break; anstelle der länglichen Liste case 7: case 8: case 9: case 10: case 11: Anweisung break;. In solchen Fällen ist es vielleicht günstiger, einige Vergleiche aus der switch-Anweisung herauszunehmen und als Abfrage if(ausdruck >= startwert && ausdruck <= endwert)... zu realisieren.

1.8.5 Wiederholungen

Häufig muss die gleiche Teilaufgabe oft wiederholt werden. Denken Sie nur an die Summation von Tabellenspalten in der Buchführung oder an das Suchen einer bestimmten Textstelle in einem Buch. In C++ gibt es zur Wiederholung von Anweisungen drei verschiedene Arten von Schleifen. In einer Schleife wird nach Abarbeitung einer Teilaufgabe (zum Beispiel Addition einer Zahl) wieder an den Anfang zurückgekehrt, um die gleiche Aufgabe noch einmal auszuführen (Addition der nächsten Zahl). Durch bestimmte Bedingungen gesteuert, zum Beispiel Ende der Tabelle, bricht irgendwann die Schleife ab.

Schleifen mit while

Abbildung 1.9 zeigt die Syntax von while-Schleifen. *AnweisungOderBlock* ist wie auf Seite 63 definiert. Die Bedeutung einer while-Schleife ist: Solange die Bedingung wahr ist, die Auswertung also ein Ergebnis ungleich 0 oder true liefert, wird die Anweisung bzw. der Block ausgeführt. Die Bedingung wird auf jeden Fall zuerst geprüft. Wenn die

Bedingung von vornherein unwahr ist, wird die Anweisung gar nicht erst ausgeführt (siehe Abbildung 1.10). Die Anweisung oder den Block innerhalb der Schleife nennt man *Schleifenkörper*. Schleifen können wie *if*-Anweisungen beliebig geschachtelt werden.

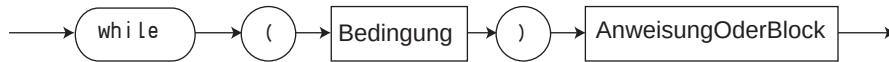


Abbildung 1.9: Syntaxdiagramm einer *while*-Schleife

```

while(Bedingung1) // geschachtelte Schleifen, ohne und mit geschweiften Klammern
while(Bedingung2) {
    ....
    while(Bedingung3) {
        ....
    }
}
  
```

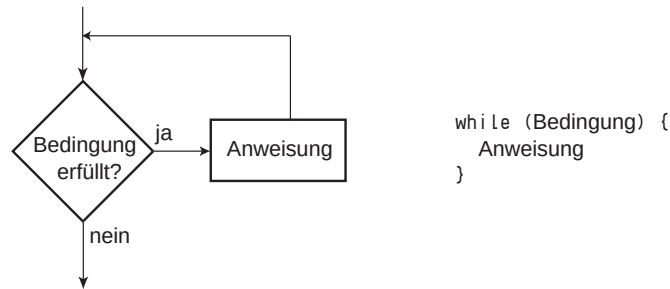


Abbildung 1.10: Flussdiagramm für eine *while*-Anweisung

Beispiele

■ Unendliche Schleife:

```
while(true) Anweisung
```

■ Anweisung wird nie ausgeführt (unerreichbarer Programmcode):

```
while(false) Anweisung
```

■ Summation der Zahlen 1 bis 99:

```

int sum = 0;
int n = 1;
int grenze = 99;
while(n <= grenze) {
    sum += n++;
}
  
```

■ Berechnung des größten gemeinsamen Teilers GGT(x,y) für zwei natürliche Zahlen x und y nach Euklid. Es gilt:

- $\text{GGT}(x, x)$, also $x = y$: Das Resultat ist x .
 - $\text{GGT}(x, y)$ bleibt unverändert, falls die größere der beiden Zahlen durch die Differenz ersetzt wird, also $\text{GGT}(x, y) == \text{GGT}(x, y-x)$, falls $x < y$.
- Das Ersetzen der Differenz geschieht im folgenden Beispiel iterativ, also durch eine Schleife.

Listing 1.10: Beispiel für while-Schleife

```
// cppbuch/k1/ggt.cpp   Berechnung des größten gemeinsamen Teilers
#include<iostream>
using namespace std;

int main() {
    unsigned int x, y;
    cout << "2 Zahlen > 0 eingeben :";
    cin >> x >> y;
    cout << "Der GGT von " << x << " und " << y << " ist ";
    while( x!= y) {
        if(x > y) {
            x -= y;
        }
        else {
            y -= x;
        }
    }
    cout << x << endl;
}
```

Innerhalb einer Schleife muss es eine Veränderung derart geben, dass die Bedingung irgendwann einmal unwahr wird, sodass die Schleife abbricht (man sagt auch *terminiert*). Unbeabsichtigte »unendliche« Schleifen sind ein häufiger Programmierfehler. Im GGT-Beispiel ist leicht erkennbar, dass die Schleife irgendwann beendet sein *muss*:

1. Bei jedem Durchlauf wird mindestens eine der beiden Zahlen kleiner.
2. Die Zahl 0 kann nicht erreicht werden, da immer eine kleinere von einer größeren Zahl subtrahiert wird. Die `while`-Bedingung schließt die Subtraktion gleich großer Zahlen aus, und nur die könnte 0 ergeben.

Daraus allein ergibt sich, dass die Schleife beendet wird, und zwar in weniger als x Schritten, wenn x die anfangs größere Zahl war. Im Allgemeinen sind es erheblich weniger, wie eine genauere Analyse ergibt.



Tipp

Falls der Schleifenkörper aus vielen Anweisungen besteht, sollten die Anweisungen zur Veränderung der Bedingung an den Schluss gestellt werden, um sie leicht finden zu können.

Schleifen mit `do while`

Abbildung 1.11 zeigt die Syntax einer `do while`-Schleife. *AnweisungOderBlock* ist wie auf Seite 63 definiert. Die Anweisung oder der Block einer `do while`-Schleife wird ausgeführt,

und *erst anschließend* wird die Bedingung geprüft. Ist sie wahr, wird die Anweisung ein weiteres Mal ausgeführt usw. Die Anweisung wird also mindestens einmal ausgeführt.



Abbildung 1.11: Syntaxdiagramm einer `do while`-Schleife

Im Flussdiagramm ist die Anweisung ein Block (siehe rechts in der Abbildung 1.12). `do while`-Schleifen eignen sich unter anderem gut zur sicheren Abfrage von Daten, indem die Abfrage so lange wiederholt wird, bis die abgefragten Daten in einem plausiblen Bereich liegen, wie im Primzahlprogramm unten zu sehen ist.

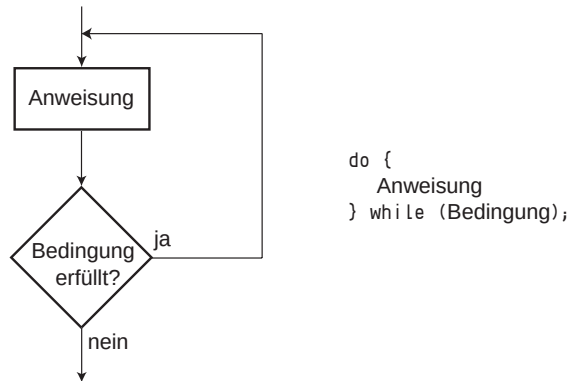


Abbildung 1.12: Flussdiagramm für eine `do while`-Anweisung

Es empfiehlt sich zur besseren Lesbarkeit, `do while`-Schleifen strukturiert zu schreiben. Die schließende geschweifte Klammer soll genau unter dem ersten Zeichen der Zeile stehen, die die öffnende geschweifte Klammer enthält. Dadurch und durch Einrücken des dazwischen stehenden Textes ist sofort der Schleifenkörper erkennbar, auch bei längeren Programmen.

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

Das *direkt hinter* die abschließende geschweifte Klammer geschriebene `while` macht unmittelbar deutlich, dass dieses `while` zu einem `do` gehört. Das ist besonders wichtig, wenn der Schleifenkörper in einer Programmliste über die Seitengrenze ragt. Eine `do while`-Schleife kann stets in eine `while`-Schleife umgeformt werden (und umgekehrt).

Listing 1.11: Beispiel für `do while`-Schleife

```
// cppbuch/k1/primzahl.cpp: Berechnen einer Primzahl, die auf eine gegebene Zahl folgt
#include <iostream>
#include <cmath>
using namespace std;
int main() {
```

```

// Mehrere, durch " getrennte Texte ergeben eine lange Zeile in der Ausgabe.
cout << "Berechnung der ersten Primzahl, die >="
      " der eingegebenen Zahl ist\n";

long z;
// do while-Schleife zur Eingabe und Plausibilitätskontrolle
do {
    // Abfrage, solange z ≤ 3 ist
    cout << "Zahl > 3 eingeben :";
    cin >> z;
} while(z <= 3);
if(z % 2 == 0) { // Falls z gerade ist: nächste (ungerade) Zahl nehmen
    ++z;
}
bool gefunden = false;
do {
    // limit = Grenze, bis zu der gerechnet werden muss.
    // sqrt() arbeitet mit double, daher wird der Typ explizit umgewandelt.
    long limit = 1 + static_cast<long>( sqrt(static_cast<double>(z)));
    long rest;
    long teiler = 1;
    do { // Kandidat z durch alle ungeraden Teiler dividieren
        teiler += 2;
        rest = z % teiler;
    } while(rest > 0 && teiler < limit);
    if(rest > 0 && teiler >= limit)
        gefunden = true;
    else // sonst nächste ungerade Zahl untersuchen:
        z += 2;
} while(!gefunden);
cout << "Die nächste Primzahl ist " << z << endl;
}

```

Schleifen mit for

Die letzte Art von Schleifen ist die `for`-Schleife. Sie wird häufig eingesetzt, wenn die Anzahl der Wiederholungen vorher feststeht, aber das muss durchaus nicht so sein. Abbildung 1.13 zeigt die Syntax einer `for`-Schleife.

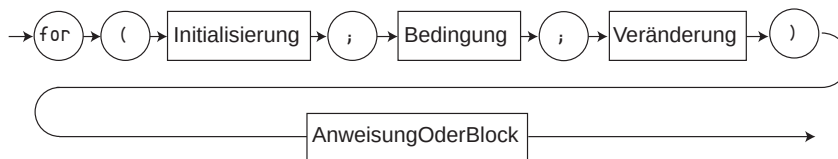


Abbildung 1.13: Syntaxdiagramm einer `for`-Schleife

Der zu wiederholende Teil (Anweisung oder Block) wird auch Schleifenkörper genannt. Beispiel: ASCII-Tabelle im Bereich 65 ... 69 ausgeben

```

for(int i = 65; i <= 69; ++i) {
    cout << i << " " << static_cast<char>(i) << endl;
}

```

Bei der Abarbeitung werden die folgenden Schritte durchlaufen:

1. Durchführung der Initialisierung, zum Beispiel Startwert für eine Laufvariable festlegen. Eine Laufvariable wird wie `i` in der Beispielschleife als Zähler benutzt.
2. Prüfen der Bedingung.
3. Falls die Bedingung wahr ist, zuerst die Anweisung und dann die Veränderung ausführen.

Die Laufvariable `i` kann auch außerhalb der runden Klammern deklariert werden, dies gilt aber als schlechter Stil. Der Unterschied besteht darin, dass außerhalb der Klammern deklarierte Laufvariablen noch über die Schleife hinaus gültig sind.

```
int i;                                // nicht empfohlen
for(i = 0; i < 100; ++i) {
    // Programmcode, i ist hier bekannt
}
// i ist weiterhin bekannt ...
```

Im Fall der Deklaration innerhalb der runden Klammern bleibt die Gültigkeit auf den Schleifenkörper beschränkt:

```
for(int i = 0; i < 100; ++i) {        // empfohlen
    // Programmcode, i ist hier bekannt
}
// i ist hier nicht mehr bekannt
```

Die zweite, unten im Beispielprogramm verwendete Art erlaubt es, `for`-Schleifen als selbstständige Programmteile hinzuzufügen oder zu entfernen, ohne Deklarationen in anderen Schleifen ändern zu müssen. Derselbe Mechanismus gilt für Deklarationen in den runden Klammern von `if`-, `while`- und `switch`-Anweisungen.

Listing 1.12: Beispiel für `for`-Schleife

```
// cppbuch/k1/fakultaet.cpp
#include<iostream>
using namespace std;

int main() {
    cout << "Fakultät berechnen. Zahl >= 0? :";
    int n;
    cin >> n;
    unsigned long fak = 1;
    for(int i = 2; i <= n; ++i) {
        fak *= i;
    }
    cout << n << "! = " << fak << endl;
}
```

Um Fehler zu vermeiden und zur besseren Verständlichkeit sollte man niemals Laufvariablen in der Anweisung verändern. Das Auffinden von Fehlern würde durch die Änderung erschwert.

```
for(int i = 65; i < 70; ++i) {
    // eine Seite Programmcode
    --i; // irgendwo dazwischen erzeugt eine unendliche Schleife
```

```

    // noch mehr Programmcode
}

```

Auch wenn der Schleifenkörper nur aus einer Anweisung besteht, wird empfohlen, sie in Klammern { } einzuschließen.

Äquivalenz von for und while

Eine for-Schleife entspricht direkt einer while-Schleife, sie ist im Grunde nur eine Umformulierung, solange nicht `continue` vorkommt (das im folgenden Abschnitt beschrieben wird):

```

for(Initialisierung; Bedingung; Veraenderung)
    Anweisung

```

ist äquivalent zu:

```

{
    Initialisierung;
    while(Bedingung) {
        Anweisung
        Veraenderung;
    }
}

```

Die äußeren Klammern sorgen dafür, dass in der Initialisierung deklarierte Variablen wie bei der for-Schleife nach dem Ende nicht mehr gültig sind. `Anweisung` kann wie immer auch eine Verbundanweisung (Block) sein, in der mehrere Anweisungen stehen können, durch geschweifte Klammern begrenzt. Die umformulierte Entsprechung des obigen Beispiels (ASCII-Tabelle von 65 ... 69 ausgeben) lautet:

```

{
    int i = 65;                // Initialisierung
    while(i < 70) {            // Bedingung
        cout << i << " " << static_cast<char>(i) << endl; // Anweisung
        ++i;
    }                          // Veränderung
}

```

float- oder double-Laufvariablen vermeiden!

Wegen der Rechenungenauigkeit kann es bei nicht-integralen Typen wie `double` oder `float` zu nicht vorhersagbarem Verhalten kommen. Das Beispiel:

```

for(double d = 0.4; d <= 1.2; d += 0.4) {
    cout << d << endl;
}

```

lässt auf den ersten Blick die Ausgabe 0.4, 0.8, 1.2 erwarten, tatsächlich werden auf meinem System nur die zwei Zahlen 0.4 und 0.8 ausgegeben. Wenn ich jedoch

```

for(double d = 0.5; d <= 1.5; d += 0.5) {
    cout << d << endl;
}

```


ausführe, werden wie erwartet die drei Zahlen 0.5, 1 und 1.5 angezeigt. Der Grund liegt darin, dass 0.5 intern exakt darstellbar ist, 0.4 jedoch nicht. Schon ein Unterschied im letzten Bit lässt den Vergleich auf Gleichheit scheitern. Ganz ungünstig kann sich die Prüfung auf Ungleichheit mit `!=` auswirken:

```
for(float f = 0.4; f != 10.4; ++f) { // ∞-Schleife
    cout << f << endl;
}
```

Abgesehen von möglichen arithmetischen Problemen wird in der letzten Schleife die `float`-Variable `f` mit der `double`-Konstante 10.4 verglichen. Um den Vergleich durchführen zu können, bringt der Compiler beide Größen auf dieselbe Bitbreite, das heißt, er interpretiert den Vergleich als `(double)f != 10.4`. Die im Vergleich zu `double` wenigen Bits der `float`-Zahl reichen nicht für die erforderliche Genauigkeit aus, wie die folgenden Zeilen zeigen:

```
// Formatierung siehe Kapitel 10
cout.setf(ios::showpoint|ios::fixed, ios::floatfield);
cout.precision(15); // angezeigte Genauigkeit
                        // Ausgabe auf meinem System:
cout << 10.4 << endl;      // 10.400000000000000
cout << (double)10.4f << endl; // 10.3999999618530273
```

Wenn man bei `double`-Laufvariablen nur die Operatoren `<` oder `>` zum Vergleich verwendet, ist das Problem zum Teil entschärft. Aber die Anzahl der Schleifendurchläufe bleibt möglicherweise undefiniert: Nur eine sehr geringe Abweichung im Wert der Variablen oder der begrenzenden Konstanten entscheidet, ob der Schleifenkörper einmal mehr ausgeführt wird oder nicht. Also: Verwenden Sie nur integrale Datentypen wie `int`, `size_t` oder `char` als Laufvariable in Schleifen!

Kommaoperator

Der Kommaoperator wird gelegentlich in den Bestandteilen Initialisierung, Bedingung, Verändern einer `for`-Schleife benutzt, um die Schleife kompakter zu schreiben, meistens mit dem Ergebnis einer schlechteren Lesbarkeit. Er gibt eine Reihenfolge von links nach rechts vor. Das folgende Programmstück summiert die Zahlen 1 bis 100:

```
int sum = 0;
for(int i = 1; i <= 100; ++i) {
    sum += i;
}
```

Mit Hilfe des Kommaoperators wird der Schleifenkörper in den Veränderungsteil verlegt. Sowohl `i` als auch `sum` bekommen im Initialisierungsteil ihre Anfangswerte zugewiesen:

```
int sum;
for(int i = 1, sum = 0; i <= 100; sum += i, ++i);
```

Theoretisch könnte auch die Deklaration von `sum` in den Initialisierungsteil verlegt werden, nur kommt man dann außerhalb der Schleife nicht mehr an den Wert der Variablen. Der Kommaoperator hat die niedrigste Priorität von allen Operatoren (siehe Tabelle A.4 auf Seite 890).

1.8.6 Kontrolle mit `break` und `continue`

`break` und `continue` sind Sprunganweisungen. Bisher wurde `break` in der `switch`-Anweisung verwendet, um sie zu verlassen. `break` wirkt genauso in einer Schleife, das heißt, dass die Schleife beim Erreichen von `break` beendet wird. `continue` hingegen überspringt den Rest des Schleifenkörpers. In einer `while`- oder `do while`-Schleife würde als Nächstes die Bedingung geprüft werden, um davon abhängig die Schleife am Beginn des Schleifenrumpfs fortzusetzen. Das folgende kleine Menü-Programm zeigt `break` und `continue`.

Listing 1.13: Menü mit `break` und `continue`

```
// cppbuch/k1/menu.cpp
#include<iostream>
using namespace std;

int main() {
    char c;
    while(true) {                // unendliche Schleife
        cout << "Wählen Sie: a, b, x = Ende : ";
        cin >> c;
        if(c == 'a') {
            cout << "Programm a\n";
            continue;            // zurück zur Auswahl
        }
        if(c == 'b') {
            cout << "Programm b\n";
            continue;            // zurück zur Auswahl
        }
        if(c == 'x') {
            break;                // Schleife verlassen
        }
        cout << "Falsche Eingabe! Bitte wiederholen!\n";
    }
    cout << "\n Programmende mit break\n";
}
```

In einer `for`-Schleife würde als Nächstes die Veränderung ausgeführt und dann erst die Bedingung erneut geprüft. Insofern stimmt die oben erwähnte Äquivalenz der `for`-Schleife mit einer `while`-Schleife exakt nur für `for`-Schleifen ohne `continue`. Ohne `break` und `continue` wären gegebenenfalls viele `if`-Abfragen notwendig, die die Lesbarkeit eines Programms verschlechtern.

In einem größeren Programm können viele verteilte `break`- oder `continue`-Anweisungen die Verständlichkeit beeinträchtigen. Deshalb gibt es die Meinung, dass jeder Block nur einen einzigen Einstiegs- und einen einzigen Ausstiegspunkt haben soll (englisch *single entry/ single exit*). Um dies zu erreichen, kann man eine Hilfsvariable einführen, die den Abbruch signalisiert (siehe Übungsaufgabe 1.9).

Eine Alternative besteht darin, `break`-Anweisungen mit einem deutlichen Kommentar wie zum Beispiel `// EXIT!` am rechten Rand zu kennzeichnen. Wenn eine Schleife mit `break` nur wenige Zeilen umfasst, trägt eine Hilfsvariable nicht zur Übersichtlichkeit bei.



Übungen

1.5 Schreiben Sie eine Schleife, die eine gegebene Zahl binär ausgibt, indem Sie mit geeigneten Bit-Operationen prüfen, welche Bits der Zahl gesetzt sind. Tipp: Verwenden Sie die Zahl 1, verschoben um 0 bis z.B. 32 Bit-Positionen, als »Maske«. Beispielergebnisse:

5 → 00000000000000000000000000000101

-5 → 11111111111111111111111111111011

Es ist zu sehen, dass -5 intern als Zweierkomplement von 5 dargestellt wird.

1.6 Welche Fallstricke sind in den folgenden Schleifen verborgen? Dabei soll auch darauf geachtet werden, unter welchen Umständen die Schleifen terminieren (sich beenden).

a) `while(i > 0)`

`k = 2 * k;`

b) `while(i != 0)`

`i = i - 2;`

c) `while(n != i) {`

`++i;`

`n = 2 * i;`

`}`

1.7 Fünf Personen haben versucht, die Summe der Zahlen von 1 bis 100 zu berechnen. Beurteilen Sie die folgenden Lösungsvorschläge:

(a) `int n = 1, sum = 0;`

`while(n <= 100) {`

`++n;`

`sum += n;`

`}`

(d) `int n = 1;`

`while(n < 100) {`

`int sum = 0;`

`n = n + 1;`

`sum = sum + n;`

`}`

(b) `int n = 1, sum = 1;`

`while(n < 100)`

`n += 1;`

`sum += n;`

(e) `int n = 1, sum = 0;`

`while(n <= 0100) {`

`sum += n;`

`++n;`

`}`

(c) `int n = 100;`

`int sum = n*(n+1)/2;`

1.8 Berechnen Sie die Summe aller natürlichen Zahlen von n_1 bis n_2 mit

a) einer `for`-Schleife,

b) einer `while`-Schleife,

c) einer `do while`-Schleife,

d) ohne Schleife.

Es sei $n_2 \geq n_1$ vorausgesetzt. Tipp: Erst die vorstehende Aufgabe lösen.

1.9 Formulieren Sie das Programm in Abschnitt 1.8.6 um, sodass ein funktionsäquivalentes Programm ohne `continue` in der Schleife entsteht. Anstelle der `if`-Anweisungen soll eine `switch`-Anweisung eingesetzt werden, um das Programm zu verkürzen.

1.9 Benutzerdefinierte und zusammengesetzte Datentypen

Abgesehen von den Grunddatentypen gibt es Datentypen, die aus den Grunddatentypen zusammengesetzt sind und die Sie selbst definieren können (benutzerdefinierte Datentypen).

1.9.1 Aufzählungstypen

Häufig gibt es nicht-numerische Wertebereiche. So kann zum Beispiel ein Wochentag nur die Werte Sonntag, Montag, Dienstag, Mittwoch, Donnerstag, Freitag und Samstag annehmen. Oder ein Farbwert in unserem C++-Programm soll nur den vier Farben Rot, Grün, Blau, und Gelb entsprechen. Eine mögliche Hilfskonstruktion wäre die Abbildung auf den Datentyp `int`:

```
int eineFarbe; // rot = 0
                // grün = 1
                // blau = 2
                // gelb = 3

int einWochentag; // Sonntag = 0
                  // Montag = 1 usw.
```

Dieses Verfahren hätte einige Nachteile:

- Die Bedeutung muss im Programm als Kommentar festgehalten werden.
- Zugeordnete Zahlen sind nicht eindeutig: 0 kann rot, aber auch Sonntag bedeuten, und 1 kann für grün oder auch Montag stehen.
- Schlechter Dokumentationswert, zum Beispiel `if(eineFarbe == 2) ...`. Hieraus ist nicht zu ersehen, welche Farbe gemeint ist. Oder : `if(eineFarbe == 5) ...`. Der Wert 5 ist undefiniert!

Die Lösung für solche Fälle sind die *Aufzählungs- oder Enumerationstypen*, die eine Erweiterung der vordefinierten Typen durch den Programmierer darstellen. Abbildung 1.14 zeigt das Syntaxdiagramm einer `enum`-Deklaration.



Abbildung 1.14: Syntaxdiagramm einer `enum`-Deklaration

Typname oder Variablenliste können weggelassen werden (in diesem Fall aber nicht beide). Sinnvoll ist meistens nur das Weglassen der Variablenliste. Der neue Datentyp Farbtyp wird deklariert:

```
enum Farbtyp {rot, gruen, blau, gelb};
```

Der neue Datentyp Wochentag wird deklariert:

```
enum Wochentag {sonntag, montag, dienstag, mittwoch,
               donnerstag, freitag, samstag};
```

Wenn der Datentyp erst einmal bekannt ist, können weitere Variablen definiert werden:

```
Farbtyp gewaehlteFarbe;           // Definition
Wochentag derFeiertag, einWerktag, // Definitionen
        heute = dienstag; // Definition + Initialisierung
```

Falls ein Aufzählungstyp nur ein einziges Mal in einer Variablendefinition benötigt wird, kann der Typname weggelassen werden. Man erhält dann eine *anonyme Typdefinition*:

```
enum {fahrrad, mofa, lkw, pkw} einFahrzeug;
```

Den mithilfe von Aufzählungstypen definierten Variablen können ausschließlich Werte aus der zugehörigen Liste zugewiesen werden, Mischungen sind nicht erlaubt. Aufzählungstypen sind eigene Datentypen, werden aber intern auf die natürlichen Zahlen abgebildet, beginnend bei 0. Eine Voreinstellung mit anderen Zahlen ist möglich, wird aber nur gelegentlich erforderlich sein, vielleicht bei einer gewünschten binären Codierung, zum Beispiel

```
// Abweichung von der Standardeinstellung 0, 1, 2, 3 ...:
// Deklaration:
enum Farbtyp {rot = 0, gruen = 1, blau = 2, gelb = 4};

// Deklaration mit Variablendefinition
enum Palette {weiss = 0, grau = 1, braun = 2, amber = 4, lila = 8
} Mischung;

// Deklaration mit Bitshift-Operator
enum Bitmaske { wert1 = 1 << 0, wert2 = 1 << 1, wert3 = 1 << 2,
               wert4 = 1 << 3, wert5 = 1 << 4, wert6 = 1 << 5,
               // ... usw.
};
```

Die in Farbtyp definierten Farben rot, gruen, blau, gelb dürfen in Palette nicht mehr verwendet werden, sofern sich Palette im gleichen Gültigkeitsbereich befindet. Eine Umwandlung in int ist erlaubt, aber nicht die Umwandlung einer int-Zahl in einen enum-Typ. Als Operation auf enum-Typen ist nur die Zuweisung erlaubt, bei allen anderen Operatoren wird vorher in int gewandelt. Welche Anweisungen möglich oder falsch sind, zeigen die nächsten Zeilen, wobei die Variablenamen sich auf die obigen Definitionen beziehen:

```
int i = dienstag;           // richtig (implizite Umwandlung nach int)
heute = montag;             // richtig
heute = i;                  // Fehler, Datentyp inkompatibel
montag = heute;             // Fehler (montag ist Konstante)
i = rot + blau;             // möglich (implizite Umwandlung nach int)
Mischung = weiss + lila;    // Fehler, Rückwandlung des int-Ergebnisses
                             // in Typ Palette ist nicht möglich
++Mischung;                 // Fehler, gleicher Grund
if(Mischung > grau)         // implizite Typumwandlung
    Mischung = lila;        // richtig
```

Aufgrund der Umwandlung in int bei arithmetischen Operationen sind Operationen wie Mischung++ fehlerhaft, da sich ein undefinierter Wert ergeben kann.

1.9.2 Arrays: Der C++-Standardtyp `vector`

Im täglichen Leben benutzen wir häufig Tabellen, auch Arrays genannt. Einspaltige Tabellen, und um die geht es hier zunächst, werden in C++ durch eine *Vektor* genannte Konstruktion abgebildet. Aus der Sprache C sind primitive Arrays bekannt. Diese sind zwar auch Bestandteil von C++, sind aber nicht besonders komfortabel und werden daher erst in Abschnitt 5.2 behandelt.

Ein *Vektor* ist eine Tabelle von Elementen desselben Datentyps, also eine Tabelle zum Beispiel nur mit ganzen Zahlen oder nur mit `double`-Zahlen. Mit Ausnahme von Referenzen kann der Datentyp beliebig sein, insbesondere kann er selbst wieder zusammengesetzt sein. Auf ein Element der Tabelle wird über die *Positionsangabe* zugegriffen, also über die Nummer der Reihe, in der sich das Element befindet.

In C++ ist ein Vektor eine vordefinierte Klasse, um deren internen Aufbau wir uns erst später kümmern. Zunächst geht es nur um die Benutzung als Baustein in eigenen Programmen. Es kommen dabei zwei Sichtweisen zum Ausdruck:

1. »Bausteine« werden benutzt, um Programme oder Programmteile zu entwickeln, die selbst wieder Bausteine sein können. Dazu ist die Kenntnis notwendig, was ein Baustein leistet und wie er verwendet werden muss, aber nicht wie er intern funktioniert.
2. »Bausteine« sollen von Grund auf entwickelt oder weiterentwickelt werden. Dann ist eine gründliche Kenntnis der inneren Funktion unerlässlich.

Für Softwareentwickler sind beide Sichten wichtig. Hier wird der Standardtyp `vector` zunächst nur benutzt, und erst viel weiter unten wird erklärt, was im Innern vorgeht. Die Klasse ist eine Beschreibung für Objekte, wie aus Abschnitt 1.2 der Einführung bekannt ist. Die Anweisung

```
vector<int> v(10);
```

stellt einen Vektor `v` bereit, der 10 Elemente des Typs `int` aufnehmen kann. Die Feldelemente sind stets von 0 bis (Anzahl der Elemente - 1) durchnummeriert, hier also von 0 bis 9. Die Klasse für Vektoren stellt einige Dienstleistungen zur Verfügung, die mit der in der Einführung beschriebenen Notation abgerufen werden können. Eine dieser Dienstleistungen ist die Ermittlung der Zahl der Elemente, das heißt die Größe (englisch *size*) des Vektors:

```
cout << v.size() << endl;           // 10
```

Daten müssen in diesem Fall nicht zwischen den runden Klammern übergeben werden, das Vektor-Objekt enthält alle nötigen Informationen. Der Zugriff auf ein spezielles Element wird mit dem Indexoperator `[]` realisiert. Zum Beispiel zeigt

```
cout << v[0] << endl;               // Zählung beginnt bei 0
```

das erste Element des Vektors an. Der zwischen den eckigen Klammern stehende Wert heißt *Index*. Zugriffe auf Vektor-Elemente kommen typischerweise in Schleifen vor, weil meistens eine Operation für die gesamte Tabelle ausgeführt werden soll. Dabei ist sorgfältig darauf zu achten, dass die Indexwerte die Vektorgrenzen nicht unter- oder überschreiten.

Vorsicht Falle!

Es gibt keine Überprüfung der Bereichsüber- oder -unterschreitung! Zugriffe auf nicht existierende Elemente wie `c = v[100]` erzeugen *keine* Fehlermeldung, weder durch den Compiler noch zur Laufzeit, sofern nicht der zulässige Speicherbereich für das Programm überschritten wird! Der Grund: Schnelligkeit ist ein Designprinzip von C++, und die Überprüfung eines jeden Zugriffs kostet eben Zeit.

Umgehen der Falle

Den Compiler so einzustellen, dass der Index bei jedem Zugriff über die eckigen Klammern geprüft wird, geht leider nicht. Es gibt in C++ jedoch andere Möglichkeiten dafür (Abschnitt 9.2), die natürlich ein Programm verlangsamen (meist nur geringfügig). Man kann alternativ auf die eckigen Klammern verzichten und einen Vektor nach einem Wert *an* (englisch *at*) einer Position fragen. Diese Art des Zugriff wird geprüft:

```
cout << v.at(0) << endl; // alles bestens, dasselbe wie v[0]
// 1000 ist zuviel!
cout << v.at(1000) << endl; // Programmabbruch mit Fehlermeldung
```

Im Beispiel unten kann man sich dadurch helfen, dass der Index niemals einen falschen Wert haben kann – dies ist sowieso ein besseres Verfahren, als das Programm zu korrigieren, nachdem das Kind in den Brunnen gefallen ist. Der laufende Index wird einfach mit `v.size()` verglichen. `v.size()` gibt die Anzahl der Elemente als nicht-vorzeichenbehafteten Wert zurück. Natürlich muss man ganz sicher sein, in der `for`-Schleife `i < v.size()` abzufragen anstatt `i <= v.size()`... Wer sich leicht vertippt, sollte also doch lieber `v.at(i)` statt `v[i]` schreiben.

Ein Beispiel

Das Programm unten verdeutlicht die Arbeitsweise mit Vektoren. Es werden einige typische Operationen demonstriert, die sich weitgehend selbst erklären. Das verwendete Sortiervorgehen ist eine Variante des Bubble-Sorts und für große Tabellen zu langsam. Große Tabellen sollten mit schnellen Verfahren sortiert werden (siehe Seite 135 und die nach dem Programm angegebene Alternative).

Listing 1.14: Standardklasse `vector`

```
// cppbuch/k1/vektor.cpp
#include<iostream>
#include<vector> // Standard-Vektor bekannt machen
#include<cstdint> // size_t
using namespace std;

int main() { // Programm mit typischen Vektor-Operationen
    vector<float> kosten(12); // Tabelle mit 12 float-Werten
    // Füllen der Tabelle mit beliebigen Daten, dabei Typumwandlung int → float
    for(size_t i = 0; i < kosten.size(); ++i) {
        kosten[i] = static_cast<float>(150-i*i)/10.0;
    }
    // Als Typ der Laufvariablen i in der Schleife wird der auf Seite 47 beschriebene Typ
    // size_t statt int gewählt, weil die Anzahl der Elemente ja ≥ 0 sein muss. Tabelle ausgeben:
```

```

for(size_t i = 0; i < kosten.size(); ++i) {
    cout << i << ": " << kosten[i] << endl;
}
// Berechnung und Anzeige von Summe und Mittelwert
float sum = 0.0;
for(size_t i = 0; i < kosten.size(); ++i) {
    sum += kosten[i];
}
cout << "Summe = " << sum << endl;
cout << "Mittelwert = "
    << sum/kosten.size() // implizite Typumwandlung des Nenners nach float
    << endl;
// Maximum anzeigen
float maxi = kosten[0];
for(size_t i = 1; i < kosten.size(); ++i) {
    if(maxi < kosten[i])
        maxi = kosten[i];
}
cout << "Maximum = " << maxi << endl;

// zweite Tabelle sortierteKosten deklarieren und mit der ersten initialisieren
vector<float> sortierteKosten = kosten;
// zweite Tabelle aufsteigend sortieren (Bubble-Sort-Variante)
// (schnellere Möglichkeit siehe Text unten)
for(size_t i = 1; i < sortierteKosten.size(); ++i) {
    for(size_t j = 0; j < i; ++j) {
        if(sortierteKosten[i] < sortierteKosten[j]) {
            // Elemente an i und j vertauschen
            float temp = sortierteKosten[i];
            sortierteKosten[i] = sortierteKosten[j];
            sortierteKosten[j] = temp;
        }
    }
}
// Tabelle ausgeben
for(size_t i = 0; i < sortierteKosten.size(); ++i) {
    cout << i << ": " << sortierteKosten[i] << endl;
}
}

```

Die Standardbibliothek bietet eine schnellere und kürzere Alternative: Dazu muss erst `#include<algorithm>` am Programmanfang eingefügt werden. Dann wird der Bubble-Sort durch den Aufruf

```
sort(sortierteKosten.begin(), sortierteKosten.end());
```

ersetzt. Die Grundlagen dafür finden sich in den Abschnitten [11.2](#) und [24.4.2](#). An der Stelle

```
vector<float> sortierteKosten = kosten; // Initialisierung
```

wäre auch Folgendes möglich gewesen:

```
vector<float> sortierteKosten; // Objekt anlegen
sortierteKosten = kosten; // Zuweisung
```


In C++ ist eine Initialisierung *keine* Zuweisung. Initialisierung und Zuweisung werden in C++ unterschieden. Beides ist trotz desselben Operators (=) leicht auseinander zu halten:



Merke:

Eine Initialisierung kann nur bei der gleichzeitigen Definition (= Erzeugung) eines Objekts auftreten, eine Zuweisung setzt immer ein schon vorhandenes Objekt voraus.

Wenn man die Wahl hat so wie hier, sollte stets der ersten Variante der Vorzug gegeben werden, weil das Initialisieren während der Objekterzeugung schneller vonstatten geht, als erst das Objekt zu erzeugen und dann im zweiten Schritt die Zuweisung der Werte vorzunehmen. Im Programm oben wurde der Vektor in einer Schleife mit Werten versehen. Es ist aber auch eine direkte Initialisierung möglich, etwa

```
vector<double> einVektor = {1.1, 2.2, 3.3, 4.4, 5.5}; // direkte Initialisierung
```

Lineare Suche in Tabellen

Hier seien vier programmiertechnische Möglichkeiten gezeigt, in einer *unsortierten* Tabelle *a* mit *N* Elementen auf den Positionen $0 \dots N - 1$ ein bestimmtes Element *key* zu suchen. Die C++-Standardbibliothek bietet die `find()`-Funktion, aber hier sollen programmiertechnisch verschiedene Schleifenvarianten verglichen werden. Die Variable *i* gibt anschließend die Position an, an der das gesuchte Element *key* erstmalig auftritt. Falls *key* nicht in *a* enthalten ist, muss *i* einen Wert außerhalb $0 \dots N - 1$ annehmen. Die folgenden Algorithmen enden bei erfolgloser Suche mit $i = N$. Die letzte Variante erlaubt eine kürzere Formulierung der Schleife, setzt aber voraus, dass das Feld um einen Eintrag erweitert wird, der als »Wächter« (englisch *sentinel*) für den Abbruch der Schleife dient.

```
// Definitionen für alle vier Fälle
const int N = ...
vector<int> a(N+1); // letztes Element nur für Fall 4
int key = ...      // gesuchtes Element
int i;             // Laufvariable
                  // Ergebnis: i = 0..N - 1 : gefunden, i = N : nicht gefunden!
```

1. while-Schleife

In der Bedingung wird abgefragt, ob das aktuelle Element ungleich *key* und ob die Zählvariable noch im gültigen Bereich ist. So lange wird die Zählvariable inkrementiert.

```
i = 0;
while(i < N && a[i] != key) {
    ++i;
}
```

2. do while-Schleife

Wie oben, nur dass die Zählvariable *vorher* inkrementiert und daher anders vorbestimmt wird. Die vorhergehende Lösung sollte im Vergleich bevorzugt werden, weil es generell besser ist, eine Bedingung zu prüfen und dann zu handeln als umgekehrt.

```
i = -1;
do {
```

```
++i;
} while(i < N && a[i] != key);
```

3. for-Schleife

Die Schleife wird mit `break` verlassen, wenn das Element gefunden wird. Es wäre auch möglich gewesen, die Bedingung `i < N` zu erweitern.

```
for(i = 0; i < N; ++i) {
    if(a[i] == key) {
        break;
    }
}
```

4. (N+1). Element als »Wächter« (sentinel)

In das zusätzliche Element `a[N]` wird `key` eingetragen. Die Schleife muss spätestens hier abbrechen, auch wenn `key` vorher nicht gefunden wurde. Die Zählvariable wird als Seiteneffekt beim Zugriff auf ein Vektorelement hochgezählt.

```
i = -1;
a[N] = key;           // garantiert Abbruch der Schleife
while(a[++i] != key);
```

Vektoren sind dynamisch!

Oft weiß man nicht, wie groß ein Vektor sein soll, zum Beispiel beim Einlesen von Daten per Dialog oder aus einer Datei unbekannter Größe. Ein Vektor der C++-Standardbibliothek hat den Vorteil, dass er bei Bedarf Elemente hinten anhängt und dabei seine Größe ändert. Falls »hinten« kein Platz mehr im Computerspeicher sein sollte, wird der gesamte Vektor an eine neue, ausreichend große Stelle im Speicher verlagert. Dies geschieht ohne Zutun des Programmierers. Mit `push_back` wird er dazu aufgefordert, wobei ihm der anzuhängende Wert in runden Klammern übergeben wird (siehe Beispielprogramm). Eine tabellarische Übersicht der Möglichkeiten von Objekten der Klasse `vector` ist in Abschnitt 28.2.1 zu finden. Ein Großteil der Möglichkeiten ist aber erst nach Kenntnis der folgenden Kapitel bis einschließlich Kapitel 9 verständlich.

Listing 1.15: Vektor dynamisch vergrößern

```
// cppbuch/k1/dynvekt.cpp
#include<iostream>
#include<vector>    // Standard-Vektor
#include<cstdint>   // size_t
using namespace std;

int main() {
    vector<int> meineDaten; // anfängliche Größe ist 0
    int wert;
    do {
        cout << "Wert eingeben (0 = Ende der Eingabe):";
        cin >> wert;
        if(wert != 0) {
            meineDaten.push_back(wert); // Wert anhängen
        }
    }
```

```

    } while(wert != 0);
    cout << "Es wurden die folgenden Werte eingegeben:\n";
    for(size_t i = 0; i < meineDaten.size(); ++i) {
        cout << i << ". Wert : "
              << meineDaten[i] << endl;
    }
}

```

1.9.3 Zeichenketten: Der C++-Standardtyp string

Eine Zeichenkette, auch String genannt, ist aus Zeichen des Typs `char` zusammengesetzt. Im Grunde kann eine Zeichenkette wie eine horizontale Tabelle mit nur einer Reihe aufgefasst werden. Dennoch wird nicht ein `vector<char>`, sondern eine andere Standardklasse mit dem Namen `string` als Baustein verwendet, ohne dass wir uns um ihre Innereien kümmern – jedenfalls jetzt noch nicht. Die Klasse hat gegenüber dem uns bekannten Vektor einige zusätzliche Eigenschaften, von denen eine Auswahl im folgenden Programm beispielhaft gezeigt werden soll.

Eine tabellarische Übersicht der Möglichkeiten von Strings ist in Kapitel 32 zu finden. Wie bei der Standard-Vektorklasse ist ein Großteil der Möglichkeiten erst nach Kenntnis der folgenden Kapitel bis einschließlich Kapitel 9 verständlich. Wie die Klasse `string` im Innern aufgebaut ist, wird an anderer Stelle erläutert (Abschnitt 6.1).

Listing 1.16: Standardklasse `string`

```

// cppbuch/k1/zbststring.cpp
#include<iostream>
#include<string> // Standard-String einschließen
#include<cstdint> // size_t
using namespace std;

int main() { // Programm mit typischen String-Operationen
    // String-Objekt einString anlegen und mit "hallo" initialisieren.
    // einString kann ein beliebiger Name sein.
    string einString("hallo");
    cout << einString << endl; // String ausgeben

    // String zeichenweise ausgeben, ungeprüfter Zugriff wie bei vector:
    for(size_t i = 0; i < einString.size(); ++i) {
        cout << einString[i];
    }
    cout << endl;
    // String zeichenweise mit Indexprüfung ausgeben. Die Prüfung geschieht wie beim Vektor.
    // Ein Versuch, einString.at(i) mit einem i ≥ einString.size() abzufragen,
    // führt zum Programmabbruch mit Fehlermeldung. Die Anzahl der Zeichen kann bei Strings
    // auch mit length() ermittelt werden.
    for(size_t i = 0; i < einString.length(); ++i) {
        cout << einString.at(i);
    }
    cout << endl;
    string eineStringKopie(einString); // Kopie des Strings erzeugen
    cout << eineStringKopie << endl; // hallo
}

```

```

string diesIstNeu("neu!");
eineStringKopie = diesIstNeu;    // Kopie durch Zuweisung
cout << eineStringKopie << endl; // neu!

eineStringKopie = "Buchstaben"; // Zeichenkette zuweisen
cout << eineStringKopie << endl; // Buchstaben

// Zuweisung nur eines Zeichens vom Typ char
einString = 'X';
cout << einString << endl;      // X

// Strings mit dem +=-Operator verketten
einString += eineStringKopie;
cout << einString << endl;      // XBuchstaben

// Strings mit dem +-Operator verketten
einString = eineStringKopie + " ABC";
cout << einString << endl;      // Buchstaben ABC
einString = "123" + eineStringKopie;
cout << einString << endl;      // 123Buchstaben
einString = "123" + "ABC"; // geht nicht! Erklärung folgt in Kap. 9
} // Ende von main()

```



Übungen

1.10 Gegeben sei eine Zeichenkette des Typs `string`, die eine natürliche Zahl darstellen soll und daher nur aus Ziffern besteht; Beispiel: "17462309".

- Wandeln Sie den String in eine Zahl `z` vom Typ `long` um.
- Berechnen Sie die Quersumme von `z`.

Geben Sie die Zahl und die Quersumme auf dem Bildschirm aus.

1.11 Schreiben Sie ein Programm, das eine einzugebende natürliche Zahl in römischer Darstellung ausgibt. Die römischen Ziffern seien in einem konstanten String `ZEICHENVORRAT = "IVXLCDM"` gegeben. Die syntaktische Regel lautet: Keine Ziffer außer 'M' darf mehr als dreimal hintereinanderstehen. Das heißt, ein vierfaches Vorkommen wird durch Subtraktion vom nächsthöheren passenden Wert ersetzt. Subtraktion geschieht durch Voranstellen des kleineren Werts. So wird 4 nicht zu `IIII`, sondern zu `IV`, und 9 wird nicht zu `VIII`, sondern zu `IX`. (Etwas schwierig und nur für Knobelfreunde!)

1.12 Schreiben Sie ein Programm, das beliebig viele Zahlen im Bereich von -99 bis +100 (einschließlich) von der Standardeingabe liest. Der Zahlenbereich sei in 10 gleich große Intervalle eingeteilt. Sobald eine Zahl außerhalb des Bereichs eingegeben wird, sei die Eingabe beendet. Das Programm soll dann für jedes Intervall ausgeben, wie viele Zahlen eingegeben worden sind. Benutzen Sie für -99, +100 usw. Konstanten (`const`). Zur Speicherung der Intervalle soll ein `vector<int>` verwendet werden.

1.13 Das folgende Problem ist klassisch, und es haben sich schon viele Menschen damit beschäftigt: Wenn Zahlen Achterbahn fahren. Gegeben sei eine natürliche Zahl > 0 .

- Wenn die Zahl gerade ist, teile sie durch 2. Wenn nicht, multipliziere sie mit 3 und addiere 1.

2. Wenn die sich ergebende Zahl größer als 1 ist, wende Schritt 1 auf diese Zahl an. Wenn nicht, ist das Verfahren beendet.

Es zeigt sich, dass die Zahlen erheblich anwachsen können und auch wieder kleiner werden – daher der Name Achterbahn. Schreiben Sie ein Programm, das eine Startzahl als Eingabe erwartet und den obigen Algorithmus durchführt. Lassen Sie sich die erreichte Zahl und das erreichte Maximum anzeigen. Am Ende des Programms soll ausgegeben werden, wieviele Iterationen (Durchläufe der Schleife) bis zum Ende des Programms benötigt werden. Mit den Anweisungen

```
string dummy;
getline(cin, dummy); // weiter mit Tastendruck
```

können Sie die Ausgabe nach Erreichen eines neuen Höchstwertes anhalten. Versuchen Sie die Startzahlen 4096, 142587, 1501353. Bei der ersten Zahl (4096) ist klar, dass der Algorithmus schnell endet, weil 4096 eine Zweierpotenz ist. Die Frage ist letztlich: Gibt es eine Startzahl, mit der der Algorithmus *nicht* irgendwann endet? Dieses Problem tritt auch unter einer Reihe anderer Namen auf: Syracuse-Problem, Ulams Problem oder Collatz-Problem. Hinweis: Bei großen Zahlen wie der letzten angegebenen wird der `int`-Zahlenbereich überschritten; nehmen Sie stattdessen `long long`.

1.9.4 Strukturen

Ein Vektor hat nur Elemente desselben Datentyps. Häufig möchte man logisch zusammengehörige Daten zusammenfassen, die *nicht* vom selben Datentyp sind, zum Beispiel Vornamen und Nachnamen je vom Typ `string`, Status vom Typ `enum` {Sachbearbeiter, Gruppenleiter, Abteilungsleiter}, Monatsgehalt vom Typ `double` sowie weitere Daten, die zu einem Personaldatensatz zusammengefasst werden sollen. Ein anderes Beispiel sind die zusammengehörigen Daten eines Punktes auf dem Bildschirm, also seine Koordinaten `x` und `y`, seine Farbe, und ob er sichtbar ist. Für diese Zwecke wird von C++ die *Struktur* bereitgestellt, die einen Datensatz zusammenfasst.

Vorweg sei bemerkt, dass eine Struktur in C++ nichts anderes als eine Klasse mit öffentlichen Elementen ist. In Einschränkung dazu enthält eine Struktur im Sinne dieses Abschnitts nur Daten (und keine Funktionen) und entspricht einem Record der Sprache Pascal. Mehr über Klassen finden Sie in Kapitel 4.

Strukturen sind benutzerdefinierte Datentypen. Sie werden mit einer Syntax definiert, die bis auf den inneren Teil der Syntax von Aufzählungstypen ähnelt (siehe Abbildung 1.15).

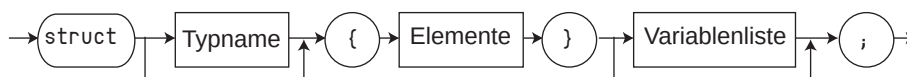


Abbildung 1.15: Syntaxdiagramm einer `struct`-Definition

In einem Grafikprogramm gehören zu jedem Punkt auf dem Bildschirm verschiedene Daten wie die Koordinaten in `x`- und `y`-Richtung, die Farbe und die Information, ob er gerade sichtbar ist. Alle diese logisch zusammengehörenden Daten werden in der Struktur `Punkt` zusammengefasst. Die strukturinternen Daten heißen *Elemente* (auch *Felder* oder

Komponenten). Im Beispiel werden zwei Variablen *p* und *q* vom Datentyp *Punkt* definiert und der Zugriff auf die internen Größen gezeigt.

```
enum Farbtyp {rot, gelb, gruen};

struct Punkt {           // Punkt ist ein Typ.
    int x;
    int y;
    bool istSichtbar;
    Farbtyp dieFarbe;
} p;                      // p ist ein Punkt-Objekt

// q ist ebenfalls ein Punkt-Objekt:
Punkt q;                  // noch undefinierter Inhalt

// Zugriff, hier: mit Werten versehen
p.x = 270;  p.y = 20;      // Koordinaten von p
p.istSichtbar = false;
p.dieFarbe = gelb;
```

Die internen Elemente sind nicht allein zugreifbar, weil sie nur in Verbindung mit einem Objekt existieren. Die Anweisung *dieFarbe = rot;* ist unsinnig, weil nicht klar ist, welcher Punkt rot werden soll. *p.dieFarbe = rot;* hingegen sagt eindeutig, dass der Punkt *p* gefärbt werden soll. Der Zugriff geschieht über einen Punktoperator ».« zwischen Variablenname und Elementnamen, wenn die Variable vom Typ *struct* ist.

Ein *struct* kann in einem Schritt initialisiert werden, wie das folgende Programm zeigt. Bei der Ausgabe wird der *bool*-Wert *false* in 0 umgewandelt, und der *enum*-Wert *gelb* in 1 (rot ergäbe 0).

Listing 1.17: Beispiel mit *struct*

```
// cppbuch/k1/struct.cpp
#include<iostream>
using namespace std;

enum Farbtyp { rot, gelb, gruen };

struct Punkt {
    int x;
    int y;
    bool istSichtbar;
    Farbtyp dieFarbe;
};

int main() {
    Punkt p1 = { 100, 200, false, gelb }; // direkte Initialisierung

    cout << "p1.x = " << p1.x << " p1.y = " << p1.y
        << " p1.istSichtbar = " << p1.istSichtbar
        << " p1.dieFarbe = " << p1.dieFarbe << endl;
}
```



Übung

1.14 Schreiben Sie eine Struktur (struct) namens `Person`, die Vorname, Nachname und Alter einer Person enthält. Vorname und Nachname seien vom Typ `string`, Alter vom Typ `int`. Verwenden Sie diese Struktur in einem Programm so, dass den Elementen der Struktur Werte mit `cin` (Eingabe über die Tastatur) zugewiesen werden. Anschließend sollen die Elemente auf dem Bildschirm ausgegeben werden.

1.9.5 Typermittlung mit `auto`

Das Schlüsselwort `auto` sagt dem Compiler, er soll selbst den Typ bei der Initialisierung ermitteln. Das kann die Schreibarbeit verringern und vermeidet Tippfehler bei komplexen Datentypen. Beispiele:

```
auto a = 2;    // a ist vom Typ int
auto b = 2.9;  // b ist vom Typ double
```

Das folgende Programm zeigt verschiedene Anwendungen von `auto` bei selbstgeschriebenen und Standard-Datentypen:

Listing 1.18: Beispiele mit `auto`

```
// cppbuch/k1/auto.cpp
#include<iostream>
#include<vector>    // Standard-Vektor einschließen
#include<string>    // Standard-String einschließen
#include<cstdint>   // size_t
using namespace std;

struct Punkt {
    int x;
    int y;
};

int main() {
    Punkt p1 = { 100, 200 };
    auto p2 = p1;    // p2 ist vom Typ Punkt
    cout << "p2.x= " << p2.x << " p2.y= " << p2.y << endl;

    vector<double> v1 = {1.1, 2.2, 3.3, 4.4, 5.5};
    auto v2 = v1;    // v2 ist vom Typ vector<double>
    for(size_t i = 0; i < v2.size(); ++i) {
        cout << i << ": " << v2[i] << endl;
    }

    string s1("Ende!");
    auto s2 = s1;    // s2 ist vom Typ string
    cout << s2 << endl;
}
```

In den obigen Fällen sind die Typnamen recht kurz. Dies ist jedoch in Teilen der C++-Standardbibliothek anders, so dass dort das Schlüsselwort `auto` erheblich zur Erleichte-

rung der Schreibarbeit beiträgt. Im Programm liegen die Anweisungen mit `auto` direkt nach der Deklaration, auf die Bezug genommen wird. Bei größeren Programmen ist das nicht unbedingt der Fall, sodass sich als weiterer Vorteil ergibt, dass man nicht irgendwo weiter oben im Programm oder in der Dokumentation der Standardbibliothek die genaue Deklaration nachschlagen muss.

1.9.6 Unions und Bitfelder⁸

Unions

Unions sind Strukturen, in denen verschiedene Elemente *denselben* Speicherplatz bezeichnen. Daher sind Unions nur in Sonderfällen einzusetzen. Im folgenden Beispiel werden eine `int`-Zahl und ein `char`-Array überlagert. Eine Variable vom Typ `char` belegt genau ein Byte. Das Array hat hier also genau so viele Elemente, wie eine `int`-Zahl Bytes hat. Der gesamte Speicherplatz einer Variablen vom Typ `union` ist identisch mit dem Speicherplatzbedarf des jeweils größten internen Elements, in diesem Beispiel also `sizeof(int)` Bytes. Das unsigned `char`-Array `c` belegt denselben Platz wie `zahl`. Die Bezeichnungen `was.zahl` und `was.c[]` beziehen sich auf denselben Bereich im Speicher, nur dass die Interpretation verschieden ist. Das Programmbeispiel gibt die interne Repräsentation einer `int`-Zahl Byte für Byte aus.

Listing 1.19: Union

```
// cppbuch/k1/union.cpp
#include<iostream>
using namespace std;

union HiLo {
    int zahl;
    unsigned char c[sizeof(int)];
};

int main() {
    HiLo was;
    do {
        cout << " Zahl eingeben (0 = Ende):";
        cin >> was.zahl;
        cout << "Byte-weise Darstellung von "
              << was.zahl << endl;
        for(int i = sizeof(int)-1; i >= 0; --i) {
            cout << "Byte Nr. " << i << " = "
                  << static_cast<int>(was.c[i]) << endl;
        }
    } while (was.zahl);
}
```

Bitfelder

Gerade in der Hardware-nahen oder Systemprogrammierung ist es oft wünschenswert, Bitfelder verschiedener Länge anzulegen, in denen die einzelnen Bitpositionen verschie-

⁸ Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

dene Bedeutungen haben. Die Deklaration eines Bitfelds in C++ hat die Syntax *Datentyp Name :Konstante*, wobei *Datentyp* char, short, int, long (mit den vorzeichenlosen unsigned-Varianten) oder ein Aufzählungstyp sein kann. Beispiel: `int Bitfeld : 13;`.

Es dürfen keine Annahmen darüber getroffen werden, wo die 13 Bits innerhalb eines 16- oder 32-Bit-Worts angeordnet sind – dies kann je nach Implementation verschieden sein. Es gibt keine Zeiger oder Referenzen auf ein Bitfeld. Bitfelder sollten nicht zum Sparen von Speicher benutzt werden, weil der Aufwand des Zugriffs auf einzelne Bits beträchtlich sein kann und nicht einmal sicher ist, ob wirklich Speicher gespart wird: Es *könnte* sein, dass eine Implementation jedes Bitfeld auch der Länge 1 stets an einer 32-Bit-Wortgrenze beginnen lässt. Üblich ist allerdings, dass aufeinanderfolgende Bitfelder aneinandergereiht werden. Beispielprogramm zum Zugriff auf Bitfelder (es sind die für int-Typen erlaubten Bit-Operatoren möglich):

Listing 1.20: Bitfeld

```
// cppbuch/k1/bitfeld.cpp
#include<iostream>
using namespace std;

struct Bitfeldstruktur {
    unsigned int a : 4; // a und b sind Bitfelder
    unsigned int b : 3;
};

int main() {
    Bitfeldstruktur x;
    x.a = 06;
    x.b = x.a | 03;
    // Umwandlung in unsigned und Ausgabe
    cout << x.b << endl;
}
```

2

Einfache Ein- und Ausgabe

Dieses Kapitel behandelt die folgenden Themen:

- Abfrage der Tastatureingabe
- Daten aus Dateien lesen
- Daten in Dateien schreiben

Ein- und Ausgabe werden soweit beschrieben, dass Abfragen der Tastatur und Darstellungen auf dem Bildschirm ebenso wie das Lesen und Schreiben von Dateien auf einfache Weise möglich sind. Speziellere Fragen und Einzelheiten werden zunächst zurückgestellt.



Mehr dazu lesen Sie in [Kapitel 10](#)

2.1 Standardein- und -ausgabe

Ein Programm empfängt einen Strom von Eingabedaten, verarbeitet diese Daten und gibt einen Strom von Ausgabedaten aus. Unter »Strom« (englisch *stream*) wird eine Folge

von Bytes verstanden, die nacheinander vom Programm interpretiert beziehungsweise erzeugt werden. In C++ sind einige Ein- und Ausgabekanäle vordefiniert. Sie werden zuerst beschrieben.

```
cin    Standardeingabe (Tastatur)
cout   Standardausgabe (Bildschirm)
cerr   Standardfehlerausgabe (Bildschirm)
```

Eingabe

Der Operator `>>`, der uns in anderem Zusammenhang schon als Bit-Verschiebeoperator begegnet ist, sorgt bei der Eingabe dafür, dass automatisch die nötigen Umformatierungen vorgenommen werden. `int zahl; cin >> zahl;` bewirkt, dass eine Folge von Ziffernzeichen bis zu einem Nicht-Ziffernzeichen eingelesen und in die interne Darstellung einer `int`-Zahl umgewandelt wird. Die Auswertung durch den `>>`-Operator hat bestimmte Eigenschaften:

- Führende Zwischenraumzeichen (englisch *whitespace*) werden ignoriert. Zwischenraumzeichen sind Leerzeichen, Tabulatorzeichen `'\t'`, Zeilenrücklauf `'\r'`, Zeilensprung `'\v'`, Seitenvorschub `'\f'` und die Zeilenendekennung `'\n'`. In der ASCII-Tabelle auf Seite 887 sind es die Zeichen 0x20 und 0x09 bis 0x0d in Hexadezimalschreibweise.
- Zwischenraumzeichen werden als Endekennung genutzt.
- Andere Zeichen werden entsprechend dem verlangten Datentyp interpretiert.

Sollen Zwischenraumzeichen nicht ignoriert werden, ist die Funktion `get()` zu verwenden, die zum Einlesen einzelner Zeichen, also nicht von Zahlen, verwendet werden kann:

```
// einzelnes Zeichen einlesen
char c;
cin.get(c);
```

Der nach außen hin nicht sichtbare Ablauf einer Tastaturabfrage mit `»cin >>«` besteht aus mehreren Schritten, wobei angenommen wird, dass noch nichts auf der Tastatur eingegeben worden ist:

1. Aufforderung an das Betriebssystem zur Zeichenübergabe.
2. Eingabe der Zeichen auf der Tastatur (mit Korrekturmöglichkeit durch die Backspace-Taste). Die Zeichen werden vom Betriebssystem der Reihe nach in einem besonderen Speicherbereich abgelegt, dem Tastaturpuffer.
3. Abschluss der Eingabe mit der ENTER-Taste. Damit wird das `'\n'`-Zeichen als Zeilenendekennung im Tastaturpuffer abgelegt, und der Puffer wird durch das Betriebssystem an C++ übergeben.
4. Auswertung des Tastaturpufferinhalts durch den Operator `>>` je nach Datentyp der gefragten Variable.
5. Daten, die nach der Auswertung übrigbleiben, weil sie nicht zu dem Datentyp passen, verbleiben im Tastaturpuffer und werden mit dem nächsten `cin >>` gelesen.

Das Beispielprogramm verlangt der Reihe nach eine `int`- und eine `double`-Zahl. Falls das Format mit dem aktuell erwarteten Datentyp nicht übereinstimmt, wird die Schleife abgebrochen, beispielsweise bei Eingabe eines Buchstabens¹.

¹ Zur Auswertung der `while`-Bedingung siehe Abschnitt 10.5.

Listing 2.1: Zahl einlesen

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

int main( ) {
    int i;
    double d;
    while(cin >> i >> d) {
        cout << i << endl << d << endl;
    }
}
```

Wenn 100_12.4 ENTER eingegeben wird, wobei _ für irgendein Whitespace-Zeichen steht, ist das Ergebnis wie erwartet `i == 100` und `d == 12.4`. Falls die nächste Eingabe 2.7 lautet, erhalten wir 2 und 0.7 als Ausgabe. Warum? An dieser Stelle wird eine `int`-Zahl erwartet. Es wird also die 2 als `int`-Zahl gelesen, der Dezimalpunkt gehört *nicht* mehr dazu. Die folgenden Zeichen .7 werden als `double` interpretiert und als 0.7 ausgegeben.

Eingabe von Strings

Die Eingabe von Strings unterscheidet sich nicht von der Eingabe von Zahlen, wie oben beschrieben. Häufig möchte man jedoch nicht nur durch Zwischenraumzeichen getrennte Zeichenfolgen einlesen, sondern ganze Zeilen. Beispielsweise sollen Vor- und Nachname eingegeben werden. Das Programm dazu sei:

Listing 2.2: Einfache String-Eingabe

```
// cppbuch/k2/eingabe1.cpp
#include<iostream>
#include<string>
using namespace std;

int main() {
    cout << "Bitte Vor- und Nachnamen eingeben:";
    string derName;
    cin >> derName;
    cout << derName;
}
```

Sie tippen ein:

Donald_Duck ENTER

und sind über die knappe Ausgabe *Donald* nicht erbaut. Das war es nicht, was Sie wollten! Der `>>`-Operator versteht die hier mit _ gekennzeichneten Leerzeichen als Endeckennung, weswegen der Rest des Namens im Tastaturpuffer hängen bleibt. Er könnte durch weitere `cin >> ...` Anweisungen ausgelesen werden. Besser geht es mit der Funktion `getline()`, die eine ganze Zeile einliest. Dazu wird `main()` etwas modifiziert:

```
int main() {
    cout << "Bitte Vor- und Nachnamen eingeben:";
    string derName;
    getline(cin, derName); // mehr auf Seite 382
```

```
cout << derName; // Donald Duck
}
```

Ausgabe

Der Operator `<<` formt automatisch aus der internen Darstellung in eine Textdarstellung um. Es wird stets die mindestens notwendige Weite genommen: `cout << 7 << 11;` erscheint als »711«. Formatierungen sind möglich, zum Beispiel:

```
cout << 7;
cout.width(6);
cout << 11;
```

Jetzt wird »7 11« angezeigt, weil vor der Zahl 11 vier Leerzeichen eingefügt werden, um die Gesamtweite von sechs Zeichen zu erreichen. Weitere Formatierungen sind im Abschnitt 10.1.1 beschrieben. Ein Beispiel zur formatierten tabellarischen Ausgabe von Zahlen mit einer festgelegten Zahl von Nachkommastellen finden Sie dort auf Seite 380.

`cin` und `cout` können auf Betriebssystemebene mit `<` beziehungsweise `>` umgeleitet werden. Wenn ein Programm namens *prog1* mit Bildschirmausgabe und Tastatureingabe korrekt kompiliert und gelinkt worden ist, können mit den Umleitungszeichen `<` und `>` die zugehörigen Eingabedaten aus einer Datei *eingabe* anstelle der Tastatur geholt werden. Anstelle der Bildschirmausgabe kann in eine Datei *ausgabe* geschrieben werden, wobei die Dateinamen natürlich wählbar sind: `prog1 < eingabe > ausgabe`.

2.2 Ein- und Ausgabe mit Dateien

Die Ein- und Ausgabeoperatoren `>>` und `<<` sind bei Dateien ebenso wie bei der Standard-ein- und -ausgabe verwendbar. Für die Funktion `get()` zur Eingabe eines Zeichens gibt es das Gegenstück `put()` zur Ausgabe eines Zeichens. Der Header `<fstream>` enthält die vom Compiler verlangten Beschreibungen für Dateiobjekte. Die benutzten Funktionen `get()`, `put()`, `open()` und `close()` und weitere nutzen die Dateifunktionen des zugrunde liegenden Betriebssystems. Ein Programm, das auf Dateien zugreift, enthält folgende wesentliche Elemente:

- Es wird ein Dateiojekt mit einem beliebigen Namen definiert, das von nun an im Programm verwendet wird. Der Datentyp des Dateiojekts ist `ifstream` für Ein- und `ofstream` für Ausgabedateien (Abkürzungen für input file stream beziehungsweise output file stream).
- Die Verbindung des Dateiojekts zu einer existierenden oder anzulegenden Datei auf Betriebssystemebene wird mit der Funktion `open()` hergestellt, d.h. die Datei wird »geöffnet«. Der externe Dateiname wird der Funktion `open` als Zeichenkette übergeben. Dadurch kann ein und dasselbe Programm auf beliebige Dateien zugreifen. Eine zu beschreibende Datei wird normalerweise *exklusiv* reserviert, kann also nicht gleichzeitig von anderen Programmen beschrieben oder gelesen werden.

- Die Verbindung wird mit der Funktion `close()` wieder gelöst; man sagt auch, dass die Datei geschlossen wird. Ab diesem Zeitpunkt kann die Datei wieder von anderen Benutzern oder Programmen benutzt werden.

Das Schreiben geschieht im Allgemeinen gepuffert, indem in einen dafür reservierten Speicherbereich (Puffer) geschrieben wird, der erst bei Überlauf auf die Festplatte transferiert wird. Für das Lesen gilt Entsprechendes. `close()` sorgt dafür, dass der im Puffer befindliche Rest geschrieben und das Inhaltsverzeichnis (englisch *directory*) aktualisiert wird.

Bei Programmende wird automatisch ein `close()` durchgeführt. Es empfiehlt sich jedoch, eine Datei zu schließen, sobald nichts mehr geschrieben werden soll und noch weitere Programmteile folgen. Die Begründung: Ein irregulärer Programmabbruch, aus welchen Gründen auch immer, verhindert das Schreiben des Pufferinhalts und die Aktualisierung des Inhaltsverzeichnisses auf der Festplatte für eine zum Zeitpunkt des Abbruchs noch geöffnete Datei. Sie ist danach, zum Beispiel für ein weiteres Programm, nur teilweise oder gar nicht mehr lesbar.

Beispiel: Kopieren von Dateien

Das Beispielprogramm zum Kopieren beliebiger Dateien teilt sich in die Abschnitte

- Definieren und Öffnen der Eingabedatei mit Programmabbruch, falls die Datei nicht existiert
- Definieren und Öffnen der Ausgabedatei mit Programmabbruch, falls die Datei existiert, aber nicht beschrieben werden darf (*read-only*-Datei)
- Kopiervorgang – die Schleife wird bei Fehlern oder am Dateiende abgebrochen
- (automatisches) Schließen beider Dateien

Der verwendete Aufruf `exit(int)` beendet das Programm im Fehlerfall, wobei der übergebene Parameter an das Betriebssystem gemeldet wird.

Listing 2.3: Datei kopieren

```
// cppbuch/k2/datscopy.cpp kopiert eine Datei
#include<cstdlib> // für exit( )
#include<fstream>
#include<iostream>
#include<string>
using namespace std;

int main( ) {
    // Definieren der Eingangsdatei
    ifstream quelle;           // Datentyp für Eingabestrom
    string quelldateiname;
    cout << "Quelldatei?";
    cin >> quelldateiname; // Darf wegen der Eigenschaften von cin
                          // keine Leerzeichen enthalten!
    // Datei öffnen. Zu ios::binary siehe Text.
    quelle.open(quelldateiname.c_str(), ios::binary|ios::in);
    if(!quelle) { // Fehlerabfrage
        cerr << quelldateiname
              << " kann nicht geöffnet werden!\n";
```

```

        exit(-1);
    }
    string zieldateiname;
    cout << "Zieldatei? ";
    cin >> zieldateiname; // Darf wegen der Eigenschaften von cin
                        // keine Leerzeichen enthalten!
    // Definieren und Öffnen der Ausgabedatei hier in einem Schritt.
    // ios::binary muss bei ofstreams mit ios::out verknüpft werden.
    ofstream ziel(zieldateiname.c_str(), ios::binary|ios::out);
    if(!ziel) {
        cerr << zieldateiname
              << " kann nicht geöffnet werden!\n";
        exit(-1);
    }
    char ch;
    while(quelle.get(ch))
        ziel.put(ch); // zeichenweise kopieren
} // Dateien werden am Programmende automatisch geschlossen.

```



Anmerkung zu ios::binary

Ohne `ios::binary` wären nur Textdateien kopierbar. Der Schalter `ios::binary` verhindert, dass die Umwandlung der Zeilenendekennung `\n` in `CR/LF` (= `0x0d 0x0a`) automatisch beim Schreiben beziehungsweise zurück beim Lesen erfolgen soll (nur bei MS DOS/Windows und OS/2 von Bedeutung). `ios::binary` muss bei `ifstreams` mit `ios::in` verknüpft werden.

Dateien beliebigen Inhalts (binäre Dateien) haben keine Zeilenstruktur. Die Funktion `c_str()` stellt den Quelldateinamen in einer für `open` verträglichen Form dar, nämlich als C-String. C-Strings werden ab Seite 193 behandelt. Falls das Programm nach Abschluss des Kopierens fortgesetzt werden sollte, wäre die Ergänzung um die Anweisungen `quelle.close();` und `ziel.close();` sinnvoll (vergleiche obige Diskussion). In den Abfragen `if(!quelle)` und `if(!ziel)` wird im Programmbeispiel scheinbar auf die Negation des Dateiobjekts geprüft, um einen Fehler festzustellen. Die Erklärung dafür muss auf den Abschnitt 10.5 verschoben werden.



Tipp

Wenn ein Objekt für eine Eingabedatei *mehrfach* benutzt wird, etwa um mehrere Dateien einzulesen, sind nach dem `close()` die objektinternen Statusbits mit `clear()` zu löschen.

Der Grund: Beim Aufruf von `close()` wird `clear()` leider nicht von jedem Compiler automatisch ausgeführt, sodass das Objekt, hier `quelle` genannt, sich möglicherweise noch am Ende einer Datei wähnt (EOF = end of file), obwohl es schon ein neues `open()` gab. Beispiel:

```

quelle.open("ersteDatei.txt");
// ... hier die Datei verarbeiten
quelle.close(); // Datei schließen
quelle.clear(); // Statusbits löschen nicht vergessen!

```

```
quelle.open("zweiteDatei.txt");
// ... hier die nächste Datei verarbeiten usw.
```



Übungen

2.1 (Fast) Jeder braucht mal einen Kredit. Schreiben Sie ein Programm zur Berechnung eines Tilgungsplans für einen Ratenkredit, damit Sie gegebenenfalls Ihre Bank kontrollieren können. Es sei vorausgesetzt, dass die Rate monatlich gezahlt wird. Das Programm verlange die folgenden Eingaben: Kreditumfang (in Euro), nominaler Zinssatz in Prozent (pro Jahr), Anfangsmonat, Anfangsjahr, Höhe der monatlichen Rate, Laufzeit des Kredits in Jahren. Der Tilgungsplan soll in eine Datei *tilgungsplan.txt* ausgegeben werden. Von der Rate werden zunächst die Zinsen beglichen, der Rest wird zur Tilgung der Restschuld verwendet. Daraus ergibt sich, dass die Rate größer als die anfänglichen Zinsen sein muss, wenn die Restschuld kleiner werden soll. Die Laufzeit ist die Zinsbindungsfrist, nach deren Ablauf die Restschuld zurückgezahlt oder ein neuer Zinssatz festgelegt wird. Damit das Programm in der Praxis nutzbar sein kann, muss kaufmännisch auf ganze Cent-Beträge gerundet werden. Die auszugebende Datei könnte wie folgt aussehen:

Anfangsschulden : 20000.00 Zinssatz nominal : 5.00

Zahlmonat	Rate	Zinsen	Tilgung	Rest
1.2012	500.00	83.33	416.67	19583.33
2.2012	500.00	81.60	418.40	19164.93

⋮

usw. bis zum letzten Jahr:

Zahlmonat	Rate	Zinsen	Tilgung	Rest
1.2015	500.00	16.05	483.95	3368.84

⋮

7.2015	500.00	3.83	496.17	422.53
8.2015	424.29	1.76	422.53	0.00
Summen:	3924.29	71.50	3852.79	pro Jahr

Gesamt: 21924.29 1924.29 20000.00

Bei kleinen Raten endet die Tabelle mit dem Ende der Laufzeit, in diesem Beispiel jedoch wegen der großen Raten schon früher. Hinweis: Die Spaltenbreite kann vor jeder Ausgabe mit `ausgabe.width(10)` eingestellt werden. `ausgabe` sei der zur Datei gehörige `ofstream`. Mit

```
ausgabe.setf(ios::showpoint|ios::fixed, ios::floatfield);
ausgabe.precision(2); // Nachkommastellen
```

wird die Ausgabe mit Dezimalpunkt und zwei Nachkommastellen eingestellt.



Mehr dazu lesen Sie in Kapitel 10

2.2 Schreiben Sie ein Programm, das den Inhalt einer Datei hexadezimal ausdruckt.

2.3 Erweiterung zur vorigen Aufgabe: Erst 16 Buchstaben und dann die zugehörigen 16 Hex-Codes pro Zeile ausgeben.

2.4 Schreiben Sie ein Programm *stat.cpp*, das eine Statistik für eine Textdatei ausgibt, deren Name eingegeben werden soll. Das Ergebnis soll eine Ausgabe folgender Art hervorbringen:

Anzahl der Zeichen = 16437

Anzahl der Worte = 2526

Anzahl der Zeilen = 220

Ein Wort sei hier als ununterbrochene Folge von Buchstaben definiert, wobei Umlaute hier nicht als Buchstaben zählen sollen, weil sie nicht Teil des ASCII-Zeichensatzes sind. In der Anzahl der Zeichen soll die Zeilenendekennung nicht enthalten sein.

3

Programmstrukturierung

Dieses Kapitel behandelt die folgenden Themen:

- Funktionen und ihr Aufbau
- Rückgabe von Funktionsergebnissen
- Die verschiedenen Arten der Parameterübergabe
- Makros
- Modulare Gestaltung und Strukturierung von Programmen
- Funktionen mit parametrisierten Datentypen
- Namespaces

Große Programme müssen in übersichtliche Teile zerlegt werden. Sie werden dazu verschiedene Mechanismen kennenlernen. Sie erfahren, wie eine Funktion aufgebaut ist, wie man einer Funktion die von ihr benötigten Daten mitteilen kann und auf welche Weise die Ergebnisse von Funktionen zurückgegeben werden können. Die Simulation eines Taschenrechners zeigt beispielhaft den wechselseitigen Einsatz von Funktionen. Anschließend werden Grundsätze der modularen Gestaltung behandelt, ohne deren Einhaltung große Programme oder Programmsysteme kaum mehr handhabbar sind. Der Einsatz von Funktionen mit parametrisierten Datentypen ermöglicht einen breiteren Einsatz von Funktionen ohne fehlerträchtige Vervielfachung des Programmcodes.

3.1 Funktionen

Eine Funktion erledigt eine abgeschlossene Teilaufgabe. Die Teilaufgabe kann einfach, aber auch sehr komplex sein. Die notwendigen Daten werden der Funktion mitgegeben, und sie gibt das Ergebnis der erledigten Aufgabe an den Aufrufer (Auftraggeber) zurück. Eine einfache mathematische Funktion ist zum Beispiel $y = \sin(x)$, wobei x der Funktion als notwendiges Datum übergeben und y das Ergebnis zugewiesen wird. In C++ können die verschiedensten Funktionen programmiert oder benutzt werden, nicht nur mathematische.

Eine Funktion muss nur einmal definiert werden. Anschließend kann sie beliebig oft nur durch Nennung ihres Namens aufgerufen werden, um die ihr zugewiesene Teilaufgabe abzuarbeiten. Dieses Prinzip setzt sich in dem Sinne fort, dass Teilaufgaben selbst wieder in weitere Teilaufgaben unterteilbar sein können, die durch Funktionen zu bearbeiten sind. Wie in einer großen Firma die Aufgaben nur durch Arbeitsteilung, Delegation und einer sich daraus ergebenden hierarchischen Struktur zu bewältigen sind, wird in der Informatik die Komplexität einer Aufgabe durch Zerlegung in Teilaufgaben auf mehreren Ebenen reduziert – nach dem Prinzip »Teile und herrsche«. Bereits vorhandene Standardlösungen von Teilaufgaben können aus Funktionsbibliotheken abgerufen werden – ebenso wie neu entwickelte Funktionen in Bibliotheken aufgenommen werden können.

3.1.1 Aufbau und Prototypen

Auf Seite 74 wird die Fakultät einer Zahl berechnet. Dies soll die Grundlage für eine einfache Funktion `fakultaet()` bilden, die diese Aufgabe ausführt. Die Fakultät $n!$ ist definiert als das Produkt $n(n-1)(n-2)\dots 3 \cdot 2 \cdot 1$. Dabei ist $0! = 1$ festgelegt. Das Beispiel zeigt die Integration der Funktion in ein `main`-Programm:

Listing 3.1: Beispielprogramm mit einer Funktion

```
// cppbuch/k3/fakulta2.cpp
#include<iostream>
using namespace std;
unsigned long fakultaet(int);      // Funktionsprototyp (Deklaration)

int main() {
    int n;
    do {
        cout << "Fakultät berechnen. Zahl >= 0? :";
        cin >> n;
    } while(n < 0);
    cout << "Das Ergebnis ist " << fakultaet(n) << endl;    // Aufruf
}

unsigned long fakultaet(int zahl) { // Funktionsimplementation (Definition)
    unsigned long fak = 1;
    for(int i = 2; i <= zahl; ++i)
        fak *= i;
    return fak;
}
```

Eine *Deklaration* sagt dem Compiler, dass eine Funktion oder eine Variable mit diesem Aussehen irgendwo definiert ist. Damit kennt er den Namen bereits, wenn er auf einen Aufruf der Funktion stößt, und ist in der Lage, eine Syntaxprüfung vorzunehmen. Eine *Definition* veranlasst den Compiler, entsprechenden Code zu erzeugen und den notwendigen Speicherplatz anzulegen. Eine Funktionsdeklaration, die nicht gleichzeitig eine Definition ist, wird *Funktionsprototyp* genannt. Eine Vereinbarung einer Variablen mit `int i;` ist sowohl eine Deklaration als auch eine Definition. Auf die Begriffe Deklaration und Definition wird in Abschnitt 3.3.4 genauer eingegangen. Der Aufruf der Funktion geschieht einfach durch Namensnennung. Von der Funktion auszuwertende Daten werden in runden Klammern `()` übergeben. Der Funktionstyp `void` bedeutet, dass nichts zurückgegeben wird. Wenn eine Funktion einen Rückgabotyp ungleich `void` hat, muss im Funktionskörper `{...}` irgendwo ein Ergebnis dieses Typs mit der Anweisung `return` zurückgegeben werden. Wenn eine Funktion etwas tut, ohne dass ein Funktionsergebnis zurückgegeben wird, wirkt sie nur durch sogenannte *Seiteneffekte*. Andere Möglichkeiten der Ergebnismrückgabe werden in Abschnitt 3.2 vorgestellt. Die Wirkung eines Funktionsaufrufs ist, dass das zurückgegebene Ergebnis an die Stelle des Aufrufs tritt! Abbildung 3.1 zeigt die Syntax eines *Funktionsprototyps* (siehe obiges Beispiel).



Abbildung 3.1: Syntaxdiagramm eines Funktionsprototyps

Der Rückgabotyp, auch Typ der Funktion genannt, kann ein nahezu beliebiger Datentyp sein. Ausnahmen sind die Rückgabe einer Funktion sowie die Rückgabe des bisher noch nicht besprochenen C-Arrays. Betrachten Sie die Zuordnung der einzelnen Teile der obigen Deklaration von `fakultaet()`:

```

unsigned long      fakultaet      (      int      );
      ⋮              ⋮              ⋮              ⋮
Rückgabotyp  Funktionsname  (  Parameterliste  );
  
```

Die *Parameterliste* besteht in diesem Fall nur aus einem einzigen Parametertyp. Je nach Aufgabenstellung bestehen für den Aufbau einer *Parameterliste* folgende Möglichkeiten:

	Beispiel:
leere Liste:	<code>int func();</code>
gleichwertig ist:	<code>int func(void);</code>
Liste mit Parametertypen:	<code>int func(int, char);</code>
Liste mit Parametertypen und -namen:	<code>int func(int x, char y);</code>

Parameternamen wie `x` und `y` dienen der Erläuterung. Sie dürfen entfallen, was aber nur dann tolerierbar ist, wenn der Sinn unmissverständlich ist. In allen anderen Fällen ist es vorteilhafter, die Namen hinzuschreiben, damit später die Benutzung der Funktion sofort klar wird, ohne die Dokumentation bemühen zu müssen. Abbildung 3.2 zeigt die Syntax einer *Funktionsdefinition*. Der eigentliche Programmcode ist im Block der Funktionsdefinition enthalten. Betrachten wir auch jetzt die Zuordnung der einzelnen Teile der obigen Definition von `fakultaet()`, wobei der Programmcode durch »...« angedeutet ist:



Abbildung 3.2: Syntaxdiagramm einer Funktionsdefinition

```

unsigned long    fakultaet    (    int zahl    ) {...}
    :            :            :            :      :
    :            :            :            :      :
Rückgabtyp  Funktionsname  (  Formalparameterliste  )  Block
  
```

Die *Formalparameterliste* enthält im Unterschied zur reinen Deklaration zwingend einen Parameternamen (hier *zahl*), der damit innerhalb des Blocks bekannt ist. Der Name ist frei wählbar und völlig unabhängig vom Aufruf, weil er nur als Platzhalter dient. Abbildung 3.3 zeigt die Syntax eines *Funktionsaufrufs*.



Abbildung 3.3: Syntaxdiagramm eines Funktionsaufrufs

Die *Aktualparameterliste* enthält Ausdrücke und/oder Namen der Objekte oder Variablen, die an die Funktion übergeben werden sollen. Sie kann leer sein. In unserem Beispiel besteht die Aktualparameterliste nur aus *n*. Dass der Datentyp von *n* mit dem Datentyp in der *Deklaration* übereinstimmt, wird vom Compiler geprüft. Der Linker stellt fest, ob eine entsprechende *Definition* der Funktion mit dem richtigen Datentyp in der Formalparameterliste vorhanden ist. Der Aufruf der Funktion bewirkt, dass der Wert von *n* an die Stelle des Platzhalters *zahl* gesetzt und dann der Programmcode im Block durchgeführt wird. Am Schluss wird die berechnete Fakultät mit dem richtigen Ergebnisdatentyp zurückgegeben. Zurückgegeben wird nur der Wert von *fak*, nicht *fak* selbst. Die Variablen *fak* und *zahl* sind *lokal*, d.h. im Hauptprogramm nicht bekannt und nicht zugreifbar. Ergebnissrückgabe heißt einfach, dass an die Stelle des Aufrufs von *fakultaet()* im Hauptprogramm das Ergebnis eingesetzt wird.

Das Prinzip der Ersetzung der Formalparameter durch die Aktualparameter ist eine wichtige Voraussetzung, um eine Funktion universell verwenden zu können. Es ist ganz gleichgültig, ob die Funktion in einem Programm mit *fakultaet(zahl)* oder in einem anderen Programm mit *fakultaet(xyz)* aufgerufen wird, wenn nur der Datentyp des Parameters mit dem vorgegebenen (in diesem Fall *int*) übereinstimmt.

3.1.2 Gültigkeitsbereiche und Sichtbarkeit in Funktionen

In C++ gelten Gültigkeits- und Sichtbarkeitsregeln für Variable (siehe Seite 58). Die gleichen Regeln gelten auch für Funktionen. Der Funktionskörper ist ein Block, also ein durch geschweifte Klammern `{ }` begrenztes Programmstück. Danach sind alle Variablen einer Funktion nicht im Hauptprogramm gültig und auch nicht sichtbar. Eine Sonderstellung haben die in der Parameterliste aufgeführten Variablen: Sie werden innerhalb der Funktion wie lokale Variable betrachtet, und von außen gesehen stellen sie die *Datenschnittstelle* zur Funktion dar. Die Datenschnittstelle ist ein Übergabepunkt für Daten. *Eingabeparameter* dienen zur Übermittlung von Daten an die Funktion, und über *Ausgabeparameter* (Abschnitt 3.2.2) sowie den `return`-Mechanismus gibt eine Funktion Daten

an den Aufrufer zurück. Die Variable `zahl` aus `fakultaet()` ist also von `main()` aus nicht zugreifbar, wie umgekehrt alle in `main()` deklarierten Variablen in `fakultaet()` nicht benutzt werden können. Diese Variablen sind *lokal*. Ein Beispiel soll das verdeutlichen, wobei hier die *Deklaration* von `f1()` gleichzeitig eine *Definition* ist, weil sie nicht nur den Namen vor dem Aufruf von `f1()` einführt, sondern auch den Funktionskörper enthält. Dieses Vorgehen ist nur für sehr kleine Programme wie hier zu empfehlen.

Listing 3.2: Sichtbarkeitsbereich

```
// cppbuch/k3/scope.cpp
#include<iostream>
using namespace std;

int a = 1;                // überall bekannt, also global

void f1( ) {
    int c = 3;            // nur in f1() bekannt, also lokal
    cout << "f1: c= " << c << endl;
    cout << "f1: globales a= " << a << endl;
}

int main() {
    cout << "main: globales a= " << a << endl;
    // cout << "f1: c= " << c; ist nicht compilierbar, weil c in main() unbekannt ist.
    f1( );                // Aufruf von f1()
}
```

Das Programm erzeugt folgende Ausgabe:

```
main: globales a= 1
f1: c= 3
f1: globales a= 1
```

Beim Betreten eines Blocks wird für die innerhalb des Blocks deklarierten Variablen Speicherplatz beschafft; die Variablen werden gegebenenfalls initialisiert. Der Speicherplatz wird bei Verlassen des Blocks wieder freigegeben. Dies gilt auch für Variablen in Funktionen, wobei der Aufruf einer Funktion dem Betreten des Blocks entspricht. Die Rückkehr zum Aufrufer der Funktion wirkt wie das Verlassen des Blocks.

3.1.3 Lokale static-Variable: Funktion mit Gedächtnis

Die Ausnahme bilden Variablen, die innerhalb eines Blocks oder einer Funktion als `static` definiert werden. Wenn es Konstante sind, die schon zur Compilationszeit bekannt sind, geschieht die Initialisierung *vor* dem Aufruf jedweder Funktion. In allen anderen Fällen wird die Variable *beim ersten Aufruf der Funktion* initialisiert. Im Beispiel unten wird `anz` schon vor dem Aufruf von `func()` mit 0 initialisiert (zur Compilationszeit bekannte Konstante). Würde `anz` den Wert von einer anderen Funktion `g()` erhalten, zum Beispiel `static int anz = g();`, dann würde `anz` erst beim ersten Aufruf von `func()` initialisiert. Falls kein Initialisierungswert vorgegeben ist, werden `static`-Zahlen auf 0 gesetzt. `static`-Variable wirken wie ein Gedächtnis für eine Funktion, weil sie zwischen Funktionsaufrufen ihren Wert nicht verlieren. Eine Funktion, die anzeigt, wie oft sie aufgerufen wurde, sieht so aus:

Listing 3.3: Funktion mit Gedächtnis

```
// cppbuch/k3/static.cpp
#include<iostream>
using namespace std;

void func( ) {           // zählt die Anzahl der Aufrufe
    static int anz = 0;   // behält den letzten Wert
    cout << "Anzahl = " << ++anz << endl;
}

int main() {
    for(int i = 0; i < 3; ++i)
        func( );
}
```

Die Ausgabe des Programms ist

```
Anzahl = 1
Anzahl = 2
Anzahl = 3
```

Ohne das Schlüsselwort `static` würde drei Mal 1 ausgegeben werden, weil die Zählung stets bei 0 begänne. Lokale `static`-Variablen sind globalen Variablen vorzuziehen, weil unabsichtliche Änderungen in anderen Funktionen vermieden werden und mit dieser Variablen verbundene Fehler leichter lokalisiert werden können. Außerdem erfordert eine globale Variable eine Absprache unter allen Benutzern der Funktion über den Namen. Gerade das soll aber vermieden werden, um eine Funktion universell einsetzbar zu machen. Auf die dateiübergreifende Gültigkeit von Variablen und Funktionen wird in Abschnitt 3.3.3 eingegangen.

3.2 Schnittstellen zum Datentransfer

Der Datentransfer in Funktionen hinein und aus Funktionen heraus kann unterschiedlich gestaltet werden. Er wird durch die Beschreibung der Schnittstelle festgelegt. Unter Schnittstelle ist eine formale Vereinbarung zwischen Aufrufer und Funktion über die Art und Weise des Datentransports zu verstehen und darüber, was die Funktion leistet. In diesem Zusammenhang sei nur der Datenfluss betrachtet. Die Schnittstelle wird durch den Funktionsprototyp eindeutig beschrieben und enthält

- den Rückgabotyp der Funktion,
- den Funktionsnamen,
- Parametertypen, die der Funktion bekannt gemacht werden, und somit
- die Art der Parameterübergabe.

Der Compiler prüft, ob die Definition der Schnittstelle bei einem Funktionsaufruf eingehalten wird. Zusätzlich zur Rückgabe eines Funktionswerts gibt es die Möglichkeit, die

an die Funktion über die Parameterliste gegebenen Daten zu modifizieren. Danach unterscheiden wir *zwei Arten des Datentransports*: die Übergabe *per Wert* und *per Referenz*.

3.2.1 Übergabe per Wert

Der Wert wird *kopiert* und der Funktion übergeben. Innerhalb der Funktion wird mit der *Kopie* weitergearbeitet, das Original beim Aufrufer *bleibt unverändert erhalten*. Im Beispiel wird beim Aufruf der Funktion `addiere_5()` der aktuelle Wert von `i` in die funktionslokale Variable `x` kopiert, die in der Funktion verändert wird. Der Rückgabewert wird der Variablen `erg` zugewiesen, `i` hat nach dem Aufruf denselben Wert wie zuvor. Abbildung 3.4 verdeutlicht den Ablauf.

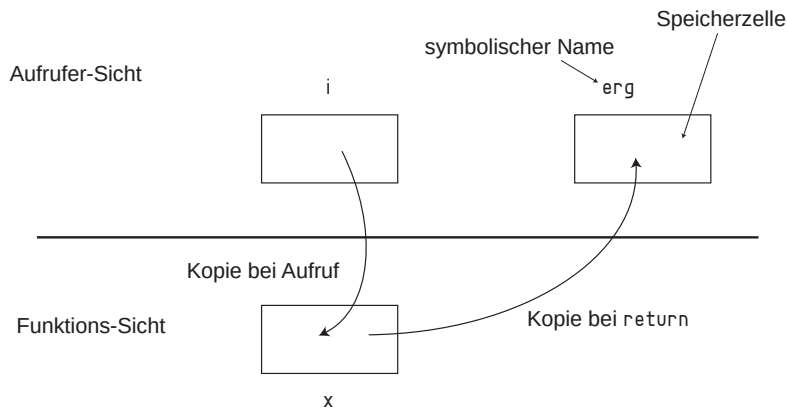


Abbildung 3.4: Parameterübergabe per Wert (Bezug: Programmbeispiel)

Listing 3.4: Übergabe per Wert

```
// cppbuch/k3/per_wert.cpp
#include<iostream>
using namespace std;
int addiere_5(int);    // Deklaration (Funktionsprototyp)

int main() {
    int erg, i = 0;
    cout << i << " = Wert von i\n";
    erg = addiere_5(i);
    cout << erg << " = Ergebnis von addiere_5\n";
    cout << i << " = i unverändert!\n";
}

int addiere_5(int x) { // Definition
    x += 5;
    return x;
}
```

Die Übergabe per Wert soll generell bevorzugt werden, wenn ein Objekt nicht geändert werden soll und es nicht viel Speicherplatz einnimmt. Letzteres ist für Grunddatentypen der Fall.



Übungen

3.1 Schreiben Sie eine Funktion `int dauerInSekunden(int stunden, int minuten, int sekunden)`, die die Gesamtzahl der Sekunden zurückgibt, berechnet aus den Parametern.

3.2 Schreiben Sie eine Funktion `double power(double x, int y)`, die x^y berechnen soll. Wenn Sie nicht mehr genau wissen sollten, was x^y bedeutet – hier ein paar Beispiele: $x^3 = x \cdot x \cdot x$, $x^{-2} = 1/(x \cdot x)$, $x^0 = 1$.

Rekursion

Innerhalb von Funktionen können andere Funktionen aufgerufen werden, die wiederum andere Funktionen aufrufen. Die Verschachtelung kann beliebig tief sein. Der Aufruf einer Funktion durch sich selbst wird *Rekursion* genannt. Das Programm zur Berechnung der Quersumme einer Zahl zeigt die Rekursion:

Listing 3.5: Beispielprogramm 1 mit Rekursion

```
// cppbuch/k3/qsum.cpp
#include<iostream>
using namespace std;

int qsum(long z){
    // Parameter per Wert übergeben
    if(z != 0 ) {
        int letzteZiffer = z % 10;
        return letzteZiffer + qsum(z/10); // Rekursion
    }
    else { // Abbruchbedingung z == 0
        return 0;
    }
}

int main() {
    cout << "Zahl: ";
    long zahl;
    cin >> zahl;
    cout << "Quersumme = " << qsum(zahl);
}
```

Die letzte Ziffer einer Zahl erhält man durch modulo 10 (Restbildung), und sie kann durch ganzzahlige Division durch 10 von der Zahl abgetrennt werden. Anstatt die Summation in einer Schleife vorzunehmen, lässt sich das Prinzip des Programms in zwei Sätzen zusammenfassen:

1. Die Quersumme der Zahl 0 ist 0.
2. Die Quersumme einer Zahl ist gleich der letzten Ziffer plus der Quersumme der Zahl, die um diese Ziffer gekürzt wurde.

Die Quersumme von 348156 ist also (6 + die Quersumme von 34815). Auf jede Quersumme wird Satz 2 angewendet, bis Satz 1 gilt. Durch das sukzessive Abtrennen wird die Zahl irgendwann 0, sodass Satz 1 erfüllt ist und die Rekursion anhält. In diesem Fall ist die Verschachtelungstiefe gleich der Anzahl der Ziffern. Eine Rekursion *muss* auf eine

Abbruchbedingung zulaufen, damit keine unendlich tiefe Verschachtelung entsteht mit der Folge eines Stacküberlaufs. Zum Vergleich sei hier eine iterative Variante gezeigt:

```
int qsum(long z) {
    int sum = 0;
    while(z > 0) {
        sum += z % 10;
        z = z / 10;
    }
    return sum;
}
```

Eines der bekanntesten Beispiele zur Rekursion sind die »Türme von Hanoi«. Dieses Beispiel hat eine leicht zu entwickelnde rekursive Lösung. Eine nicht-rekursive Lösung ist komplizierter und schwieriger zu finden. Die Geschichte: Buddhistische Mönche des Brahma-Tempels haben die Aufgabe, 64 Scheiben aus Gold, die ein Loch in der Mitte haben, von Stab A nach Stab B zu bringen. Stab C kann als Zwischenablage dienen.

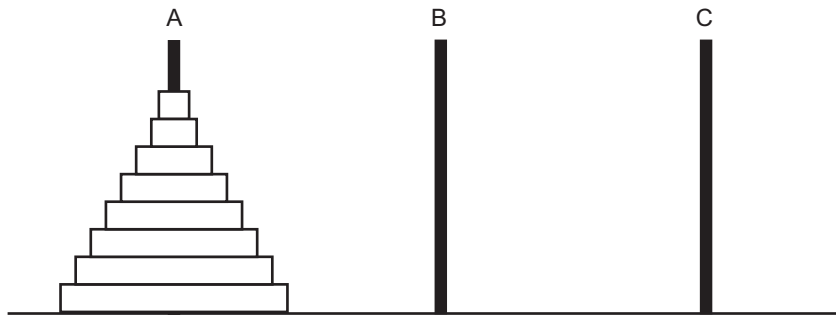


Abbildung 3.5: Türme von Hanoi

Die Mönche müssen zwei Regeln beachten:

1. Es darf nur eine Scheibe zurzeit bewegt werden.
2. Nie darf eine größere auf einer kleineren Scheibe zu liegen kommen.

Die Legende sagt, dass das Ende der Welt kommt, wenn die Mönche ihre Aufgabe beendet haben. Wie sieht ein Algorithmus aus, der den Mönchen sagt, welche Scheibe von welchem Stapel zu welchem Stapel bewegt werden soll, um die Aufgabe zu erfüllen? Ein einfacher Vorschlag:

1. Bringe 63 Scheiben von Stapel A nach Stapel C.
2. Bringe die unterste Scheibe von A nach Stapel B.
3. Bringe alle 63 Scheiben von Stapel C nach Stapel B – fertig!

Die Lösung ist sehr einfach, jedoch sagt sie nichts darüber, wie 63 Scheiben zu bewegen sind. Aber die Komplexität des Problems ist reduziert: Wenn wir wüssten, wie 63 Scheiben zu bewegen sind, wissen wir, wie alle 64 zu bewegen sind. Ein einfacher Vorschlag, 63 Scheiben von A nach C zu bringen:

1. Bringe 62 Scheiben von Stapel A nach Stapel B.
2. Bringe die unterste Scheibe von A nach Stapel C.

3. Bringe 62 Scheiben von Stapel B nach Stapel C.

Wir sehen ein allgemeines Muster, und auch, dass die Rollen von A, B, C gewechselt haben. Eine allgemeinere Formulierung für n Scheiben wäre:

1. Bringe $n - 1$ Scheiben vom Quell-Stapel zum Arbeits-Stapel.
2. Bringe die unterste Scheibe vom Quell-Stapel zum Ziel-Stapel.
3. Bringe $n - 1$ Scheiben vom Arbeits-Stapel zum Ziel-Stapel.

Dies ruft nach einer rekursiven Formulierung! Die Abbruchbedingung ist klar: Falls 0 Scheiben zu bewegen sind, tun wir nichts.

Listing 3.6: Beispielprogramm 2 mit Rekursion

```
// cppbuch/k3/hanoi.cpp
#include<iostream>
using namespace std;

void bewegen(int n, int quelle, int ziel, int zwischen) {
    if (n > 0) {
        // Abbruchbedingung: n == 0
        bewegen(n - 1, quelle, zwischen, ziel); // rekursiver Aufruf
        cout << "Bringe eine Scheibe von " << quelle
              << " nach " << ziel << endl;
        bewegen(n - 1, zwischen, ziel, quelle); // rekursiver Aufruf
    }
}

int main() {
    cout << "Türme von Hanoi! Anzahl der Scheiben: ";
    int scheiben;
    // besser nicht 64 eingeben, sondern eine kleinere Zahl,
    // zum Beispiel 4 (Begründung siehe unten).
    cin >> scheiben;
    bewegen(scheiben, 1, 2, 3);
}
```

Analyse des Algorithmus

Wie viele Bewegungen braucht es? Jeder Aufruf von `bewegen()` erzeugt zwei neue Aufrufe. Auf jedem Level n gibt es zwei Aufrufe des Levels $(n-1)$. Die Anweisung zwischen den Aufrufen, also die tatsächliche Bewegung, wird nur ausgeführt, wenn $n \geq 1$ ist. Also ist die Gesamtzahl der Bewegungen $N = 1 + 2 + 4 + 8 + 16 + \dots + 2^{n-1} = 2^n - 1$.

Wenn wir $n = 64$ und eine Sekunde pro Bewegung annehmen, erhalten wir eine *sehr* lange Zeitdauer, nämlich $N = 18.446.744.073.709.551.615$ Sekunden, also ungefähr $5.85 \cdot 10^{11}$ Jahre oder etwa das 50-fache des Alters unseres Universums. Selbst wenn die Legende stimmen sollte, dass nach der Erledigung der Aufgabe die Welt untergeht, bräuchten wir uns keine Sorgen zu machen!



Übungen

3.3 Schreiben Sie die Funktion zur Berechnung der Fakultät von Seite 102 als rekursive Funktion. Dabei gilt: $0! = 1$, $1! = 1$, $n! = n \cdot (n - 1)!$

3.4 Für Menschen mit Informatik-Vorkenntnissen: Ein rekursiver Aufruf *am Ende* einer Funktion, die keinen Wert liefert (sogenannte Restrekursion), kann stets durch Einführung einer Schleife in die Funktion beseitigt werden. Wie müsste die Funktion `bewegen()` im Beispiel oben umgebaut werden, damit nur der erste rekursive Aufruf übrig bleibt? Hinweis: Eine `while(n > 0)`-Schleife umschließt den ersten rekursiven Aufruf. Die Änderung von `n` und die Änderung der Reihenfolge der Parameter `a`, `b`, `c` ersetzen den zweiten Aufruf.

3.2.2 Übergabe per Referenz

Wenn ein übergebenes Objekt modifiziert werden soll, kann die Übergabe durch eine *Referenz* des Objekts geschehen. Die Syntax des Aufrufs ist die gleiche wie bei der Übergabe per Wert; anstatt mit einer Kopie wird jedoch *direkt mit dem Original* gearbeitet, wenn auch unter anderem Namen (vergleiche Seite 55). Der Name ist lokal bezüglich der Funktion, und er bezieht sich auf das übergebene Objekt. Es wird also keine Kopie angelegt. Daher ergibt sich bei großen Objekten ein Laufzeitvorteil. Innerhalb der Funktion vorgenommene Änderungen wirken sich direkt auf das Original aus.

Es wurde darauf hingewiesen, dass die Übergabe von *nicht zu verändernden* Objekten generell per Wert erfolgen soll mit der Ausnahme großer Objekte aus Effizienz- und Speicherplatzgründen. Wenn zwar der Laufzeitvorteil, aber keine Änderung des Originals erwünscht ist, kommt die Übergabe eines Objekts als *Referenz auf const* in Frage. Die Angabe in der Parameterliste könnte zum Beispiel `const TYP& unveraenderliches_grosses_Objekt` lauten. Innerhalb der Funktion darf auf das übergebene Objekt natürlich nur lesend zugegriffen werden; dies wird vom Compiler geprüft. Das Prinzip der Übergabe per Referenz zeigt folgendes Beispielsprogramm.

Abbildung 3.6 zeigt, dass dasselbe Objekt unter verschiedenen Namen vom aufrufenden Programm und von der Funktion zugreifbar ist.

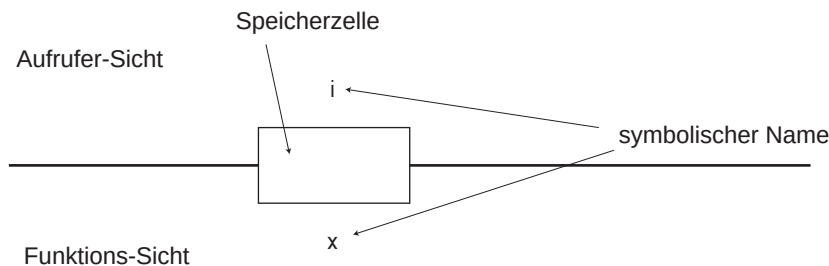


Abbildung 3.6: Parameterübergabe per Referenz (Bezug: Programmbeispiel)

Die Stellung des `&`-Zeichens in der Parameterliste ist beliebig. `(int& x)` ist genau so richtig wie `(int &x)` oder `(int & x)`. Bei der Diskussion über Laufzeitvorteile durch Referenzparameter darf nicht vergessen werden, dass es häufig Fälle gibt, in denen bewusst die Kopie eines Parameters *ohne* Auswirkung auf das Original geändert werden soll, sodass nur eine Übergabe per Wert in Frage kommt. Ein Beispiel ist der Parameter `z` der Funktion `qsum()` von Seite 109.

Listing 3.7: Übergabe per Referenz

```
// cppbuch/k3/per_ref.cpp
#include<iostream>
using namespace std;

void addiere_7(int&); // int& = Referenz auf int

int main() {
    int i = 0;
    cout << i << " = alter Wert von i\n";
    addiere_7(i);
    cout << i << " = neuer Wert von i nach addiere_7\n";
}

void addiere_7(int& x) {
    x += 7;          // Original des Aufrufers wird geändert!
}
```

3.2.3 Gefahren bei der Rückgabe von Referenzen

Bei der Rückgabe von Referenzen muss darauf geachtet werden, dass das zugehörige Objekt tatsächlich noch existiert (vgl. Kapitel 3.1.2). Das folgende Beispiel zeigt, wie man es *nicht* machen soll:

Negativ-Beispiel!

```
int& maxwert(int a, int b) { // Referenz ?
    // a und b sind lokale Kopien der übergebenen Daten!
    if(a > b) return a;      // Fehler!
    else      return b;      // Fehler!
}

int main() {
    int x = 17, y = 4;
    int& z = maxwert(x, y);
    cout << z << endl;      // z ist undefiniert
    int x1 = maxwert(y, x); // Anweisung enthält kein z!
    cout << z << endl;      // vermutlich anderer Wert!
}
```

Fehler! Begründung: Es wird eine Referenz auf eine *lokale* Variable zurückgegeben, die nicht mehr definiert ist und deren Speicherplatz früher oder später überschrieben wird. Korrekt wäre es, nicht die Referenz, sondern eine Kopie des Objekts zurückzugeben (Rückgabetypp `int` statt `int&`):

```
int maxwert(int a, int b) { ... }
```

Eine weitere Möglichkeit `int& maxwert(int& a, int& b) {...}` ist nicht empfehlenswert. Sie funktioniert zwar im obigen Programmbeispiel, erlaubt aber keine konstanten Argumente wie zum Beispiel in einem Aufruf `z = maxwert(23, y)`. Eine Konstante hat keine Adresse, weil der Compiler den Wert direkt in das Compilationsergebnis eintragen kann, ohne sich auf eine Speicherstelle zu beziehen.

3.2.4 Vorgegebene Parameterwerte und variable Parameterzahl

Funktionen können mit variabler Parameteranzahl aufgerufen werden. In der Deklaration des Prototypen werden für die nicht angegebenen Parameter *vorgegebene Werte* (englisch *default values*) spezifiziert. Der Vorteil liegt *nicht* in der ersparten Schreibarbeit, weil die Standardparameter nicht angegeben werden müssen! Eine Funktion kann um verschiedene Eigenschaften *erweitert* werden, die durch weitere Parameter nutzbar gemacht werden. Die Programme, die die alte Version der Funktion benutzen, sollen aber weiterhin wartbar und übersetzbar sein, ohne dass jeder Funktionsaufruf geändert werden muss. Nehmen wir an, dass ein Programm eine Funktion `adressenSortieren()` zum Beispiel aus einer firmenspezifischen Bibliothek benutzt. Die Funktion sortiert eine Adressendatei alphabetisch nach Nachnamen. Der Aufruf sei

```
// Aufruf im Programm1
adressenSortieren(adressdatei);
```

Die Sortierung nach Postleitzahlen und Telefonnummern wurde später benötigt und nachträglich eingebaut. Der Aufruf in einer neuen Anwendung könnte lauten:

```
// anderes, NEUES Programm2
enum Sortierkriterium {Nachname, PLZ, Telefon};
adressenSortieren(adressdatei, PLZ);
```

Das alte Programm1 soll ohne Änderung übersetzbar sein. Durch den Funktionsaufruf mit unterschiedlicher Parameterzahl ist dies möglich. Der Vorgabewert wäre hier `Nachname`. Die Parameter mit Vorgabewerten erscheinen in der Deklaration *nach* den anderen Parametern. Programmbeispiel:

Listing 3.8: Vorgegebene Parameter

```
// cppbuch/k3/preis.cpp
#include<iostream>
#include<string>
using namespace std;

// Funktionsprototyp 2. Parameter mit Vorgabewert:
void preisAnzeige(double preis,
                  const string& waehrung = "Euro");

// Hauptprogramm
int main() {
    // zwei Aufrufe mit unterschiedlicher Parameterzahl :
    preisAnzeige(12.35); // vorgegebener Parameter wird eingesetzt
    preisAnzeige(99.99, "US-Dollar");
}

// Funktionsimplementation
void preisAnzeige(double preis, const string& waehrung) {
    cout << preis << ' ' << waehrung << endl;
}
```

Ausgabe des Programms: 12.35 Euro und 99.99 US-Dollar

Falls der Preis in € angezeigt werden soll, braucht keine Währung genannt zu werden. Dies ist der Normalfall. Andernfalls ist die Währungsbezeichnung als Zeichenkette im zweiten Argument zu übergeben.

3.2.5 Überladen von Funktionen

Funktionen können überladen werden. Deswegen darf für gleichartige Operationen mit Daten verschiedenen Typs *derselbe Funktionsname* verwendet werden, obwohl es sich nicht um dieselben Funktionen handelt. Ein Programm wird dadurch besser lesbar. Die Entscheidung, welche Funktion von mehreren Funktionen gleichen Namens ausgewählt wird, hängt vom Kontext, also der Umgebungsinformation ab: Der Compiler trifft die richtige Zuordnung anhand der *Signatur* der Funktion, die er mit dem Aufruf vergleicht. Die Signatur besteht aus der Kombination des Funktionsnamens mit Reihenfolge und Typen der Parameter. Beispiel:

Listing 3.9: Überladen von Funktionen

```
// cppbuch/k3/ueberlad.cpp
#include<iostream>
using namespace std;

double maximum(double x, double y) {
    return x > y ? x : y; // Bedingungsoperator siehe Seite 67
}

// zweite Funktion gleichen Namens, aber unterschiedlicher Signatur
int maximum(int x, int y) {
    return x > y ? x : y;
}

int main() {
    double a = 100.2;
    double b = 333.777;
    int c = 1700;
    int d = 1000;
    cout << maximum(a,b) << endl; // Aufruf von maximum(double, double)
    cout << maximum(c,d) << endl; // Aufruf von maximum(int, int)
}
```

Der Compiler versucht, nach bestimmten Regeln immer die beste Übereinstimmung mit den Parametertypen zu finden:

```
const float E = 2.7182, PI = 3.14159;
cout << maximum(E, PI);
```

führt zum Aufruf von `maximum(double, double)`, und `maximum(31, 'A')` zum Aufruf von `maximum(int, int)`, weil `float`-Werte in `double`-Wert konvertiert und der Datentyp `char` auf `int` abgebildet wird. Dies gelingt nur bei einfachen und zueinander passenden Datentypen und eindeutigen Zuordnungen. Der Aufruf `maximum(3.1, 7)` ist nicht eindeutig interpretierbar. Das erste Argument spricht für `maximum(double, double)`, das zweite für `maximum(int, int)`. Der Compiler kann sich nicht entscheiden und erzeugt eine Fehler-

meldung. Es bleibt einem natürlich unbenommen, selbst eine Typumwandlung vorzunehmen. Die Aufrufe

```
cout << maximum(3.1, static_cast<float>(7));
cout << maximum(3.1, static_cast<double>(7));
int x = 66;
char y = static_cast<char>(x);
cout << maximum(static_cast<int>(0.1), static_cast<int>(y));
```

sind daher zulässig und unproblematisch, abgesehen vom Informationsverlust durch die Typumwandlung in der letzten Zeile. Die Umwandlung nach `int` schneidet die Nachkommaziffern ab. Der Typ `char` kann vorzeichenbehaftet (`signed`) sein. In diesem Fall ergibt die interne Umwandlung von `int` in `char` nur dann ein positives Ergebnis, wenn nach dem Abschneiden der höherwertigen Bits das Bit Nr. 7 nicht gesetzt ist, wobei die Zählung mit dem niedrigstwertigen Bit beginnt, das die Nr. 0 trägt:

```
// Voraussetzung: char ist signed char. Aufgerufen wird maximum(int, int).
cout << maximum(-1000, static_cast<char>(600)); // ergibt 88
cout << maximum(-1000, static_cast<char>(128)); // ergibt -128
cout << maximum(-1000, static_cast<char>(129)); // ergibt -127 usw.
```

Das Abschneiden der höherwertigen Bits wird deutlich, wenn man zum Beispiel 600 als $2^9 + 88$ schreibt. In den Abschnitten 4.3.4 und 9.4 werden wir eine Möglichkeit zur benutzerspezifischen Typumwandlung für beliebige Datentypen kennenlernen.

Gemäß der Regel, dass ein C++-Name, gleichgültig ob Funktions- oder Variablenname, alle gleichen Namen eines äußeren Gültigkeitsbereichs überdeckt, funktioniert das oben beschriebene Überladen nur innerhalb *desselben* Gültigkeitsbereichs. Ein Test:

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

void f(char c) {
    cout << "f(char) c=" << c << endl;
}

void f(double x) {
    cout << "f(double) x=" << x << endl;
}

int main() { // neuer Gültigkeitsbereich (Block) beginnt
    void f(double); // ***
    f('a');
}
```

Die Deklaration innerhalb eines anderen Gültigkeitsbereichs führt dazu, dass `f` mit dem `char`-Parameter nicht mehr sichtbar ist. Es wird `f(double)` ausgeführt, wobei das Zeichen 'a' in eine `double`-Zahl umgewandelt wird. Machen Sie die Gegenprobe, indem Sie die ***-Zeile löschen! Der Compiler findet sich dann wieder zurecht, und `f(char)` wird ausgeführt.

3.2.6 Funktion main()

`main()` ist eine spezielle Funktion. Jedes C++-Programm startet definitionsgemäß mit `main()`, sodass `main()` in jedem C++-Programm genau einmal vorhanden sein muss. Die

Funktion ist nicht vom Compiler vordefiniert, ihr Rückgabetyt soll `int` sein und ist ansonsten aber implementationsabhängig. `main()` kann nicht überladen oder von einer anderen Funktion aufgerufen werden. Die zwei folgenden Varianten sind mindestens gefordert und werden daher von jedem Compilerhersteller zur Verfügung gestellt:

```
// erste Variante
int main() {
    ...
    return 0;    // Exit-Code
}

// zweite Variante
int main( int argc, char* argv[]) { // siehe Text
    ...
    return 0;    // Exit-Code
}
```

Die zweite Variante verwendet *Zeiger* (`char*`) und C-Arrays, die in Kapitel 5 besprochen werden. Die Auswertung der Argumente wird bis dahin zurückgestellt (ab Seite 207).

Es bleibt dem Hersteller eines Compilers überlassen, ob er weitere Versionen mit erweiterten Argumentlisten anbietet. Die mit `return` zurückgegebene Zahl wird an die aufrufende Umgebung des Programms übergeben. Damit kann bei einer Abfolge von Programmen ein Programm den Rückgabewert des Vorgängers abfragen, zum Beispiel zur gezielten Reaktion auf Fehler. Wenn irgendwo im Programm die im Header `<cstdlib>` deklarierte Funktion `void exit(int)` aufgerufen wird, ist die Wirkung dieselbe, wobei jedoch der aktuelle Block verlassen wird, ohne automatische Objekte (Stackvariable) freizugeben. Der Argumentwert von `exit()` ist dann der Rückgabewert des Programms. `return` darf in `main()` weggelassen werden; dann wird automatisch 0 zurückgegeben.

3.2.7 Beispiel Taschenrechnersimulation

Um ein etwas umfangreicheres Beispiel mit Funktionen zu geben, wird ein Taschenrechner simuliert, eine beliebige Aufgabe (siehe auch [Mar], nach dem dieses Beispiel entworfen wurde, oder etwas komfortabler und aufwendiger [Str, Kapitel 6.1]). Die hier verwendete und nur kurz beschriebene Methode des *rekursiven Abstiegs* ermöglicht es, auf elegante und einfache Art beliebig verschachtelte Ausdrücke auszuwerten. In [ALSU] können fortgeschrittene Interessierte ausführliche Erläuterungen der Methode finden.

Syntax eines mathematischen Ausdrucks

Zunächst sei die Syntax eines mathematischen Ausdrucks wie zum Beispiel $(13 + 7) * 5 - (2 * 3 + 7) / (-8)$ beschrieben, wobei der Schrägstrich das Zeichen für die ganzzahlige Division sein soll. Ein *Ausdruck* wird als *Summand* oder *Summe von Summanden* aufgefasst, die sich ihrerseits aus *Faktoren* zusammensetzen. Durch die zuerst auszuführende Berechnung der Faktoren ist die Prioritätsreihenfolge »Punktrechnung vor Strichrechnung« gewährleistet. Ein *Faktor* kann eine *Zahl* oder ein *Ausdruck in Klammern* sein. Die Verschachtelung mit Klammern sei beliebig möglich. Eine *Zahl* besteht aus einer oder mehreren Ziffern. Eine Ziffer ist eines der Zeichen 0 bis 9. Zur Vereinfachung sei ein mathematischer Ausdruck auf ganze Zahlen und die vier Grundrechenarten beschränkt.

Leerzeichen sind im Ausdruck nicht erlaubt. Abbildung 3.7 zeigt die Syntax eines Ausdrucks.

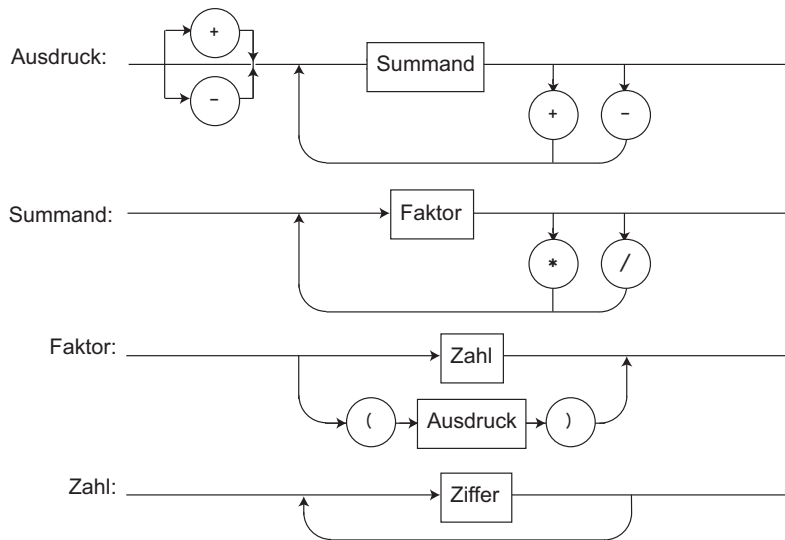


Abbildung 3.7: Syntaxdiagramm für einen mathematischen Ausdruck

Aus dem Syntaxdiagramm wird die indirekte Rekursion deutlich: Ausdruck ruft Summand, Summand ruft Faktor, Faktor ruft Ausdruck etc. Da jeder arithmetische Ausdruck endlich ist, endet die Rekursion irgendwann. Die Auflösung eines Ausdrucks bis zum Rekursionsende nennt man *rekursiver Abstieg*. Abbildung 3.8 zeigt den Ableitungsbaum des Ausdrucks $(12 + 3) * 4$, in dem die äußeren Elemente (die »Blätter« des »Baums«) die Zahl- oder Operatorzeichen sind. Die inneren Elemente, durch Kästen dargestellt, sind noch aufzulösen.

Abbildung 3.8 ist wie folgt zu interpretieren: Der *Ausdruck* ist ein *Summand*, nämlich $(12 + 3) * 4$, bestehend aus dem *Faktor* $(12 + 3)$, dem Multiplikationszeichen $*$ und dem *Faktor* 4. Die Faktoren werden dem Syntaxdiagramm entsprechend weiter ausgewertet. Der erste Faktor zum Beispiel ist ein durch runde Klammern $()$ begrenzter *Ausdruck* usw. Wir gehen so vor, dass wir das obige Syntaxdiagramm 3.7 direkt in ein Programm transformieren. Rekursive Syntaxstrukturen werden dabei auf rekursive Strukturen im Programm abgebildet. Ziel:

- Berechnung beliebig verschachtelter arithmetischer Ausdrücke, wobei hier zur Vereinfachung nur ganze Zahlen zugelassen sein sollen.
- Leerzeichen sind nicht erlaubt; keine aufwendige Syntaxprüfung
- Vorrangregeln sollen beachtet werden.

Wie kann man nun ein Programm schreiben, das die gewünschte Berechnung liefert? Zunächst ein paar Vorgaben:

- a) Das Programm soll ein Promptzeichen $>>$ ausgeben und dann die Eingabe des Ausdrucks erwarten.

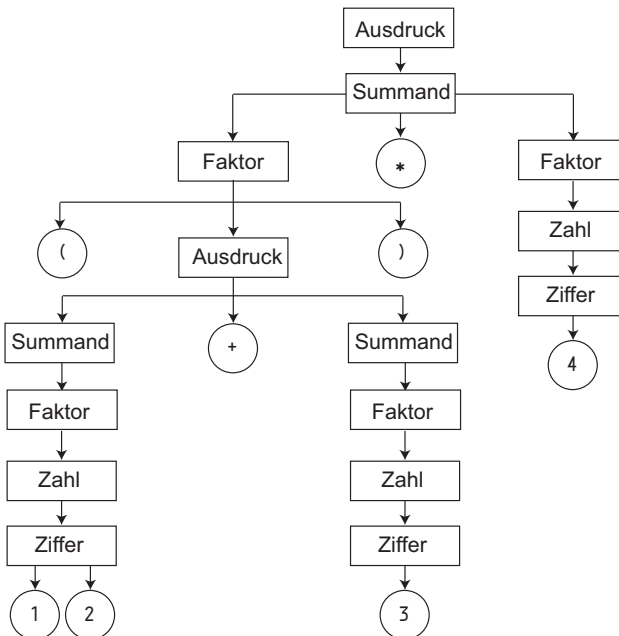


Abbildung 3.8: Ableitungsbaum für $(12+3)*4$

- b) Der Ausdruck wird mit `ENTER` abgeschlossen. Anschließend wird das Ergebnis ausgegeben.
- c) a) und b) sollen wiederholt werden, bis 'e' als Endekennung eingegeben wird. Damit kann das Hauptprogramm geschrieben werden:

```

int main() {
    char ch;
    do {
        cout << "\n>>";
        cin.get(ch);
        if (ch != 'e') {
            cout << ausdruck(ch);
        }
    } while(ch != 'e');
}
  
```

`cin.get(ch)` ist eine vordefinierte Prozedur, die das nächste Zeichen aus dem Tastaturpuffer, in den das Betriebssystem die eingegebenen Zeichen der Reihe nach abgelegt hat, einliest, wie auf Seite 94 beschrieben. Mit jedem weiteren Aufruf von `cin.get()` wird ein weiteres Zeichen geholt. `cin >> ch` wird nicht gewählt, weil `ENTER` dann ignoriert wird. Nachdem der Rahmen abgesteckt ist, geht es nun an den Kern des Problems: `ausdruck()` ist offensichtlich eine Funktion, die die eingegebene Zeichenkette auswertet und einen `int`-Wert, nämlich das Ergebnis, zurückgibt. Wir haben es uns einfach gemacht und die ganze Arbeit an die Funktion delegiert. Wie kann die Funktion `ausdruck()` aussehen?

Dazu ein paar Vorüberlegungen: Laut Syntaxdiagramm ist *Ausdruck* entweder

- a) *-Summand*
- b) *+Summand* oder einfach nur
- c) *Summand*

sowie mögliche zusätzliche, durch + oder - getrennte weitere Summanden. Man kann also die Zeichen + oder - gegebenenfalls überlesen und dann `ausdruck()` den Wert einer Funktion `summand()` zuweisen, die den Rest der Zeichenkette auswertet und ein `int`-Ergebnis zurückgibt. Das ermöglicht es `ausdruck()`, seinerseits einen Teil der Arbeit an `summand()` zu delegieren. Einer schiebt es auf den anderen, wie im richtigen Leben! Daraus ergibt sich die Vorgehensweise:

- Aus dem Syntaxdiagramm leitet sich die folgende syntaktische Konstruktion ab, wobei das aktuelle Zeichen überlesen wird, wenn es *nicht* zu dieser Konstruktion gehört. *Andernfalls* ist das Zeichen das erste zu analysierende Zeichen der syntaktischen Folgekonstruktion und wird der zugehörigen Funktion übergeben.
- Die Folgekonstruktion wird als Funktion aufgerufen und verhält sich wie der Aufrufer. Wenn die Funktion auf ein Zeichen stößt, das *nicht* zu der zugehörigen syntaktischen Konstruktion passt, wird es an den Aufrufer *zurückgegeben*.

Beispiel :

Ausdruck: Aus dem Syntaxdiagramm ergibt sich *Summand* als folgende syntaktische Konstruktion. '-' oder '+' müssen gegebenenfalls übersprungen werden, weil sie kein Element von *Summand* sind.

Summand wird anschließend genauso behandelt wie *Ausdruck* usw. Die Rekursion muss wegen der endlichen Länge eines Ausdrucks irgendwann ein Ende haben.

Nach diesen Vorbemerkungen bilden wir das Syntaxdiagramm direkt auf ein C++-Programm ab, wobei dem syntaktischen Term *Ausdruck* eine Funktion mit dem Namen `ausdruck()` zugeordnet wird. Eine Schleife wird im Diagramm in eine `while()`-Anweisung transformiert. Die Entsprechung zwischen dem Syntaxdiagramm auf Seite 117 und dem Programmcode ist offensichtlich. Die Variable `c` wird als Referenz übergeben, damit bei Ende der Funktion der neue Wert der aufrufenden Funktion zur weiteren Analyse zur Verfügung steht.

```
long ausdruck(char& c) {           // Übergabe per Referenz!
    long a;                       // Hilfsvariable für Ausdruck
    if(c == '-') {
        cin.get(c);               // - im Eingabestrom überspringen
        a = -summand(c);          // Rest an summand() übergeben
    }
    else {
        if(c == '+')
            cin.get(c);            // + überspringen
        a = summand(c);
    }
    while(c == '+' || c == '-')
        if(c == '+') {
            cin.get(c);            // + überspringen
            a += summand(c);
        }
}
```

```

        else {
            cin.get(c);          // - überspringen
            a -= summand(c);
        }
    return a;
}

```

Summand wird auf die gleiche Art wie `ausdruck()` gebildet:

```

long summand(char& c) {
    long s = faktor(c);
    while(c == '*' || c == '/')
        if(c == '*') {
            cin.get(c);          // * überspringen
            s *= faktor(c);
        }
        else {
            cin.get(c);          // / überspringen
            s /= faktor(c);
        }
    return s;
}

```

Auch Faktor wird auf ähnliche Art konstruiert:

```

long faktor(char& c) {
    long f;
    if(c == '(') {
        cin.get(c);              // ( überspringen
        f = ausdruck(c);
        if(c != ')')
            cout << "Rechte Klammer fehlt!\n"; // *** siehe Text unten
        else cin.get(c);         // ) überspringen
    }
    else
        f = zahl(c);
    return f;
}

```

Nun bleibt nur noch die Funktion zur Analyse einer Ziffernfolge:

```

long zahl(char& c) {
    long z = 0;
    // isdigit() ist eine Funktion (genauer: ein Makro), das zu true ausgewertet wird,
    // falls c ein Zifferzeichen ist. Die Verwendung setzt #include<cctype> voraus.
    while(isdigit(c)) { // d.h. c >= '0' && c <= '9'
        // Zur Subtraktion von '0' siehe Seite 54.
        z = 10*z + long(c-'0'); // implizite Typumwandlung
        cin.get(c);
    }
    return z;
}

```

Letztlich ist die Umsetzung einer Syntax in ein Programm reine Fleißarbeit, wenn man weiß, wie es geht. Deswegen gibt es dafür Werkzeuge wie die Programme *lex* und *yacc* oder *bison*. Nun haben wir alle Bausteine zusammen, die zur Auswertung eines beliebig verschachtelten arithmetischen Ausdrucks nötig sind. Es bleibt dem Leser überlassen, das Programm zu vervollständigen, einschließlich Trennung von Prototypen und Definitionen, und es zum Laufen zu bringen. Erweiterungen können leicht eingebaut werden, um Leerzeichen an syntaktisch sinnvollen Stellen zu erlauben oder Hinweise auf Syntaxfehler auszugeben, wie in der mit ******* markierten Zeile gezeigt wird. Falls doch noch Verständnisschwierigkeiten auftreten sollten, spielt man am besten selbst »Computer«, indem man einen Ausdruck Schritt für Schritt am Schreibtisch dem Programm folgend abarbeitet.

3.2.8 Spezifikation von Funktionen

Eine Funktion erledigt eine Teilaufgabe. Es ist sinnvoll, diese Teilaufgabe im Funktionskopf als Kommentar zu spezifizieren. Dazu gehören Annahmen über die Importschnittstelle (Eingabedaten, zum Beispiel Wertebereich), die Fehlerbedingungen, die Exportschnittstelle (Ausgabedaten). Die Bedingung, die ein Eingabeparameter erfüllen muss, damit die Funktion richtig arbeitet, nennt man Vorbedingung. Der Zustand eines Programms nach Abarbeitung der Funktion wird Nachbedingung genannt. Die Spezifikation ist für den Benutzer einer Funktion von Interesse.

Wie die Aufgabe gelöst wird, sollte im Funktionskopf nicht beschrieben werden, um die Möglichkeit einer späteren Änderung der Implementierung nicht einzuschränken, zum Beispiel einen langsamen durch einen schnelleren Algorithmus zu ersetzen. Das schließt nicht aus, dass innerhalb der Funktion manche Stellen kommentierend erklärt werden. Die Interna einer Funktion sind nur für Entwickler von Interesse, nicht aber für den Benutzer.

Eine Spezifikation kann als *Vertrag* zwischen Aufrufer und Funktion aufgefasst werden. Die Funktion gewährleistet die Nachbedingung, wenn der Aufrufer die Vorbedingung einhält. Die Analogie zu einem Vertrag zwischen Kunde und Softwarehaus liegt auf der Hand. Eine Vertiefung des Themas »Design by Contract« ist in [Mey] zu finden.

Die Spezifikation sollte mit in eine Header-Datei übernommen werden. Eine Header-Datei soll unter anderem die Prototypen von Funktionen enthalten (siehe folgender Abschnitt 3.3). Mehr zur Dokumentation von Programmen erfahren Sie in Abschnitt 19.1.



Übungen

3.5 Schreiben Sie eine Funktion `void str_umkehr(string& s)`, die die Reihenfolge der Zeichen im String `s` umkehrt.

3.6 Vervollständigen Sie das Beispiel in Abschnitt 3.2.7 und bringen Sie es zum Laufen.

3.7 Schreiben Sie eine Funktion `istAlphanumerisch(const string& text)`, die `true` zurückgibt, wenn `text` nur Buchstaben und Ziffern enthält, andernfalls `false`.

3.3 Modulare Programmgestaltung

C++ bietet eine große Flexibilität in der Organisierung eines Softwaresystems. Die Erfahrung lehrt, dass die Aufteilung eines großen Programms in einzelne, getrennt übersetzbare Dateien, die zusammengehörige Programmteile enthalten, sinnvoll ist. Folgender Aufbau empfiehlt sich:

- Die Standard-Header haben die uns schon bekannte Form `<headername>`. Darüber hinaus kann es eigene (oder andere) Header-Dateien geben, die typischerweise die Endung `*.h` [oder auch `*.hpp`, `*.hxx`, je nach Computer- oder Entwicklungssystem] im Dateinamen haben. Sie enthalten Konstanten, Schnittstellenbeschreibungen wie Klassendeklarationen, Deklarationen globaler Daten und Funktionsprototypen.
- Implementationsdateien enthalten die Implementation der Klassen und den Programmcode der Funktionen (Endung im Dateinamen: `*.cpp` [auch `*.cxx`, `*.cc`, `*.c`]).
- Main-Datei. Sie enthält das Hauptprogramm `main()`.

Wirkung von `#include`

Damit eine Datei einzeln für sich übersetzbar ist, müssen Konstanten, Klasseninterfaces und Funktionsprototypen bekannt sein. Das wird erreicht durch das Einschließen der Header-Dateien mit der Präprozessordirektive `#include "filename.h"`. Präprozessordirektiven werden von einem dem eigentlichen Compiler vorgeschalteten Präprozessor verarbeitet, der auch die Kommentare ausblendet.

Anstelle von `filename.h` ist natürlich der richtige Name einzutragen. Die Datei `filename.h` wird im aktuellen Verzeichnis gesucht und an dieser Stelle eingelesen. Die eingelesene Datei kann selbst auch `#include`-Direktiven enthalten, die genauso verarbeitet werden. Weiteres zu diesen Direktiven, insbesondere auch zur Form `#include<Header>` (keine Anführungszeichen als Begrenzer), ist auf Seite 128 zu finden.

Zwei Strukturen, die in den nächsten Abschnitten behandelt werden, sind möglich:

- die Steuerung der Übersetzung nur durch `#include`-Anweisungen;
- das Einbinden von bereits vorübersetzten Programmteilen; besonders sinnvoll bei großen Programmen, von denen einige Teile schon stabil laufen.

3.3.1 Steuerung der Übersetzung nur mit `#include`

Nehmen wir an, dass das `main`-Programm (Datei `meinprog.cpp`) die Funktionen `func_a1()` und `func_a2()` aus der Datei `a.cpp` und eine Funktion `func_b()` aus der Datei `b.cpp` benutzt. Mit `#include` werden diese Dateien in `meinprog.cpp` eingeschlossen. Nur bei sehr kleinen Programmen ist dieses Verfahren ausreichend. Im Normalfall gelten jedoch die Empfehlungen des folgenden Abschnitts. `#include "a.cpp"` wirkt, als ob an der Stelle der `#include`-Anweisung die Datei `a.cpp` selbst hingeschrieben worden wäre:

```
// nicht empfehlenswert! (»quick and dirty«)
#include "a.cpp"
#include "b.cpp"
int main() {
    func_a1(); // Funktionsaufrufe
```

```
func_a2( );  
func_b( );  
}
```

3.3.2 Einbinden vorübersetzter Programmteile

Bei größeren und sehr großen Programmen ist es sinnvoll, Schnittstellen (Funktionsprototypen und Klassen) und Implementationen (Programmcode) zu trennen. Daher nehmen wir ferner an, dass die Schnittstellen in den Header-Dateien *a.h* und *b.h* abgelegt sind.

Um die automatische Prüfung der Schnittstellen durch den Compiler zu ermöglichen, werden die Header-Dateien mit `#include` in allen Dateien eingeschlossen, die diese Schnittstellen verwenden. Mit den Header-Dateien kann jede Datei einzeln übersetzt werden. Wenn es Änderungen gibt, müssen nur noch die davon betroffenen Dateien neu compiliert werden. Die Dateien könnten folgenden Inhalt haben:

```
// a.h  
void func_a1();  
void func_a2();
```

```
// a.cpp  
#include "a.h"  
void func_a1() {  
    // Programmcode zu func_a1  
}  
  
void func_a2() {  
    // Programmcode zu func_a2  
}
```

```
// b.h  
void func_b();
```

```
// b.cpp  
#include "b.h"  
void func_b() {  
    // Programmcode zu func_b  
}
```

In diesem sehr einfachen Beispiel ist es nicht zwingend, *a.h* in *a.cpp* und *b.h* in *b.cpp* einzubinden, weil die **.cpp*-Dateien keine Informationen verwenden, die nur in den **.h*-Dateien vorkommen. Das ist jedoch nicht die Regel, wie wir später sehen werden.

```
// meinprog.cpp  
#include "a.h"  
#include "b.h"  
  
int main() {  
    func_a1( );  
    func_a2( );  
    func_b( );  
}
```


Die erste Zeile gibt jeweils den Namen der Datei im Kommentar an. Wir nehmen an, dass *a.cpp* und *b.cpp* bereits übersetzt sind, die Dateien *a.o* und *b.o* also existieren. In *meinprog.cpp* sei eine Änderung notwendig gewesen. Den Übersetzungsablauf zeigt Abbildung 3.9. Der *lib*-Anteil unten links in der Abbildung enthält die benötigten Systemfunktionen.

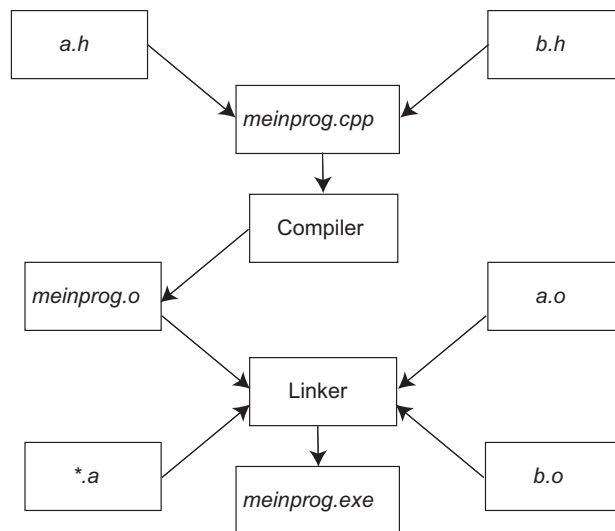


Abbildung 3.9: Compilations- und Link-Ablauf

Die Steuerung der Übersetzung und des Bindens ist je nach System unterschiedlich. Üblich sind Make-Dateien, auch Makefiles genannt, in denen die Reihenfolge und die Abhängigkeiten der Dateien beschrieben sind, sodass bei Änderungen nur die davon betroffenen neu übersetzt werden müssen. Make-Dateien werden in Kapitel 17 beschrieben. Eine andere Methode mit gleicher Wirkung sind sogenannte »Projekte«, in denen die zu übersetzenden und zu bindenden Dateien angegeben werden. Wenn eine ganze Reihe gut getesteter Programmbausteine zu einem Thema vorliegen, können die zugehörigen **.o*- (oder **.obj*)-Dateien in einer Bibliotheks- oder **.a*- (oder **.lib*)-Datei zusammengefasst werden. Die Konzepte

- Trennung von Schnittstellen und Implementation und
 - Gruppierung zusammengehöriger Funktionen und Klassen zu Bibliotheksmodulen
- sind Standard in allen größeren Programmierprojekten.

3.3.3 Dateiübergreifende Gültigkeit und Sichtbarkeit

Die Speicherklasse einer Variablen wird unter anderem durch die Worte *static*, *extern*, und *mutable* bestimmt. Mit *static* und *extern* werden Sichtbarkeit und Lebensdauer von Variablen eingestellt. *mutable* kann erst in Abschnitt 4.5 erläutert werden.

Alle nicht globalen und nicht-*static*-Variablen sind sogenannte *automatische* Variablen. Automatische Variablen werden bei Betreten eines Blocks mit undefiniertem Inhalt an-

gelegt, sofern sie nicht explizit initialisiert werden. Sie haben dann also *nicht* den Wert 0. Bei Verlassen des Blocks werden sie wieder zerstört (siehe Abschnitt 1.7).

extern bei Variablen

Variablen, die außerhalb von `main()` und jeglicher anderer Funktion definiert sind, heißen *global*. Sie sind in *allen* Teilen eines Programms gültig, auch in anderen Dateien. Eine globale Variable muss nur in einer anderen Datei als `extern` deklariert werden, um dort benutzbar zu sein.

```
// datei1.cpp
int global;           // Deklaration und Definition
int main() {
    global = 17;
}
```

```
// datei2.cpp
extern int global;    // Deklaration, aber keine Definition
void func1( ) {
    global = 123;
}
```

`datei2.cpp` ist für sich allein übersetzbar. Das Schlüsselwort `extern` sagt dem Compiler, dass eine Variable irgendwo anders definiert ist. Erst beim Binden, auch Linken genannt, wird die Referenz aufgelöst.



Tipp

Globale Variablen und Objekte sollen vermieden werden, weil sie für alle zugreifbar sind. Ursachen von mit ihnen verbundenen Fehlern sind daher schwer lokalisierbar.

static

Um den Gültigkeitsbereich von Variablen und Funktionen auf *eine Datei zu beschränken*, wird manchmal das Schlüsselwort `static` eingesetzt. Diese Bedeutung von `static` ist nicht mit der aus Abschnitt 3.1.2 zu verwechseln. In C++ kommt es mehrfach vor, dass Schlüsselwörter oder Operatoren mehrere Bedeutungen haben, die sich im konkreten Fall aus dem Kontext ergeben. Wegen der mehrfachen Bedeutung von `static` kann es aber zu Missverständnissen kommen, sodass die Verwendung für dateiglobale Variablen nicht empfohlen wird.

`static`-Variablen werden stets mit 0 initialisiert, wobei 0 gegebenenfalls in den passenden Datentyp umgewandelt wird. `static` dient zur Vermeidung von Namenskonflikten zwischen verschiedenen Dateien. Der gewünschte Zweck, die Gültigkeit einer Variable auf eine Datei zu beschränken, lässt sich ohne die Verwendung von `static` mit einem anonymen Namespace erreichen:

```
namespace {           // anonymer Namespace
    int global;
}
```

```
int main() {
    global = 17;
}
```

Anonyme Namespaces in verschiedenen Übersetzungseinheiten sind verschieden und nicht von außen zugreifbar. *Innerhalb* einer Übersetzungseinheit ist ein anonymer Namespace jedoch bekannt. Die Wirkung ist, als ob

```
namespace XXXX {
    int global;
}

using namespace XXXX;

int main() {
    global = 17;
}
```

geschrieben worden wäre, wobei XXXX irgendein beliebiger Name ist, der sonst nirgendwo in demselben Sichtbarkeitsbereich vorkommt. Mit dieser Änderung in *datei1.cpp* würden beide Dateien anstandslos übersetzt, aber es würde einen Linker-Fehler bei *Datei2.o* geben, weil jetzt die Gültigkeit von `global` nur auf *datei1.cpp* beschränkt ist. Alles, was der Compiler in einem Durchgang liest, ist eine *Übersetzungseinheit*. Man sagt, dass die nur innerhalb einer Übersetzungseinheit gültigen Variablen und Funktionen *intern gebunden* werden (internes Linken (englisch *internal linkage*)), während globale Variablen und Funktionen *extern gebunden* werden (externes Linken (englisch *external linkage*)).

extern bei Konstanten

Auf Dateiebene (außerhalb von `main()`) definierte *Variablen* sind *global* und in anderen Dateien benutzbar, wenn sie dort als *extern* deklariert sind. Bei *Konstanten* (`const`) ist es jedoch anders: Konstanten sind nur in der Definitionsdatei sichtbar! Sollen Konstanten anderen Dateien zugänglich gemacht werden, müssen sie als *extern* deklariert und initialisiert werden:

```
// datei1.cpp
extern const float PI = 3.14159; // Deklaration und Definition

// datei2.cpp
// Deklaration ohne Definition
extern const float PI; // ohne Initialisierung
```

Ohne *extern* in *datei1.cpp* wäre der Geltungsbereich von `PI` auf *datei1.cpp* beschränkt.

3.3.4 Übersetzungseinheit, Deklaration, Definition

Der Text, den der Compiler in einem Durchgang verdauen muss, heißt *Übersetzungseinheit*. In diesem Sinn geht es bei der Steuerung der Übersetzung mit `#include` in Abschnitt 3.3.1 nur um eine einzige Übersetzungseinheit. Große Programme werden jedoch in viele Übersetzungseinheiten gegliedert, um sie handhabbar zu machen. Insbesondere müssen bereits übersetzte und funktionstüchtige Teile nicht immer wieder neu übersetzt werden

(Abschnitt 3.3.2). Zum Verständnis ist es wichtig, klar zwischen den Begriffen *Deklaration* und *Definition* zu unterscheiden:

- Eine *Deklaration* führt einen Namen in ein Programm ein und gibt dem Namen eine Bedeutung.
- Eine Deklaration ist auch eine *Definition*, wenn *mehr als nur der Name eingeführt wird*, zum Beispiel wenn Speicherplatz für Daten oder Code angelegt oder die innere Struktur eines Datentyps beschrieben wird, aus der sich der benötigte Speicherplatz ergibt.

Der Verdeutlichung dienen einige Beispiele. Folgende Deklarationen sind gleichzeitig Definitionen:

```
int a; // Speicherplatz für a wird angelegt
extern const float PI = 3.14159; // Speicherplatz für PI wird angelegt
int f(int x) { return x*x; } // enthält Programmcode
struct meinStrukt { // definiert meinStrukt, d.h.
    int c; // beschreibt die innere Struktur
    int d; // beschreibt die innere Struktur
};
meinStrukt X; // Speicherplatz für X wird angelegt
enum meinEnum { li, re }; // definiert meinEnum
meinEnum Y; // Speicherplatz für Y wird angelegt
```

Die folgenden Zeilen sind Deklarationen, aber *keine* Definitionen:

```
extern int a;
extern const float PI;
int f(int);
struct meinStrukt;
enum meinEnum;
```

One Definition Rule

Die unter dem englischen Namen *one definition rule* bekannte Regel ist bei der Strukturierung von Programmen zu beachten: Jede Variable, Funktion, Struktur, Konstante und so weiter in einem Programm hat *genau eine* Definition. Dabei spielt es keine Rolle, ob das Programm aus vielen oder wenigen Übersetzungseinheiten besteht, ob die Definition selbst geschrieben wurde oder von einer Programmbibliothek (englisch *library*) zur Verfügung gestellt wird. Aus der *one definition rule* ergibt sich, was in den verschiedenen Dateitypen enthalten sein sollte (mit Beispielen):

Header-Dateien (*.h)

- Funktionsprototypen (Schnittstellen)

```
void meineFunktion(int einParameter);
```

- reine Deklaration (nicht Definition) globaler Variablen

```
extern int global;
```

- Deklaration globaler Konstanten (nicht Definition, das heißt ohne Initialisierung)

```
extern const int GLOBALE_KONSTANTE;
```

- Definition von Konstanten, die nur in der Übersetzungseinheit sichtbar sind

```
const int MAXI = 10;
```

- Definition von Datentypen wie `enum` oder `struct` (weil die `*.cpp`-Dateien die Größe von Objekten dieser Datentypen kennen müssen)

```
struct Punkt {
    int x;
    int y;
};
enum Wochenende {Samstag, Sonntag};
```

Implementationsdateien (*.cpp)

- Funktionsdefinitionen (Implementation)

```
void meineFunktion(int Parameter) {
    // ... Programmcode
}
```

- Definition globaler Objekte (nur *einmal* im ganzen Programm)

```
int global;
Punkt einPunkt;
Wochenende einWochenende;
```

- Definition und Initialisierung globaler Konstanten (nur *einmal* im Programm)

```
extern const int GLOBALE_KONSTANTE = 1;
```

Variablen, die ohne das Schlüsselwort `extern` in der Header-Datei auftreten, sind global. Wenn dieselbe Header-Datei von mehreren Implementationsdateien eingebunden wird, werden diese Variablen *mehrfach* angelegt – im Widerspruch zur »one definition rule«. Der Linker kann diese mehrfach angelegten Variablen gleichen Namens stillschweigend zusammenlegen oder er gibt eine Warnung oder Fehlermeldung aus, dass die Variable doppelt oder mehrfach definiert ist. Globale Variablen sollten also immer extern deklariert werden, und die Definition sollte nur in einer Übersetzungseinheit vorkommen.

Konstanten, die ohne das Schlüsselwort `extern` in der Header-Datei auftreten, sind nicht global und beziehen sich nur auf die Übersetzungseinheit. Wenn dieselbe Header-Datei von mehreren Implementationsdateien eingebunden wird, werden diese Konstanten entsprechend mehrfach angelegt. Falls der Compiler die Konstanten in besonderen Speicherplätzen ablegt (was durchaus nicht sein muss), bedeutet das Mehrfachanlegen zugleich Speicherplatzverschwendung.

3.3.5 Compilerdirektiven und Makros

Compilerdirektiven sind Anweisungen an den dem Compiler vorgeschalteten Präprozessor, die den Übersetzungsprozeß steuern, wie zum Beispiel `#include`. Compilerdirektiven beginnen stets mit `#` am Zeilenanfang.

#include

Bereits bekannt ist die `#include`-Anweisung (siehe Seiten 32 und 122). Die Dateispezifikation kann außer dem Dateinamen den vollständigen Pfad enthalten, wobei Verzeichnisnamen durch einen Schrägstrich `/` zu trennen sind. In der MS-Windows-Welt ist auch der

\ (Backslash) möglich, der Schrägstrich ist aber aus Portabilitätsgründen zu bevorzugen. Beispiele:

```
// relativer Pfad
#include "dateiname.h"
#include "../include/dateiname.h" // .. kennzeichnet das übergeordnete Verzeichnis
```

```
// absoluter Pfad
#include "/home/users/breymann/cppbuch/include/dateiname.h" // Unix
#include "C:/cppbuch/include/dateiname.h" // Windows
```

Wird die Datei im aktuellen Verzeichnis nicht gefunden, wird in den voreingestellten *include*-Verzeichnissen gesucht. Falls auch diese Suche fehlschlägt, wird versucht, die Direktive in der Standard-Header-Form zu interpretieren.

Die Standard-Header-Form ist `#include<header>`. Der Platzhalter *header* muss nicht unbedingt eine Datei sein [ISOC++]. Die bisher gängigen Implementierungen fassen *header* jedoch als Datei auf, und es wird in den voreingestellten *include*-Verzeichnissen gesucht, die Suche im aktuellen Verzeichnis entfällt. Die voreingestellten *include*-Verzeichnisse sind die zum System gehörenden *include*-Verzeichnisse, in denen zum Beispiel mit `#include<iostream>` alles Nötige zur Ein- und Ausgabe gefunden wird. Sie können aber auch eigene *include*-Verzeichnisse als voreingestellte definieren. Wenn Sie zum Beispiel das Programm *merge1.cpp* von Seite 673 mit

```
g++ merge1.cpp
```

im aktuellen Verzeichnis compilieren wollen, erhalten Sie eine Fehlermeldung, weil ein im Programm geforderter Header nicht gefunden wird. Bei Voreinstellung des *include*-Verzeichnisses mit der *I*-Option des Compilers verschwindet der Fehler:

```
g++ -I../../include merge1.cpp
```

#define, #ifdef, #ifndef

Es kann zu Problemen beim Übersetzen führen, wenn Header-Dateien *mehrfach* eingebunden sind, sodass sich mehrfache Definitionen ergäben. Wenn im Beispiel auf Seite 123 sowohl *a.h* als auch *b.h* eine Datei *c.h* benötigten, müssten beide Dateien die Anweisung `#include "c.h"` enthalten. Durch die `#include`-Anweisungen in *meinprog.cpp* würde also *c.h* *zweimal* eingelesen. Abhilfe schaffen die Anweisungen `#if defined` (Abkürzung `#ifdef`), `#if !defined` (Abkürzung `#ifndef`), und `#define`.

```
// c.h
#ifndef C_H
#define C_H

void func_c1();
void func_c2();
enum Farbtyp {rot, gruen, blau, gelb};
#endif // C_H
```

Bedeutung:

Falls der (beliebige) Name *C_H* nicht definiert ist,
dann definiere *C_H* und akzeptiere alles bis `#endif`.

Die Wirkung des *ersten* Lesens von *c.h* als indirekte Folge von `#include "a.h"` in *meinprog.cpp* ist:

- `#ifndef C_H` liefert TRUE, weil `C_H` noch nicht definiert ist.
- `#define C_H` definiert `C_H`.
- Alles bis `#endif` wird gelesen.

Die Wirkung des *zweiten* Durchlaufs von *c.h* als indirekte Folge von `#include "b.h"` in *meinprog.cpp* ist:

- `#ifndef C_H` liefert FALSE (d.h. 0), weil `C_H` bereits definiert ist.
- Alles bis `#endif` wird ignoriert.

`#if`-Blöcke erstrecken sich nicht über Dateigrenzen. Nach `#endif` in derselben Zeile stehender Text zur Dokumentation ist nur erlaubt, wenn er als Kommentar markiert ist (siehe oben: `// C_H`). Mit `#undef` kann eine Definition rückgängig gemacht werden.

Makros mit `#define`

Es gibt eine weitere Bedeutung von `#define`, nämlich das Ersetzen von Makros durch Zeichenketten, wobei Parameter erlaubt sind. Mehrere Parameter werden durch Kommas getrennt. Die Makrodefinitionen

```
// nicht empfehlenswert! (Begründung folgt)
#define PI 3.14
#define schreibe cout
#define QUAD(x) ((x)*(x))
```

erlauben in einem Programm den Text

```
schreibe << PI << endl;
y = QUAD(z);
```

und würden interpretiert werden als:

```
cout << 3.14 << endl;
y = ((z)*(z));
```

Wenn ein Makro durch einen sehr langen Text ersetzt werden soll, der über mehrere Zeilen geht, ist jede Zeile mit Ausnahme der letzten mit einem `\` (Backslash) abzuschließen. Es ist möglich, mit einem Makro ganze Unterprogramme für verschiedene Datentypen zu schreiben, wobei der Datentyp der Parameter ist, der dem Makro übergeben wird. Eine bessere Möglichkeit dafür sind jedoch Funktionsschablonen oder `-templates`, die in Abschnitt 3.4 besprochen werden.

Die Textersetzung mit `#define` sollte im Allgemeinen *nicht* verwendet werden, wenn es Alternativen gibt. Wie gefährlich Makros sein können, lässt sich schon an dem einfachen `QUAD`-Makro zeigen. Der Aufruf

```
int z = 3;
int y = QUAD(++z);
```

soll `y` das Quadrat von `z` zuweisen, nachdem `z` um 1 erhöht wurde – oder? In Wirklichkeit wird `z` *zweimal* erhöht:

```
int y = ((++z)*(++z)); // expandierter Makroaufruf
```

und das Ergebnis ist falsch. Makronamen sind zudem einem symbolischen Debugger nicht zugänglich, wie die Pseudo-Konstante `PI` im obigen Beispiel. Ferner kann auf `PI` kein Zeiger (siehe Kapitel 5) gerichtet werden. Ein weiterer Nachteil von Makros besteht in der Umgehung der Typkontrolle:

```
// Makrodefinition
#define MULT(a,b) ((a)*(b))

// Makroaufruf
int a, b = 2, c = 3;
a = MULT(b,c); // ok
a = MULT(b, "Fehler!");
```

Der Compiler bekommt das Makro durch den vorgeschalteten Präprozessor gar nicht erst zu sehen, sondern bekommt nur das Ergebnis der Makroexpansion (= Textersetzung) vorgesetzt. Deshalb sind Compilerfehlermeldungen bei Fehlern innerhalb großer Makros manchmal nicht ohne Weiteres nachvollziehbar. Eine übliche Anwendung des Makros `#define` zur Textersetzung mit Parametern ist die gezielte Ein- und Ausblendung von Testsequenzen in einem Programm. Beispiel:

```
#define TEST_EIN
#ifdef TEST_EIN
    #define TESTANWEISUNG(irgendwas) irgendwas
#else
    #define TESTANWEISUNG(irgendwas) /* nichts */
#endif
// ... irgendwelcher Programmcode

// nur im Test soll bei Fehlern eine Meldung ausgegeben werden:
TESTANWEISUNG(if(x < 0) cout << "sqrt(negative Zahl)!" << endl;)
y = sqrt(x);
// ... mehr Programmcode
```

Der Parameter ist `irgendwas`. Falls `TEST_EIN` gesetzt ist, wird beim Compilieren durch den Präprozessor überall im Programm `TESTANWEISUNG(irgendwas)` durch `irgendwas` ersetzt. Wenn nach erfolgreichem Testen des Programms alle Testanweisungen verschwinden sollen, genügt es, die Zeile `#define TEST_EIN` zu löschen oder mit `//` in einen Kommentar zu verwandeln, mit der Wirkung, dass der Präprozessor jede `TESTANWEISUNG()` durch einen Kommentar `/*nichts*/` ersetzt, der schlicht ignoriert wird. `#define`-Makros können mehrere durch Kommas getrennte Parameter enthalten. In `irgendwas` sollte kein Komma enthalten sein, weil der Präprozessor sich sonst über die falsche Parameteranzahl beschwert. Zusammengefasst hat dieses Vorgehen zwei Vorteile:

- Nach Testabschluss wird das lauffähige Programm schneller und benötigt weniger Speicher durch die fehlenden Testanweisungen.
- Die Testanweisungen können im Programm zum späteren Gebrauch stehen bleiben. Sie müssen nicht einzeln auskommentiert oder gelöscht werden.

Die Technik, durch Makros gesteuert verschiedene Dinge ein- oder auszuschließen, wird sehr gut in den Header-Dateien des *include*-Verzeichnisses des Compilers sichtbar. Schauen Sie mal nach! Diese Art der Makrobenutzung ist weit verbreitet und hat ihre Vorteile.

Es gibt jedoch eine Lösung, die nur mit den Sprachelementen von C++ auskommt (also ohne Makros, die ja vom Präprozessor verarbeitet werden):

```
const bool TEST_EIN = true;
// ... irgendwelcher Programmcode
// nur im Test soll bei Fehlern eine Meldung ausgegeben werden:
if(TEST_EIN) if(x < 0) cout << "sqrt(negative Zahl)!" << endl;
y = sqrt(x);
```

Diese Lösung hat die gleichen oben genannten Vorteile. Die einzige Voraussetzung ist, dass der Compiler »toten« Programmcode von vornherein ignoriert, falls nämlich nach Abschluss der Testphase die erste Zeile in `const bool TEST_EIN = false;` geändert und dadurch die `if`-Anweisung überflüssig wird. Dies ist für einen modernen Compiler kein Problem.

Umwandlung von Parametern in Zeichenketten

Speziell für Testausgaben ist das Makro `PRINT` nützlich, das das Argument mit vorangestelltem `#` in eine Zeichenkette wandelt. Ohne einen Namen oder einen Ausdruck doppelt schreiben zu müssen, hat man Text und Ergebnis auf dem Bildschirm:

```
#define PRINT(X) cout << (#X) << "= " << (X) << endl;
```

Damit kann kurz zum Beispiel

```
PRINT(int(xptr)-int(xptr2));
```

geschrieben werden anstatt

```
cout << "int(xptr)-int(xptr2) = " << int(xptr)-int(xptr2) << endl;
```

mit dem Ergebnis `int(xptr)-int(xptr2) = 4` auf dem Bildschirm.

Empfehlung für den Aufbau von Header-Dateien

Eine Möglichkeit für den Aufbau von Header-Dateien ist das folgende Schema:

```
// Dateiname: fn.h
#ifndef fn_h
#define fn_h fn_h
// hier folgen die Deklarationen
#endif // fn_h
```

Um stets eindeutige Namen zu gewährleisten, empfiehlt sich die Ableitung aus dem Dateinamen, so wie `fn_h` aus `fn.h` entstanden ist. Warum tritt aber `fn_h` doppelt auf? Sehen wir uns dazu folgendes Beispiel an:

```
// Dateiname: fn.h
#ifndef fn_h
#define fn_h

// hier folgen die Deklarationen
enum fn_h { a, b, c};           // Fehler!
#endif // fn_h
```

Es kann sein, dass zufällig derselbe Name im nachfolgenden Programmcode auftritt, weil der Dateiname meistens mit dem Dateiinhalt zu tun hat. `#define` dient zur Textersetzung oder definiert etwas als logisch wahr. Das zweite Auftreten ist für den Compiler nicht verständlich, weil anstatt `fn_h` eine 1 (= wahr) gesehen wird. Es empfiehlt sich also, entweder Variablennamen mit der Endung `_h` zu vermeiden oder den Text durch sich selbst zu ersetzen, damit er keine Änderung erfährt: `#define fn_h fn_h`.

Natürlich vermindert bereits die Endung `_h` die Gefahr einer zufälligen Namensgleichheit. Eine Alternative ist das Hervorheben durch Großschreibung, wie in der C-Welt üblich, meistens ohne Verdopplung – so wird es in diesem Buch gehandhabt:

```
#ifndef FN_H
#define FN_H
// ... usw.
```

Namespaces sollten *nicht* mit `using` in einer Header-Datei eingeführt werden, weil sie damit in allen Dateien bekannt werden, die diese Header-Datei verwenden, und damit Namenskonflikte produzieren können (Einzelheiten siehe Abschnitt 3.6 ab Seite 141). Besser ist die qualifizierte Ansprache entsprechend der zweiten der auf Seite 60 beschriebenen Möglichkeiten. Beispiel: In einer Header-Datei sollte `cout` weder mit `using std::cout;`, noch mit `using namespace std;` eingeführt, sondern qualifiziert als `std::cout` benannt werden.

Verifizieren logischer Annahmen zur Laufzeit mit `assert`

Ein weiteres sehr nützliches Makro ist `assert()` zur Überprüfung logischer Annahmen, die an der Stelle des Makros gültig sein sollen. Insbesondere lassen sich die in Abschnitt 3.2.8 beschriebenen Vor- und Nachbedingungen verifizieren. Das Wort `assert()` leitet sich vom englischen Wort *assertion* ab, das auf Deutsch »Zusicherung« heißt. Zusicherungen werden mit dem Header `<cassert>` eingebunden. Beispiel:

```
#include<cassert> // enthält Makrodefinition

const int GRENZE = 100;
int index;
// .... Berechnung von index
// Test auf Einhaltung der Grenzen:
assert(index >= 0 && index < GRENZE);
```

Falls die Annahme `(index >= 0 && index < GRENZE)` nicht stimmen sollte, wird das Programm mit einer Fehlermeldung abgebrochen, die die zu verifizierende logische Annahme, die Datei und die Nummer der Zeile enthält, in der der Fehler aufgetreten ist. Eine andere Möglichkeit wäre das »Werfen einer Ausnahme«, siehe Abschnitt 8.1. `assert()` ist wirkungslos, falls `NDEBUG` vor `#include<cassert>` definiert wurde, entweder durch die Compilerdirektive `#define NDEBUG` oder durch Setzen des Compilerschalters `-D`, mit dem Makrodefinitionen voreingestellt werden. Anwendungsbeispiel: `g++ -DNDEBUG mein-Programm.cpp`

Werfen Sie einen Blick in die Datei `assert.h`, um die Wirkungsweise des Makros zu studieren! Der oben im `PRINT()`-Makro verwendete Präprozessoroperator `#` verwandelt die logische Annahme in eine Zeichenkette. Die innerhalb von `assert()` verwendeten vordefinierten Makros `__FILE__` und `__LINE__` werden beim Compilieren durch einen String mit

dem Dateinamen beziehungsweise durch die Zeilennummer ersetzt. Die vordefinierten Makros können Sie innerhalb selbst geschriebener Makros verwenden, ebenso wie `__DATE__` und `__TIME__`, die Datum und Uhrzeit der Übersetzung in einen String verwandeln.



Tipp

Vermeiden Sie Seiteneffekte in `assert()` und anderen Makros!

Eine in der Zusicherung aufgerufene Funktion wird bei gesetztem `NDEBUG` nicht ausgeführt! Beispiel:

```
assert(datei_oeffnen(filename) == erfolgreich); // Fehler
assert(GRENZE <= max(x, y));                  // Fehler
```

Falls `NDEBUG` definiert ist, wird die Datei nicht geöffnet, und weder wird das Maximum von `x` und `y` berechnet noch `GRENZE` mit irgendeinem Wert verglichen.

Verifizieren logischer Annahmen zur Compilationszeit mit `static_assert`

Die Prüfung mit `assert` geschieht zur Laufzeit. Manchmal möchte man aber bereits zur Compilationszeit bekannte Annahmen prüfen. Zum Beispiel soll `long` statt `int` eingesetzt werden, um den Zahlenbereich zu erweitern. Es ist aber systemabhängig und nicht garantiert, dass die Anzahl der Bits für `long` größer als die für `int` ist. Die Prüfung wird mit `static_assert` durchgeführt:

```
static_assert(sizeof(long) > sizeof(int), "long hat nicht mehr Bits als int!");
```

Wenn die Behauptung `sizeof(long) > sizeof(int)` falsch ist, gibt schon der Compiler die Fehlermeldung »long hat nicht mehr Bits als int!« aus. In diesem Fall bringt der Ersatz von `int` durch `long` gar nichts. `static_assert` ist kein Makro, sondern ein neues Schlüsselwort.



Übungen

3.8 Warum sollte man das oben vorgestellte Makro `QUAD(x)` *nicht* viel einfacher so formulieren: `#define QUAD(x) x*x` ?

3.9 Strukturieren Sie die Lösung der Taschenrechneraufgabe von Seite 121 entsprechend den Empfehlungen des Abschnitts 3.3.

3.4 Funktions-Templates

Oft ist dieselbe Aufgabe für verschiedene Datentypen zu erledigen, zum Beispiel Sortieren eines `int`-Arrays, eines `double`-Arrays und eines `String`-Arrays. Das Kopieren einer Sortierfunktion für verschiedene Datentypen ist fehleranfällig, weil Änderungen in allen Versionen nachgezogen werden müssen. Mit *Templates* (deutsch *Schablonen*) können Funktionen mit parametrisierten Datentypen geschrieben werden. Mit »parametrisierten Datentypen« ist gemeint, dass eine Funktion für einen beliebigen, noch festzulegenden

Datentyp geschrieben wird. Für den noch unbestimmten Datentyp wird ein Platzhalter (Parameter) eingefügt, der später durch den tatsächlich benötigten Datentyp ersetzt wird. Die allgemeine Form für Funktions-Templates zeigt Abbildung 3.10.

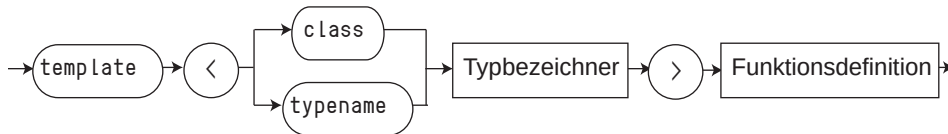


Abbildung 3.10: Syntax eines Funktions-Templates

Anstatt `class` kann auch `typename` geschrieben werden. Dabei ist *Typbezeichner* ein beliebiger Name, der in der *Funktionsdefinition* als Datentyp verwendet wird. In diesem Buch wird in der Regel `class` verwendet, wenn das Template eher für Klassen-Objekte verwendet werden soll. Wenn sowohl Klassen wie auch Grunddatentypen wie `int` und `double` gemeint sind, wird meistens `typename` gewählt.

Das folgende Programmbeispiel sortiert ein `int`-Feld und ein `double`-Feld, obwohl nur eine *einzig*e Funktion `quicksort()` geschrieben wird, die hier den Datentypplatzhalter `T` benutzt. In der C++-Standardbibliothek gibt es eine Funktion `sort()` zum Sortieren (siehe Seite 667), die hier bewusst ignoriert wird, um den Template-Mechanismus am Beispiel zu demonstrieren. Die Wirkungsweise des Quicksort ist zum Beispiel in [CLR] beschrieben.

Damit ist `quicksort()` leicht anwendbar für Vektoren von Objekten einer beliebigen Klasse. Die Begriffe Objekt und Klasse sind Ihnen schon im ersten Kapitel begegnet, mehr erfahren Sie im nächsten Kapitel.

Listing 3.10: Funktions-Templates

```

// cppbuch/k3/qsor.cpp
#include<iostream>
#include<vector>
using namespace std;

template<typename T> // Template mit T als Parameter für den Datentyp (Platzhalter)
void tausche(T& a, T& b) { // a und b vertauschen
    const T TEMP = a;
    a = b;
    b = TEMP;
}

// Die folgende Funktion würde auch mit einer Parameterliste (T a, T b) arbeiten, d.h. einer
// Kopie per Wert. Da T aber für einen beliebigen Datentyp steht, wird eine Referenz bevorzugt,
// um Kopien von möglicherweise sehr großen Objekten zu vermeiden.
template<typename T>
bool kleiner(const T& a, const T& b) { // Vergleich
    return a < b;                    // zu < siehe auch Text am Abschnittsende
}

template<typename T>
void drucke(const vector<T>& V) {
    for (unsigned int i = 0; i < V.size(); ++i) {

```

```

        cout << V[i] << " ";
    }
    cout << endl;
}

template<typename T>
void quicksort(vector<T>& a, int links, int rechts) {
    int li = links,
        re = rechts;
    // Die Verwendung von unsigned int für li und re wäre falsch, weil bei der
    // --Operation unten auch der Wert -1 auftreten kann. Eine Mischung der Typen
    // int und unsigned in Vergleichen oder bei impliziten Typkonversionen provoziert
    // ohnehin leicht Programmierfehler, die der Compiler nicht entdeckt (siehe Beispiel
    // if(u < i)...auf Seite 66).
    T el = a[(links+rechts)/2];
    do {
        while(kleiner(a[li], el)) ++li;
        while(kleiner(el, a[re])) --re;
        if (li < re) tausche(a[li], a[re]);
        if (li <= re) ++li; --re;
    } while (li <= re);
    if (links < re) quicksort(a, links, re);
    if (li < rechts) quicksort(a, li, rechts);
}

int main () {
    vector<int> iV(10);
    iV[0]=100; iV[1]=22; iV[2]=-3; iV[3]=44; iV[4]=6;
    iV[5]= -9; iV[6]=-2; iV[7]= 1; iV[8]=8; iV[9]=9;
    // In den folgenden beiden Anweisungen werden vom Compiler, gesteuert durch den
    // Datentyp vector<int> des Parameters iV, aus den obigen Templates die Funktionen
    // quicksort( vector<int>&, int, int) und drucke(const vector<int>&) erzeugt,
    // ebenso wie die implizit aufgerufenen Funktionen tausche(int&, int&)
    // und kleiner(const int&, const int&).
    quicksort(iV, 0, iV.size()-1);
    drucke(iV);
    vector<double> dV(8);
    dV[0]=1.09; dV[1]=2.2; dV[2]=79.6; dV[3]=-1.9; dV[4]=2.7;
    dV[5]=100.9; dV[6]=18.8; dV[7]=99.9;
    // Generierung der überladenen Funktionen quicksort(vector<double>&, int, int) und
    // drucke(const vector<double>&) (und der aufgerufenen Funktionen
    // tausche(double&, double&) und kleiner(const double&, const double&)):
    quicksort(dV, 0, dV.size()-1);
    drucke(dV);
} // Ende von main()

```

Innerhalb von `main()` stellt der Compiler anhand des Funktionsaufrufs fest, für welchen Datentyp die Funktion benötigt wird, und bildet die Definition mithilfe des Templates. Für jeden verwendeten Datentyp wird vom Compiler aus dem Template eine Funktion erzeugt – ebenso wie Sie mit *einer* Form einen Schokoladen- und einen Nußkuchen backen können.

Mit `quicksort()` liegt ein universelles Sortierprogramm vor, das für verschiedene Datentypen geeignet ist. Auffällig ist, dass der Vergleich, welches Element kleiner ist, als Funktionsaufruf `kleiner()` innerhalb `quicksort()` formuliert wurde, anstatt `while(feld[i] < feld[j])` zu schreiben. Warum? Es ist nicht selbstverständlich, dass der Operator `<` für beliebige Klassen und Datentypen definiert ist. Durch Auslagern des Vergleichs braucht die Funktion `quicksort()` nicht verändert zu werden, wenn der Operator `<` anders definiert werden muss. Man kann mit einer ausgelagerten Vergleichsfunktion oder mit später zu besprechenden Zeigern auf Funktionen (Seite 223) oder mit Funktionsobjekten (Seite 344) leichter die Sortierung nach verschiedenen Kriterien realisieren.

3.4.1 Spezialisierung von Templates

Nehmen wir an, dass `double`-Zahlen wie bisher, `int`-Zahlen jedoch nach dem *Absolutbetrag* sortiert werden sollen. Der Vergleichsoperator `<` kann dann nicht mehr direkt auf die Zahlen angewendet werden. Zur Realisierung können wir aber ausnutzen, dass ein Template für festzulegende Datentypen *spezialisiert* werden kann. Um die Sortierung nach dem Absolutbetrag nur für `int`-Zahlen durchzuführen, muss das Template für die Funktion `kleiner()` spezialisiert werden. Spezialfälle von überladenen Funktionen werden *nach* den nicht-spezialisierten Templates eingefügt, so auch das spezialisierte Template `kleiner<int>()`:

```
// #include<cstdlib> für abs() nicht vergessen!
template<>
bool kleiner<int>(const int& a, const int& b) {
    // Das int in kleiner<int> darf weggelassen werden (Typdeduktion).
    return abs(a) < abs(b); // Vergleich nach dem Absolutbetrag!
}
```

Anstelle eines spezialisierten Templates kann auch eine gewöhnliche Funktion treten, die vom Compiler bevorzugt gewählt wird, wenn die Parametertypen passen, etwa

```
bool kleiner(int a, int b) { // gewöhnliche Funktion
    return abs(a) < abs(b); // Vergleich nach dem Absolutbetrag!
}
```

Der Unterschied besteht darin, dass in der Parameterliste einer gewöhnlichen Funktion weitgehende Typumwandlungen möglich sind, in einem spezialisierten Template jedoch nicht. Zum Beispiel könnte man `kleiner()` benutzen, um sich die jeweils kleinere Zahl anzeigen zu lassen:

```
// OHNE gewöhnliche Funktion, aber mit spezialisiertem Template
cout << (kleiner(3, 6) ? 3 : 6) << endl; // Ausgabe 3
cout << (kleiner(3.4, 6) ? 3 : 6) << endl; // Fehlermeldung:
// no matching function for call to kleiner(double, int)

// MIT gewöhnlicher Funktion, aber OHNE Templates
cout << (kleiner(3, 6) ? 3 : 6) << endl; // Ausgabe 3
cout << (kleiner(3.1, 3.3) ? 3.1 : 3.3) << endl; // Ausgabe 3.3
// falsch wegen Genauigkeitsverlust bei der Umwandlung
```

Weil schärfere Typprüfungen normalerweise erwünscht sind, sollten spezialisierte Templates statt gewöhnlicher Funktionen eingesetzt werden.

3.4.2 Einbinden von Templates

Im Abschnitt 3.3.2 auf Seite 123 wurde gezeigt, wie Funktionsimplementationen vorübersetzt und dann eingebunden werden können. Das gilt nicht für Funktions-Templates! Eine *.o-Datei, die vom Compiler durch Übersetzen einer Datei nur mit einem Template erzeugt wird, ohne in der Datei einen konkreten Datentyp davon zu erzeugen, enthält keinen Programmcode und keine Daten.

Ein Template ist keine Funktionsdefinition im bisherigen Sinne, sondern eben eine Schablone, nach der der Compiler *erst bei Bedarf* eine Funktion zu einem konkreten Datentyp erzeugt. Dateien mit Templates sind deswegen mit `#include` einzulesen. Demzufolge könnten die Template-Definitionen ebenso gut in der Header-Datei stehen. Manchmal wird eine besondere Extension für den Dateinamen verwendet, zum Beispiel `.t`:

```
// Datei schablone.t
#ifndef SCHABLONE_T
#define SCHABLONE_T
// hier folgen die Template-Deklarationen
//...

// ---- Implementierung ----
// hier folgen die Template-Definitionen
//...
#endif
```

Diese Dateien werden wie *.h-Dateien eingelesen, obwohl sie Template-Definitionen enthalten. Die Template-Definitionen können natürlich auch `inline` sein. Diese Lösung wird häufig in diesem Buch verwendet, weil sie einfacher ist (weniger Dateien) und weil die nicht eindeutige Zuordnung von Templates zu Header- oder Implementationsdateien vermieden wird. Es handelt sich hierbei nur um organisatorische Maßnahmen zur Einbindung von Templates. Ein weiteres Template-Compilationsmodell wird in Abschnitt 23.7 (Seite 619) diskutiert.



Übungen

3.10 Schreiben Sie eine Template-Funktion `getType(T t)` mit Template-Spezialisierungen, die den Typ des Parameters `t` als String zurückgibt. Eine möglich Anwendung könnte so aussehen (Ausgabe des Programms siehe Kommentare `//`):

```
#include<iostream>
#include"gettype.t"
using namespace std;
int main() { // Ausgabe
    int i;
    cout << getType(i) << endl; // int
    unsigned int ui;
    cout << getType(ui) << endl; // unsigned int
    char c;
    cout << getType(c) << endl; // char
    bool b;
    cout << getType(b) << endl; // bool
    float f; // Annahme: float ist nicht in getType() berücksichtigt:
    cout << getType(f) << endl; // unbekannter Typ!
}
```

3.11 Schreiben Sie eine Template-Funktion `betrag(T t)`, die genau wie `abs()` den Betrag von `t` zurückgibt. Für manche Grunddatentypen wie `char` oder `bool` ist der Begriff »Betrag« nicht sinnvoll. Überlegen Sie, wie Sie durch eine spezialisierte Template-Funktion erreichen können, dass eine fälschliche Verwendung von `betrag()` mit einem `bool`-Argument zur Ausgabe einer Fehlermeldung und anschließendem Programmabbruch führt.

3.5 inline-Funktionen

Ein Funktionsaufruf kostet Zeit. Der Zustand des Aufrufers muss gesichert und Parameter müssen eventuell kopiert werden. Das Programm springt an eine andere Stelle und nach Ende der Funktion wieder zurück zur Anweisung nach dem Aufruf. Der relative Verwaltungsaufwand fällt umso stärker ins Gewicht, je weniger Zeit die Abarbeitung des Funktionskörpers selbst verbraucht. Der absolute Aufwand macht sich mit steigender Anzahl der Aufrufe bemerkbar, zum Beispiel in Schleifen. Um diesen Aufwand zu vermeiden, können Funktionen als `inline` deklariert werden. `inline` bewirkt, dass bei der Compilation der Aufruf durch den Funktionskörper ersetzt wird, also gar kein echter Funktionsaufruf erfolgt. Die Parameter werden entsprechend ersetzt, auch die Syntaxprüfung bleibt erhalten. Betrachten wir die einfache Funktion `quadrat()`, die das Quadrat einer Zahl zurückgibt:

```
inline int quadrat(int x) {
    return x*x;
}
```

Der Aufruf `z = quadrat(100)`; wird wegen des Schlüsselworts `inline` vom Compiler durch `z = 100*100`; ersetzt. Gute Compiler würden darüber hinaus den konstanten Ausdruck berechnen und `z = 10000`; einsetzen. Der Verwaltungsaufwand für den Aufruf einer Funktion entfällt, das Programm wird schneller. Es ist nicht sinnvoll, die Ersetzung von vornherein *selbst* vorzunehmen, weil bei einer Änderung der Funktion alle betroffenen Stellen geändert werden müssten anstatt nur die Funktion selbst. `inline`-Deklarationen empfehlen sich ausschließlich für Funktionen mit einem Funktionskörper kurzer Ausführungszeit im Vergleich zum Verwaltungsaufwand für den Aufruf. `inline` ist nur eine *Empfehlung* an den Compiler, die Ersetzung vorzunehmen, er muss sich nicht daran halten. `inline`-Deklarationen sollten sich ausschließlich in Header-Dateien befinden. Der Grund wird klar, wenn wir das Gegenteil annehmen. Im Beispiel liegen drei Dateien vor:

```
// Deklarationsdatei X.h
int f(int);
int g(int);
```

```
// Datei X.cpp
#include "X.h"
inline int f(int a) { // Fehler: inline in Definitionsdatei!
    return a*a;
}
```



```
int g(int a) {
    a += 1;
    return f(a);    // inline ist hier bekannt
}
```

```
// Datei main.cpp
#include "X.h"      // enthält kein inline
int main() {
    int a = 1, b;
    b = f(a) + g(a); // inline ist hier unbekannt!
}
```

Bei getrennter Compilation der Dateien *X.cpp* und *main.cpp* gibt es zwei Fälle:

1. Innerhalb der Funktion *g()* kann die *inline*-Ersetzung von *f()* vorgenommen werden.
2. In *main.cpp* weiß der Compiler nichts davon, dass *f()* *inline* sein soll, und nimmt einen Funktionsaufruf an.

Die Konsequenz ist, dass der Linker die Definition von *f(int)* nicht findet und sich mit einer Fehlermeldung¹ verabschiedet. Richtig wäre folgende Struktur:

```
// Datei X.h
inline int f(int a) { // inline in Deklarationsdatei!
    return a*a;
}

int g(int);
```

```
// Datei X.cpp
#include "X.h"
int g(int a) {
    a += 1;
    return f(a);    // inline ist hier bekannt
}
```

```
// Datei main.cpp
#include "X.h"
int main() {
    int a = 1, b;
    b = f(a) + g(a); // inline von f(int) ist hier bekannt!
}
```

Inline-Funktionen werden in jeder Übersetzungseinheit expandiert und unterliegen damit dem internen Linken (vergleiche Seite 126).

¹ Der G++-Compiler ignoriert *inline*, wenn keine Optimierung eingeschaltet ist. Durch Einschalten der Optimierung wird der Fehler sichtbar, etwa `g++ -O2 X.cpp main.cpp`.

3.6 Namensräume

Ein Namensraum (englisch *namespace*) ist ein mit Namen gekennzeichneter Sichtbarkeitsbereich (scope). Ein Namespace erlaubt die Gruppierung zusammengehöriger Programmteile. Namespaces sind auch eingeführt worden, damit verschiedene Programmteile zusammenarbeiten können, die vorher (ohne Namespaces) aufgrund von Namenskonflikten im globalen Sichtbarkeitsbereich nicht zusammen verwendet werden konnten. Beispiel:

```
// abc.h (nützliche Funktionen der ABC-GmbH)
int print(const char*);
void func(double);

// xyz.h (nützliche Funktionen der XYZ Enterprises Ltd.)
int print(const char*);
void func();
```

```
// main.cpp
#include "abc.h"
#include "xyz.h"
int main() {
    print("hello world!");    // welches print()?
    func(1524.926);          // ok, überladen
    func();                   // ok, überladen
}
```

Es ist so nicht möglich, die Funktionsbibliotheken beider Firmen gleichzeitig zu benutzen. Die Lösung besteht in der Einführung von zusätzlichen, übergeordneten Sichtbarkeitsbereichen, den Namespaces. Die Deklaration ähnelt der von Klassen:

```
namespace abc {
    int print(const char*);
} // ; ist nicht notwendig
```

Klassen und Funktionen werden durch *Using-Direktiven* nutzbar gemacht:

```
// abc.h (nützliche Funktionen der ABC-GmbH)
namespace abc {
    int print(const char*);
    void func(double);
}

// main.cpp
#include "abc.h"
#include "xyz.h"
int main() {
    // Using-Direktive:
    using namespace abc; // alle Namen aus abc zugänglich machen
    print("hello world!"); // = abc::print()
}
```

Eine andere Möglichkeit ist der gezielte Zugriff auf Teile eines Namespace durch eine Using-Deklaration oder einen qualifizierten Namen, der die Funktion oder Klasse über den Bereichsoperator `::` anspricht.

```
int main() {
    abc::print("hello world!");    // qualifizierter Name
    using abc::print;              // Using-Deklaration: lokales Synonym einführen
    print("hello world!");         // = abc::print()
}
```

Alle Klassen und Funktionen der C++-Standardbibliothek (Kapitel 26) sind im Namespace `std`. Aus diesem Grund wird in den Programmen dieses Buchs häufig `using namespace std;` benutzt. Alternativ ist der Zugriff über einen qualifizierten Namen möglich, zum Beispiel

```
std::cout << "keine using-Deklaration notwendig!";
```

Bei sehr langen Namen besteht die Möglichkeit der Abkürzung:

```
namespace SpecialSoftwareGmbH_KlassenBibliothek {
    // ....
}
// Abkürzung
namespace sskb = SpecialSoftwareGmbH_KlassenBibliothek;
using namespace sskb; // Benutzung der Abkürzung
```

3.7 C++-Header

In C/C++ gibt es Bibliotheken mit verschiedenen Klassen und Funktionen. Die Funktionsbibliotheken entstammen teilweise der Sprache C. Beim Linken werden die benötigten Funktionen aus den Bibliotheksdateien dazugebunden. Die Header-Dateien sind im *include*-Verzeichnis zu finden. Auf die zu C++ gehörende Bibliothek wird in Kapitel 26 ab Seite 741 eingegangen. Ab Seite 873 werden die aus der Programmiersprache C kommenden Funktionen beschrieben.

Programme erhalten Zugriff zu Standardfunktionen und Klassen über das Einschließen der passenden Header mit `#include`. Diesen Headern können (müssen aber nicht) Dateien mit demselben oder ähnlichen Namen entsprechen. Die Namen der Header sind vom C++-Standard vorgeschrieben, die Implementierung durch die Compilerhersteller nicht. Alle Funktionsprototypen der C-Include-Dateien, deren Dateiname auf *».h«* endet, gehören zur Sprache C und zum *globalen* Namensraum. Dieselben Funktionen werden auch unter C++ zur Verfügung gestellt, aber unter neuen Dateinamen, die sich durch ein vorangestelltes *»c«* und das Fehlen der Datei-Extension *».h«* unterscheiden. Die Funktionen sind dann im Namespace `std` definiert. Die Möglichkeit, Funktionen mit der Datei-Extension *».h«* einzubinden, bleibt unberührt. Eine C-Header-Datei *name.h* eines C++-Systems enthält den zugehörigen Header `<cname>`:

```
// name.h
#ifndef NAME_H
#define NAME_H
#include<cname>
using namespace std; // Namen global sichtbar machen
#endif

// cname
#ifndef CNAME
#define CNAME

namespace std {
    extern "C" void func(); // ein C-Prototyp, siehe Seite 144
    // ...
}
#endif
```

Die Beispiele zeigen die verschiedenen Möglichkeiten für #include:

```
// Beispiel 1: C-Funktionen, globaler Namensraum
#include<string.h>           // strlen()
#include<iostream>

int main() {
    char text[] = "Hello";
    std::cout << "Die Länge von " << text
                << " ist " << strlen(text) << std::endl;
}
```

```
// Beispiel 2: C++, dieselbe Funktion, Namespace std
#include<cstring>
#include<iostream>
using namespace std;

int main() {
    char text[] = "Hello";
    cout << "Die Länge von " << text << " ist " << strlen(text) << endl;
}
```

Ohne using namespace std; hätte in Beispiel 2 std::strlen(text), std::cout und std::endl geschrieben werden müssen. Außer den C-Funktionen gibt es natürlich zusätzlich Standard-Header für die Klassen der C++-Standardbibliothek:

```
// Beispiel 3: C++, String-Klasse
#include<string>           // ohne .h-Extension
#include<iostream>
using namespace std;

int main() {
    string text("Hello");
    cout << "Die Länge von " << text << " ist " << text.length() << endl;
}
```

```
// Beispiel 4: C++, String-Klasse
#include<string>
#include<iostream>
using std::cout;           // begrenzte Auswahl
using std::endl;
using std::string;

int main() {
    string text("Hello");
    cout << "Die Länge von " << text << " ist " << text.length() << endl;
}
```

Um Namenskonflikte zu vermeiden, ist es grundsätzlich empfehlenswert, für ein Projekt (oder Teilprojekte) Namespaces zu definieren und zu benutzen. Besonders wichtig ist dies beim Schreiben von Bibliotheken. Bei der Benutzung sollte die selektive Auswahl wie etwa bei `std::cout` oder wie in Beispiel 4 bevorzugt werden. In Header-Dateien für Klassen sollte `using namespace std;` vermieden werden, damit Benutzer dieser Klassen nicht gezwungenermaßen `std` »erben«. In `main()`-Dateien ist die Verwendung hingegen unproblematisch.

3.7.1 Einbinden von C-Funktionen²

Das Einbinden von C-Funktionen wird mit `#include` bewerkstelligt. C-Prototypen werden in der Header-Datei mit

```
extern "C" {
    // .. hier folgen die C-Prototypen
}
```

eingebunden. Der Grund dafür liegt in der unterschiedlichen Behandlung von Funktionsnamen durch den C++-Compiler im Vergleich zu einem C-Compiler. Die korrekte Einbindung in C++-Header-Dateien wird in den C++-Header-Dateien über die Abfrage des Makros `__cplusplus` gesteuert. Damit werden `extern "C" {` samt schließender Klammer in eine Übersetzungseinheit integriert, wie unten zu sehen. Das Makro `__cplusplus` wird bei einer C++-Compilation automatisch gesetzt. Anstelle der Prototypen kann eine weitere `#include`-Anweisung stehen.

```
#ifdef __cplusplus
extern "C" {
#endif
    // .. hier folgen die C-Prototypen
#ifdef __cplusplus
}
#endif
```



Übungen

3.12 Gegeben sei die folgende Funktion³ `fastbubblesort()`, die schneller als die Bubble-Sort-Variante von Seite 82 ist:

² Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

³ Leider habe ich die Vorlage zu dieser Aufgabe, die ich nur abgewandelt habe, nicht mehr gefunden. Der Autor des Originals möge mir verzeihen.

```

void fastbubblesort(vector<int>& feld) {
    int temp;
    do {
        temp = feld[0];
        for(size_t j = 1; j < feld.size(); j++) {
            if(feld[j] < feld[j-1]) { // vertauschen
                temp      = feld[j-1];
                feld[j-1] = feld[j];
                feld[j]   = temp;
            }
        }
    } while(temp != feld[0]); // keine Vertauschung mehr
}

```

Warum sollte diese Funktion schneller sein? Sie vergleicht wie üblich ein Vektor-Element mit dem vorhergehenden und vertauscht die Elemente, sofern das Element kleiner als der Vorgänger ist. Dieser Vorgang wird solange wiederholt, bis das Element `temp` unverändert bleibt, also nichts mehr zu sortieren ist. Weil gegebenenfalls schnell erkannt wird, dass nichts mehr zu sortieren ist, ist dieser Bubble-Sort bei teilweise vorsortierten Feldern im Mittel etwas schneller als eine Variante mit zwei geschachtelten Schleifen fixer Durchlaufanzahl. *Leider, leider, enthält die Funktion zwei schwere Fehler! Welche?*

3.13 Wer Mathematik nicht mag, überspringe bitte diese Aufgabe. Schreiben Sie eine Funktion `double polynom(const vector<double>& k, double x)`, die den Wert des Polynoms $f(x) = k_n x^n + k_{n-1} x^{n-1} + \dots + k_1 x + k_0$ zurückgibt. Vermeiden Sie zur effizienten Berechnung unnötige Mehrfachberechnungen der Potenzen von x . Der Vektor `k` soll nur die $n + 1$ Koeffizienten enthalten, das heißt, `k[0]=k0`, `k[1]=k1` usw.

3.14 Schreiben Sie ein Programm, das eine exakte Kopie seines eigenen Quellcodes auf dem Bildschirm ausgibt, *ohne* auf eine Datei zuzugreifen (schwierig und nur für Knobel-freunde, Hinweis siehe [Hof06]).

3.15 Was ist an der folgenden Funktion falsch? Zur Erinnerung: `UINT_MAX` ist die größtmögliche unsigned int-Zahl.

```

void pruefeZahl(unsigned int zahl) { // prüfen, ob zahl im Bereich liegt
    if(zahl < 0 || zahl > UINT_MAX) {
        cerr << "Zahl " << zahl
              << " liegt nicht im Bereich!" << endl;
    }
}

```


4

Objektorientierung 1

Dieses Kapitel behandelt die folgenden Themen:

- Klasse und Objekt
- Konstruktion und Initialisierung von Objekten
- Destruktoren
- Der Weg von der Problemstellung zu Klassen und Objekten
- Gegenseitige Abhängigkeit von Klassen

Zunächst lernen Sie den Begriff *Abstrakter Datentyp* kennen, der die Grundlage für die anschließend erläuterten Begriffe *Klasse* und *Objekt* bildet. Ein Objekt vom Typ `ort` dient als durchgängiges Beispiel. Objekte müssen vor ihrer Verwendung erzeugt werden, und sie müssen sinnvolle Daten enthalten. Die Erzeugung und Initialisierung ist die Aufgabe der verschiedenen *Konstruktoren*. Ein vollständiges Beispiel zum Rechnen mit rationalen Zahlen demonstriert das bis dahin Gelernte. *Destruktoren* sind zum Aufräumen da – sie zerstören nicht mehr benötigte Objekte.

4.1 Abstrakte Datentypen

Bisher haben wir Datentypen und Funktionen kennengelernt. In der Einführung auf Seite 27 wurde der Unterschied zwischen Klassen und Objekten beschrieben und darauf hingewiesen, dass Daten und Funktionen eines Objekts zusammengehören. Die Programmierung, wie wir sie bis jetzt kennengelernt haben, erlaubt es durchaus, unzulässige Funktionen auf Daten anzuwenden, zum Beispiel eine Buchung auf ein Konto unter Umgehung von Kontrollmechanismen vorzunehmen. Um unzulässige Zugriffe und damit auch versehentliche Fehler zu vermeiden, sollten die Daten *gekapselt* werden, indem man zusammengehörige *Daten* und *Funktionen* zusammenfasst. Das dadurch entstehende Gebilde heißt *Abstrakter Datentyp*.

Der Sinn liegt darin, den richtigen Gebrauch der Daten sicherzustellen. Die *tatsächliche* Implementierung der Datenstrukturen ist nach außen nicht sichtbar. Deshalb werden Datenstrukturen eines Abstrakten Datentyps ausschließlich durch die mit diesen Daten möglichen Operationen beschrieben. Von der internen Darstellung wird abstrahiert.

Mit »Funktion« ist hier *nicht* die konkrete Implementierung gemeint, das heißt, *wie* die Funktion im Einzelnen auf die Daten wirkt. Zur Verwendung eines Abstrakten Datentyps reicht die Spezifikation der Zugriffsoperation aus.

Abstrakter Datentyp = Datentypen + Funktionen

Ferner sind logisch zusammengehörige Dinge an einem Ort konzentriert. Die zusammen mit Daten gekapselten Funktionen heißen im Folgenden Elementfunktionen (englisch *member functions*). Der Zugriff auf die Daten soll *nur über Elementfunktionen* (auch Methoden genannt) möglich sein. Die Begriffe *Elementfunktion* und *Methode* werden synonym verwendet, obwohl der aus der Programmiersprache *Smalltalk* stammende Begriff *Methode* eigentlich besser auf die später zu besprechenden virtuellen Funktionen von C++ passt. Zum Vergleich sei der Zugriff auf zwei Koordinaten *x* und *y* eines Punktes auf verschiedene Arten gezeigt.

■ Unstrukturierter Zugriff

```
int x = 100; // x-Koordinate eines Punktes
int y = 0;   // y-Koordinate eines Punktes
```

Der Nachteil besteht darin, dass an jeder Stelle im Programm *x* und *y* ungeschützt verändert werden können. Eine Erweiterung der Funktionalität, zum Beispiel Protokollierung der Änderungen in einer Datei, muss an jeder Stelle nachgetragen werden.

■ Strukturierter Zugriff

Hier werden die Daten in eine Struktur gepackt (siehe Seite 88); der verändernde Zugriff geschieht über eine Funktion:

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

void aendern(Punkt& p, int x, int y) {
    // ... Plausibilitätsprüfung
    p.x = x;
    p.y = y;
}
```

```
// ... Protokollierung
}

// Aufruf
aendern(einPunkt, 10, 800);
```

Der Vorteil besteht in der Gruppierung logisch zusammengehöriger Daten und der Änderung über eine Funktion. Eine Erweiterung der Funktionalität ist leicht realisierbar. Der Nachteil besteht darin, dass auch hier ein zusätzlicher Zugriff an der Funktion vorbei möglich ist, zum Beispiel

```
einPunkt.x = -3000;
```

■ Abstrakter Datentyp

Die Daten werden, unterstützt durch die Programmiersprache, so gekapselt, dass ein Zugriff *ausschließlich* über eine Funktion geschieht. Abbildung 4.1 verdeutlicht das Prinzip. Die Funktion als öffentliche Schnittstelle gehört zur Datenkapsel und ist der einzige Zugang. Eine direkte Änderung der Daten unter Umgehung der Funktion ist unmöglich.

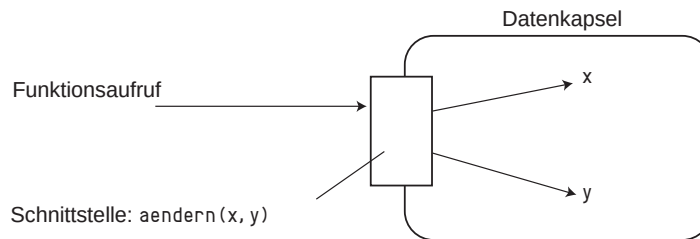


Abbildung 4.1: Abstrakter Datentyp

4.2 Klassen und Objekte

Eine Klasse ist ein Datentyp, genauer: ein Abstrakter Datentyp, der in einer Programmiersprache formuliert ist. Eine Klasse ist auch die *Beschreibung* von Objekten oder, anders ausgedrückt, *die Abstraktion von ähnlichen Eigenschaften und Verhaltensweisen ähnlicher Objekte*. Eine Klasse definiert die Struktur aller nach ihrem Muster erzeugten Objekte. In C++ dient die Klasse dazu, dem Compiler die Beschreibung von später zu definierenden Objekten mitzuteilen.

Ein Objekt hat einen inneren Zustand, der durch andere Objekte oder Elemente der in der Programmiersprache vorgegebenen Datentypen dargestellt wird. Der Zustand kann sich durch Aktivitäten des Objekts ändern, das heißt, durch Operationen, die auf Objektdaten ausgeführt werden. Die von jedem benutzbaren Operationen bilden die *öffentliche Schnittstelle*, in C++ gekennzeichnet durch das Schlüsselwort `public`.

Ein Objekt ist die konkrete Ausprägung einer Klasse, es belegt im Gegensatz zur Klasse Bereiche im Speicher, die mit definierten Bitmustern belegt sind, die Werte von Objekteigenschaften darstellen, wie der Betrag »15« der Wert in € der Eigenschaft Kontostand sein kann oder **rot** der Wert der Eigenschaft Farbe. Eigenschaften werden auch *Attribute* genannt, die bestimmte *Werte* besitzen. Der Zustand eines Objekts, der durch die zu den Attributen gehörenden Werte beschrieben wird, ist im Sinne des Abstrakten Datentyps nicht direkt änderbar, sondern nur über öffentliche Methoden. Die Attribute werden deshalb normalerweise mit dem Schlüsselwort `private` deklariert.

Ein Objekt besitzt eine *Identität*, die es unterscheidbar macht von einem beliebigen anderen Objekt, selbst wenn beide genau gleiche Daten enthalten. Die Identität zu einem bestimmten Zeitpunkt wird in C++ durch eine eindeutige Position im Speicher nachgebildet; zwei Objekte können niemals dieselbe Adresse haben, es sei denn, ein Objekt ist im anderen enthalten. Eine in C++ formulierte Klasse hat eine typische Gestalt:

```
class Klassenname {
public:
    Typ elementfunktion1();
    Typ elementfunktion2();
    // und weitere ...
private:
    Typ attribut1;
    Typ attribut2;
    // und weitere ...
};
```

Die reine Deklaration einer Elementfunktion wird auch *Prototyp* genannt. Zunächst betrachten wir ein einfaches Beispiel eines abstrakten Datentyps, nämlich einen *Ort*, der aus den X- und Y-Koordinaten besteht, sofern man sich auf zwei Dimensionen beschränkt. An Operationen seien vorgesehen:

```
getX()      X-Koordinate zurückgeben
getY()      Y-Koordinate zurückgeben
aendern()   X- und Y-Koordinaten ändern
```

Die Klasse wird `Ort1` genannt, weil sie noch Änderungen unterliegt. Sie ist wie folgt definiert:

Listing 4.1: Klasse `Ort1`, Version 1

```
// cppbuch/k4/ort1/ort1.h
#ifndef ORT1_H
#define ORT1_H
class Ort1 {                                // Version 1
public:
    int getX() const;
    int getY() const;
    void aendern(int x, int y); // x,y = neue Werte
private:
    int xKoordinate;
    int yKoordinate;
};                                           // Semikolon nicht vergessen!
#endif // ORT1_H
```

Daten *und* Methoden werden in einer Klasse zusammengefasst. Nach dem Schlüsselwort `class` folgen der Klassenname und ein durch geschweifte Klammern begrenzter Block mit den Deklarationen von Daten (Attributen) und Funktionen (Methoden). Der Gültigkeitsbereich von Klasselementen ist lokal zu der Klasse: Die Daten sind `privat`, das heißt, sie sind von außen nicht zugreifbar, sodass Anweisungen wie `xKoordinate = 13;` unmöglich sind. Das Schlüsselwort `private` könnte entfallen, weil die Voreinstellung ohnehin `private` ist, wenn die privaten Daten *vor* dem `public`-Bereich stehen würden. Alles nach dem Schlüsselwort `public` Deklarierte ist öffentlich zugänglich. Die Funktionen heißen Elementfunktionen oder Methoden, von denen es `private` und öffentliche geben kann. Eine Datenänderung kann ausschließlich über eine geeignete Methode erfolgen, zum Beispiel `aendern()`. Das Schlüsselwort `const` oben drückt aus, dass die damit ausgezeichneten Elementfunktionen das Objekt nicht verändern können und deswegen auch auf konstante Objekte anwendbar sind. `getX()` und `getY()` geben nur Zahlen zurück, ohne die privaten Daten zu verändern, während `aendern()` die vorherigen Werte der privaten Variablen überschreibt (mehr zu `const` folgt in Abschnitt 4.5).

Objekterzeugung und -benutzung

Wie kann nun ein Ort in einem Programm benutzt werden? Zunächst muss ein Objekt erzeugt werden, denn die obige Klassendeklaration *beschreibt* nur ein Objekt. Wir nehmen an, dass die Deklaration der Klasse in einer Datei `ort1.h` vorliegt. Der Mechanismus der Objekterzeugung ist die übliche Variablendefinition, wobei die Klasse den Datentyp darstellt:

Listing 4.2: Ort-Objekt benutzen

```
// cppbuch/k4/ort1/ort1main.cpp
#include "ort1.h"           // Definition der Klasse einlesen
#include <iostream>
using namespace std;

int main() {               // Anwendung der Ort1-Klasse
    Ort1 einOrt1;          // Objekt erzeugen
    einOrt1.aendern(100, 200);
    cout << "Der Ort hat die Koordinaten x = "
          << einOrt1.getX() << " und y = "
          << einOrt1.getY() << endl;
}
```

Im Sinne der Aufteilung von Schnittstelle und Implementation werden Klassen wie Strukturen (`struct`) behandelt, wie in Abschnitt 3.3.4 (Seite 126 f.) beschrieben. Die Implementation der Funktionen sei in der Datei `ort1.cpp` abgelegt (Beschreibung siehe unten), die eine Zeile `#include "ort1.h"` enthält. Bei der Variablendefinition wird für ein Objekt Speicherplatz bereitgestellt. Um dies zu bewerkstelligen, wird beim Ablauf des Programms an der Stelle der Definition eine besondere Klassenfunktion aufgerufen, die *Konstruktor* genannt wird. Oben wird ein Objekt namens `einOrt1` definiert, das durch den impliziten Aufruf eines *Konstruktors* erzeugt wird und Speicherplatz für die Daten belegt. Der Programmcode der Funktionen ist selbstverständlich nur einmal für alle erzeugten `Ort1`-Objekte vorhanden.

Der Konstruktor wird vom System automatisch bereitgestellt, kann aber auch selbst definiert werden, wie Sie ab Seite 154 sehen. Zunächst hat das Objekt keinen definierten Zustand. Der wird im Beispiel erst mit dem Aufruf `einOrt1.aendern(100, 200)`; hergestellt. Entsprechend der schon erläuterten Notation *Objektname.Anweisung* (gegebenenfalls *Daten*) erhält das Objekt den Auftrag, die Koordinaten auf (100, 200) zu ändern. Wie das Objekt diese Dienstleistung erbringt, ist in der Methode `aendern()` versteckt.

Im informationstechnischen Sprachgebrauch heißen Dinge (Objekte, Rechner etc.), die eine Dienstleistung erbringen, *Server*. Die Dienstleistung wird erbracht für einen *Client* (deutsch: Klient, Kunde), der selbst ein Rechner oder Objekt sein kann. Im obigen Programm ist das `main()`-Programm der Client, und das Objekt `einOrt1` ist der Server, der beauftragt wird, eine Koordinatenänderung durchzuführen.

Die Methode `aendern()` soll abgesehen von der Änderung der Koordinaten eine protokollierende Meldung auf den Bildschirm schreiben. Damit sind Änderungen unter Umgehung der Protokollierung unmöglich. Die Datei `ort1.cpp` enthält die Implementation der Methoden:

Listing 4.3: Implementierung von Ort1

```
// cppbuch/k4/ort1/ort1.cpp
#include "ort1.h"
#include <iostream>
using namespace std;

int Ort1::getX() const { return xKoordinate;}
int Ort1::getY() const { return yKoordinate;}

void Ort1::aendern(int x, int y) {
    xKoordinate = x;
    yKoordinate = y;
    cout << "Ort1-Objekt geändert! x = "
          << xKoordinate << " y = "
          << yKoordinate << endl;
}
```

Durch den Klassennamen und den Bereichsoperator `::` wird die Methode als zur Klasse `Ort1` gehörig gekennzeichnet. Daher darf *innerhalb* der Funktion auf die privaten Daten zugegriffen werden.

Das lauffähige Programm entsteht durch Übersetzen der Dateien `ort1.cpp` und `ort1main.cpp` und Linken der durch die Übersetzung entstandenen Dateien `ort1.o` und `ort1main.o`. Die Datei `ort1.h` wird während der Übersetzung der `*.cpp`-Dateien gelesen. Wenn sich alle zusammengehörigen Dateien in einem Verzeichnis befinden, werden Übersetzen und Linken mit dem Befehl `make` ausgeführt, falls keine Entwicklungsumgebung benutzt wird. Voraussetzung ist eine Datei namens `makefile`, die die Übersetzung steuert (mehr dazu siehe Kapitel 17). In den Beispielen auf der DVD ist dies stets der Fall.

4.2.1 inline-Elementfunktionen

Im Abschnitt 3.5 (Seite 139) haben wir den `inline`-Mechanismus für kleine Funktionen kennengelernt. Die Programmierung mit Klassen verwendet typischerweise viele kleine

Funktionen, sodass mit `inline` ein erheblicher Effizienzgewinn möglich ist. Dabei gibt es drei Möglichkeiten:

1. Deklaration *und* Definition innerhalb der Klasse. Beispiel:

```
// in ort1.h
class Ort1 {                                // Version 2
public:
    int getX() const { return xKoordinate;}
    int getY() const { return yKoordinate;}
    void aendern(int x, int y) {
        xKoordinate = x;
        yKoordinate = y;
        std::cout << "Ort1-Objekt geändert! x = "
                    << xKoordinate << " y = "
                    << yKoordinate << std::endl;
    }
private:
    int xKoordinate,
        yKoordinate;
};
```

2. Deklaration und Definition innerhalb der Header-Datei:

```
// in ort1.h
class Ort1 {                                // Version 3
public:                                     // nur Prototypen
    int getX() const;
    int getY() const;
    void aendern(int x, int y); // x,y = neue Werte
private:
    int xKoordinate,
        yKoordinate;
};

// ===== inline - Implementierung =====
inline int Ort1::getX() const { return xKoordinate;}
inline int Ort1::getY() const { return yKoordinate;}
inline void Ort1::aendern(int x, int y) {
    // ... wie oben
}
```

Diese Variante hat den Vorteil, dass die Implementierung der Methoden nicht direkt innerhalb der Klassendeklaration sichtbar ist. Dadurch kann eine Klassendeklaration übersichtlicher werden.

3. `inline`-Deklaration außerhalb der Klasse. Dies ist fehlerhaft und daher nicht empfehlenswert (siehe Text unten). Beispiel:

```
// in ort1.h
class Ort1 {                                // Version 4
    // ... wie oben
    int getX() const;
};
```

```
// in ort1.cpp!
inline int Ort1::getX() const { // Fehler!
    return xKoordinate;
}
```

Was im Beispiel 3, Version 4 auf den ersten Blick als möglich erscheint, ist aus den in Abschnitt 3.5 aufgeführten Gründen nicht praktikabel. Siehe dazu insbesondere den Hinweis auf Seite 139. Deshalb sollten nur die *ersten zwei* der drei beschriebenen Möglichkeiten benutzt werden.

4.3 Initialisierung und Konstruktoren

Objekte können während der Definition initialisiert, also mit sinnvollen Anfangswerten versehen werden. Es wurde bereits erwähnt, dass eine besondere Elementfunktion namens *Konstruktor* diese Arbeit neben der Bereitstellung von Speicherplatz übernimmt. Die Syntax von Konstruktoren ähnelt der von Funktionen, nur dass der Klassenname den Funktionsnamen ersetzt. Außerdem haben Konstruktoren keinen Return-Typ, auch nicht `void`. Im Sinn der Abstrakten Datentypen sind die Methoden einer Klasse für sinnvolle und konsistente Änderungen eines Objekts zuständig. Konstruktoren haben die Verantwortung, dass sich ein Objekt vom Augenblick der Entstehung an in einem korrekten Zustand befindet. Nun gibt es mehrere Arten von Initialisierungen, unterschieden durch verschiedene Arten von Konstruktoren, die im Folgenden beschrieben werden.

4.3.1 Standardkonstruktor

Falls kein Konstruktor angegeben wird, wird einer vom System automatisch erzeugt (implizite Konstruktordeklaration). Die Daten des Objekts enthalten dann unbestimmte Werte. Dieser vordefinierte Konstruktor (englisch *default constructor*) kann auch selbst geschrieben werden, um Attribute bei Anlage des Objekts zu initialisieren. Der Standardkonstruktor hat keine Argumente. Für eine Klasse `X` wird er einfach mit `X()` deklariert. Bei der Definition wird der Bezugsrahmen der Klasse angegeben, um dem Compiler mitzuteilen, dass es sich um eine Methode der Klasse `X` handelt, also `X::X() { ... }`. In unserem Beispiel soll erreicht werden, dass bei Erzeugung eines `Ort1`-Objekts sofort gültige Koordinaten eingetragen werden. In der Implementationsdatei `ort1.cpp` wird der Standardkonstruktor definiert mit der Wirkung, dass jedes neue `Ort1`-Objekt sofort mit den Nullpunktkoordinaten initialisiert wird:

```
Ort1::Ort1() { // neuer Standardkonstruktor
    xKoordinate = 0; // Koordinaten des Nullpunkts
    yKoordinate = 0;
}
```

Die Klassendeklaration in `ort1.h` muss im `public`-Teil noch um die Zeile `Ort1()` ergänzt werden. Die Wirkung in einem Anwendungsprogramm wird in diesem Beispiel deutlich:

```
Ort1 einOrtObjekt;
cout << einOrtObjekt.getX() << endl // 0
      << einOrtObjekt.getY();      // 0
```



Hinweis

Bei den Konstruktoren des nächsten Abschnitts können Parameter in Klammern übergeben werden. Ein Standardkonstruktor muss stets *ohne* Klammern aufgerufen werden.

```
Ort1 nochEinOrtObjekt(); // Fehler!
```

Diese Zeile wird vom Compiler nämlich nicht als Definition eines neuen Objekts, sondern als *Deklaration* einer Funktion `nochEinOrtObjekt()` verstanden, die keine Argumente braucht und ein `Ort1`-Objekt zurückgibt.

4.3.2 Allgemeine Konstruktoren

Allgemeine Konstruktoren können im Gegensatz zu Standardkonstruktoren *Argumente* haben, und sie können genau wie Funktionen *überladen* werden, das heißt, dass es mehrere allgemeine Konstruktoren mit unterschiedlichen Parameterlisten geben kann. Die zu den nachstehenden Definitionen gehörigen Prototypen sind in der Klassendeklaration nachzutragen. Wenn mindestens ein allgemeiner Konstruktor definiert worden ist, wird vom System kein Standardkonstruktor erzeugt, das heißt, es gibt keinen, wenn man ihn nicht selbst geschrieben hat. Eine versehentliche Initialisierung mit unbestimmten Daten ist damit ausgeschlossen.

```
Ort1::Ort1(int x, int y) { // Allgemeiner Konstruktor
    xKoordinate = x;
    yKoordinate = y;
}
```

Konstruktion mit Parameter- oder Initialisierungsliste

Aufruf des Konstruktors heißt Definition des Objekts `nochEinOrt`:

```
Ort1 nochEinOrt1(70, 90); // Objektdefinition = Konstruktoraufruf (Parameterliste)
Ort1 nochEinOrt2 = {70, 90}; // Alternative mit externer Initialisierungsliste {}
Ort1 nochEinOrt3 {70, 90}; // dito ohne =
```

Wenn es mehrere allgemeine Konstruktoren gibt, sucht sich der Compiler den passenden heraus, indem er Anzahl und Datentypen der Argumente der Parameterliste der Konstruktordefinition mit der Angabe im Aufruf vergleicht.

Vorgegebene Parameterwerte in Konstruktoren

Das auf Seite 113 (Abschnitt 3.2.4) beschriebene Verfahren, Parametern von Funktionen einen Wert vorzugeben, ist auf Konstruktoren übertragbar. Die vorgegebenen Werte müssen in der Deklaration angegeben werden, hier also in der Datei `ort1.h`:

```
// in ort1.h
class Ort1 {                               // Version 5
```



```
// ... wie oben
Ort1(int x, int y = 100);
};
```

Dieser Konstruktor erlaubt zum Beispiel

```
Ort1 nochEinOrt(70); // xKoordinate = 70, yKoordinate = 100!
Ort1 nochEinOrt(70, 90); // Vorgabewert wird überschrieben; oder allg. Konstruktor? (s.u.)
```

Dabei ist wie bei Funktionen darauf zu achten, dass eine Koexistenz von überladenen Konstruktoren stets zu eindeutigen Aufrufen führt. Version 5 kann nicht zusammen mit dem oben angegebenen allgemeinen Konstruktor verwendet werden, weil Aufrufe mit zwei Parametern nicht eindeutig einem der beiden zugeordnet sein können. Man kann den allgemeinen Konstruktor und den Standardkonstruktor kombinieren, indem alle Parameter mit Vorgabewerten versehen werden.

Initialisierung mit konstruktorinterner Liste

Was geschieht beim Aufruf des Konstruktors? Zunächst – noch vor Betreten des Blocks `{...}` – wird Speicherplatz für die Datenelemente `xKoordinate` und `yKoordinate` beschafft. Dann wird im zweiten Schritt der Programmcode innerhalb der geschweiften Klammern ausgeführt und die Aktualparameter werden zugewiesen. Es gibt eine Möglichkeit, beide Vorgänge in *einem* Schritt zusammenzufassen, der deshalb besonders bei größeren oder sehr vielen Objekten Laufzeitvorteile bringt. Der Weg führt über eine *Initialisierungsliste*, die noch vor dem Block angegeben und abgearbeitet wird:

```
// in ort1.h
class Ort1 { // Version 6
    // ... wie oben
    Ort1(int x, int y) // allgemeiner Konstruktor, inline
    : xKoordinate(x), yKoordinate(y) { // (interne) Initialisierungsliste
        // leerer Block
    }
};
```

Der Codeblock `{...}` kann sogar leer sein, hier bei Verzicht auf die Plausibilitätskontrolle. Die Initialisierung lässt sich auch aufteilen, indem zum Beispiel `», yKoordinate(y)«` aus der Initialisierungsliste entfernt und der Codeblock um `»yKoordinate = y;«` ergänzt würde. Die Reihenfolge der Initialisierung ist:

1. Zuerst wird die Liste abgearbeitet. Die Reihenfolge der Initialisierung richtet sich nach der Reihenfolge innerhalb der Klassendeklarationen, nicht nach der Reihenfolge in der Liste. `xKoordinate` wird zuerst initialisiert. Wenn eine Initialisierung auf dem Ergebnis einer anderen aufbaut, wäre eine falsche Reihenfolge verhängnisvoll. Um solche Fehler zu vermeiden, sollen alle Elemente der Initialisierungsliste in der Reihenfolge ihrer Deklaration aufgeführt werden.
2. Danach wird der Codeblock `{...}` ausgeführt.

Die vorgezogene Abarbeitung der Liste wird auch benutzt, um Objekte oder Größen zu initialisieren, die innerhalb des Codeblocks *konstant* sind, wie wir noch sehen werden (Abschnitt 6.2). Die Klasse `Ort` wird weiter unten gebraucht. Deswegen wird sie hier unter dem neuen Namen `Ort` (statt `Ort1`) vollständig aufgeführt.

Listing 4.4: Klasse Ort

```
// cppbuch/include/ort.h
#ifndef ORT_H
#define ORT_H
#include <cmath>           // wegen sqrt()
#include <iostream>

class Ort {
public:
    Ort(int einX = 0, int einY = 0)
        : xKoordinate(einX), yKoordinate(einY) {
    }

    int getX() const { return xKoordinate;}

    int getY() const { return yKoordinate;}

    void aendern(int x, int y) {
        xKoordinate = x;
        yKoordinate = y;
    }
private:
    int xKoordinate;
    int yKoordinate;
};

inline void anzeigen(const Ort& o) { // globale Funktion
    std::cout << '(' << o.getX() << ", " << o.getY() << ')';
}

// globale Funktion zur Berechnung der Entfernung zwischen zwei Orten
inline double entfernung(const Ort& ort1, const Ort& ort2) {
    double dx = static_cast<double>(ort1.getX() - ort2.getX());
    double dy = static_cast<double>(ort1.getY() - ort2.getY());
    return std::sqrt(dx*dx + dy*dy);
}

#endif // ORT_H
```

In der Methode `aendern()` wird auf die Kontrollausgabe verzichtet. Warum?

- Zwei verschiedene Dinge sollen nicht von derselben Methode erledigt werden.
- Eine Ausgabe (oder Eingabe) in einer Methode, die eigentlich eine andere Aufgabe hat, verhindert den universellen Einsatz. Zum Beispiel ließe sich die Methode nicht ohne Weiteres in einem System mit grafischer Benutzeroberfläche verwenden.

Eine Kontrolle sollte die Benutzung nicht beeinträchtigen und sollte daher anders realisiert werden – wie, hängt vom Anwendungsfall ab. Alle Methoden der Klasse `Ort` sind sehr kurz und der Einfachheit halber `inline` deklariert worden. Der Konstruktor definiert einen Ort (0, 0), sofern keine Koordinaten angegeben werden. Die globale Funktion `entfernung()` wird noch benötigt, und `anzeigen()` gibt die Koordinaten im Format (x, y) aus.

4.3.3 Kopierkonstruktor

Ein Kopierkonstruktor wird im Englischen *copy constructor* oder treffender *copy initializer* genannt. Er dient dazu, ein Objekt mit einem anderen zu *initialisieren*. Das erste (und im Allgemeinen einzige) Argument des Kopierkonstruktors ist eine *Referenz auf ein Objekt derselben Klasse*. Die Deklaration eines Kopierkonstruktors der Klasse `X` lautet `X(X&);`. Weil ein Objekt, das dem Kopierkonstruktor als Argument dient, nicht verändert werden soll, ist es sinnvoll, es als Referenz auf `const` zu übergeben: `X(const X&);`. Falls kein Kopierkonstruktor vorgegeben wird, wird für jede Klasse bei Bedarf vom System einer erzeugt, der die einzelnen Elemente des Objekts kopiert. Die Elemente können selbst wieder Objekte sein, deren Kopierkonstruktor dann wiederum aufgerufen wird, sei es ein selbst definierter oder der vom System bereitgestellte. Die Kopie jedes Grunddatentyps ist eine bitweise Abbildung des Speicherbereichs. Der Kopierkonstruktor der Klasse `Ort` wird wie der Standardkonstruktor in den `public`-Bereich der Klasse `Ort` geschrieben. Er ist wie folgt definiert:

```
Ort(const Ort& einOrt)      // Kopierkonstruktor
// Kopie der einzelnen Elemente:
: xKoordinate(einOrt.xKoordinate),
  yKoordinate(einOrt.yKoordinate) {
    // Anzeige des Aufrufs nur zur Demonstration
    std::cout << "Kopierkonstruktor aufgerufen\n";
}
```

Eigentlich bräuchten wir keinen eigenen Kopierkonstruktor für die schlichten Elemente der Klasse `Ort`, weil der vom System erzeugte genügen würde. Er wurde nur geschrieben, um den Aufruf auf dem Bildschirm dokumentieren zu können. *Wenn* schon ein Kopierkonstruktor mit besonderen Aktionen geschrieben wird, darf die Kopie der einzelnen Elemente nicht vergessen werden. Manchmal sind andere Operationen als die Kopie notwendig; entsprechende Beispiele werden Sie kennenlernen. Die Syntax der Initialisierung unterscheidet sich nicht vom Üblichen:

```
Ort einOrt(19, 39);
Ort derZweiteOrt = einOrt; // Aufruf des Kopierkonstruktors
// gleichwertig ist: Ort derZweiteOrt(einOrt);
cout << derZweiteOrt.getX() << ' '
      << derZweiteOrt.getY(); // 19 39
```

Diese Definition erzeugt ein Objekt `derZweiteOrt`, das mit den Werten des bereits vorhandenen Objekts `einOrt` initialisiert wird. Auch wenn das Gleichheitszeichen verwendet wird, handelt es sich hier um eine *Initialisierung* und *nicht* um eine *Zuweisung*, zwei Dinge, die in ihrer Bedeutung streng unterschieden werden.

Ein Kopierkonstruktor wird nur dann benutzt, wenn ein *neues* Objekt erzeugt wird, aber *nicht* bei Zuweisungen, also Änderungen von Objekten.

Bei *Zuweisungen* wird der vom System bereitgestellte Zuweisungsoperator benutzt, sofern kein eigener definiert wurde – auch das ist möglich, wie Sie sehen werden. Der Kopierkonstruktor wird nicht aufgerufen bei Zuweisungen oder Initialisierungen, bei denen ein temporär erzeugtes Objekt in das neu erzeugte Objekt kopiert wird. In den Folgezeilen wird daher kein Kopierkonstruktor aufgerufen.

```
Ort o1, o2;           // allg. Konstruktoren mit Vorgabewerten (0,0)
o1 = Ort(8, 7);       // allgemeiner Konstruktor + Zuweisung
o2 = o1;              // Zuweisung
Ort o3 = Ort(1, 17);  // Ein temporäres Objekt wird in das neu erzeugte Objekt kopiert.
```



Merke:

Die Übergabe von Objekten an eine Funktion per Wert und die Rückgabe eines Ergebnisobjekts wird ebenfalls als Initialisierung betrachtet, ruft also den Kopierkonstruktor implizit auf. Ausnahme: Optimierung durch den Compiler (s.u.).

Dies lässt sich am folgenden Beispiel zeigen, wobei angenommen wird, dass der Kopierkonstruktor wie oben mit einer Ausgabeanweisung versehen ist. Es gebe eine Funktion `ortsverschiebung()`, die auf einen gegebenen Ort eine bestimmte Entfernung in x- bzw. y-Richtung hinzuaddieren soll. Die Funktion greift nicht auf private Attribute eines Ortobjekts zu und braucht daher keine Elementfunktion zu sein. Diese Funktion soll innerhalb `main()` benutzt werden, zum Beispiel:

Listing 4.5: Beispiel zum Kopierkonstruktor

```
// cppbuch/k4/ortmain.cpp
#include "../include/ort.h"
using namespace std;

// Funktion zum Verschieben des Orts um dx und dy
Ort ortsverschiebung(Ort derOrt, int dx, int dy) {
    derOrt.aendern(derOrt.getX() + dx, derOrt.getY() + dy);
    return derOrt;    // Rückgabe des veränderten Orts
}

int main() {
    Ort einOrt(10, 300);
    Ort verschobenerOrt = ortsverschiebung(einOrt, 10, -90);
    cout << "alter Ort: ";
    anzeigen(einOrt);
    cout << "\n neuer Ort: ";
    anzeigen(verschobenerOrt);
}
```

Der oben definierte Kopierkonstruktor wird zweimal aufgerufen (falls der Aufruf nicht wegoptimiert wird, siehe unten): das erste Mal bei der Übergabe des Objektes `einOrt` an die Funktion und das zweite Mal während der Ausführung der `return`-Anweisung. Hier wird besonders deutlich, dass die Übergabe per Referenz auf `const` einiges an Geschwindigkeitsgewinn bringen kann, falls man keine Objektkopie in der Funktion benötigt. Im obigen Beispiel ist allerdings die Übergabe per Wert erforderlich, weil eine Objektkopie zum Aufruf von `aendern()` benötigt wird, ohne dass `einOrt` verändert wird.

Optimierung durch den Compiler

Im obigen Beispiel wird ein temporäres Objekt erzeugt, mit dem die Variable `verschobenerOrt` initialisiert wird. Dies kostet Zeit und Speicherplatz. Die Initialisierung selbst

benötigt ebenfalls Zeit, ebenso die anschließende »Entsorgung« des temporären Objekts, das nicht weiter benötigt wird. Deshalb ist es Compilern unter bestimmten Bedingungen erlaubt, die Erzeugung temporärer Objekte zu vermeiden. Bezogen auf das obige Beispiel könnte der Compiler das zurückzugebende Objekt `derOrt` der Funktion `ortsverschiebung()` *direkt* an die Stelle von `verschobenerOrt` schreiben, statt den Kopierkonstruktor aufzurufen. Dies gilt auch, wenn es Seiteneffekte gibt (Abschnitt 12.8 in [ISOC++]). Wenn etwa der Kopierkonstruktor seine Aufrufe protokolliert, kann es ein, dass die Protokollausgaben im Fall der Optimierung fehlen, weil der Kopierkonstruktor nicht aufgerufen wird.



Mehr dazu lesen Sie in Abschnitt 22.1.

4.3.4 Typumwandlungskonstruktor

Der Typumwandlungskonstruktor dient zur Umwandlung anderer Datentypen in die gewünschte Klasse. Das erste Argument des Typumwandlungskonstruktors ist verschieden vom Typ der Klasse. Falls weitere Argumente folgen, was im Allgemeinen nicht der Fall sein wird, müssen sie Initialisierungswerte haben. Im Grunde ist der Typumwandlungskonstruktor nichts anderes als der Spezialfall eines allgemeinen Konstruktors, der etwas anders eingesetzt wird.

Hier wird ein Typumwandlungskonstruktor gezeigt, der als String vorliegende Ortsangaben in ein Ort-Objekt verwandelt. Das Format ist so gewählt, dass zwei getrennte Folgen von Ziffern als x- und y-Koordinaten interpretiert werden. Andere Zeichen werden ignoriert. Damit sind Schreibweisen wie "100 200" möglich, aber auch "(100,200)".

```
// zusätzlich benötigte Include-Anweisungen in ort.h:
#include<cctype> // isdigit()
#include<string>
#include<cassert>

// im public-Bereich von ort.h (Seite 157) einfügen:
// Typumwandlungskonstruktor. Format: 2 Folgen von Ziffern
Ort(const std::string& str) {
    unsigned int pos = 0; // Position einer Ziffer im String str
    for(int j = 0; j < 2; ++j) { // für jede Koordinate
        while(pos < str.size() && !isdigit(str.at(pos))) // erste Ziffer suchen
            ++pos;
        assert(pos < str.size()); // Ziffer gefunden? Abbruch, falls nicht
        // Zahl bilden
        int koordinate = 0;
        while(pos < str.size() && isdigit(str.at(pos))) {
            // implizite Typumwandlung char → int
            koordinate = 10*koordinate + str.at(pos) - '0';
            ++pos;
        }
        switch(j) {
            case 0: xKoordinate = koordinate; break;
            case 1: yKoordinate = koordinate;
        }
    }
}
```

Ein Beispiel zeigt die Anwendung:

```
Ort nochEinOrt(string("21 99")); // mögliches Format
anzeigen(nochEinOrt);
cout << endl;
Ort einWeitererOrt("(55, 8)"); // weiteres mögliches Format
anzeigen(einWeitererOrt);
```

Im zweiten Fall findet der Compiler eine in " " eingeschlossene Folge von Zeichen. Die Umwandlung in ein String-Objekt nimmt der Compiler automatisch vor, die Typprüfung des Compilers wird eingeschränkt. Die Einschränkung wird hier besonders deutlich:

```
string wo("20,400");
Ort hier = ortsverschiebung(wo, 10, -90);
// besser: explizite Angabe der Typumwandlung (s.u.)
Ort dort = ortsverschiebung(Ort(wo), 10, -90);
```

Die Umwandlung für das Objekt `hier` funktioniert, obwohl es keine überladene Funktion `ortsverschiebung()` mit einem String-Objekt als erstem Parameter gibt. Man kann sich vorstellen, dass der Compiler intern, das heißt, ohne dass wir etwas davon mitbekommen, eine temporäre Variable erzeugt, die die Umwandlung vornimmt:

```
// Erzeugen eines temporären Ort-Objekts __temp:
Ort __temp(wo);           // Typumwandlungskonstruktor
Ort hier = ortsverschiebung(__temp, 10, -90);
// Ab hier kann das temporäre Objekt zerstört werden.
```

Die Variable `__temp` ist temporär und existiert nur als Hilfsvariable des Compilers, sodass sie später nicht weiter stört. An die Funktion wird also ein temporäres Objekt mit dem richtigen Datentyp übergeben.



Tipp

Im Allgemeinen möchte man sowohl die Möglichkeit der Typumwandlung haben als auch die Typprüfung durch den Compiler, damit keine Fehler durch implizite Typumwandlungen entstehen. Das Schlüsselwort `explicit` erlaubt es, explizite Typumwandlungen durchzuführen, aber andere, vielleicht unbeabsichtigte, zu verbieten.

```
class Ort {
public:
    // Typumwandlungskonstruktor mit explicit
    explicit Ort(const std::string& str);
    // ... Rest wie vorher
};

Ort o1;
string wo("10, 200");
o1 = wo;           // jetzt ein Fehler!
o1 = Ort(wo);      // erlaubte explizite Typumwandlung
```

Ein weiteres Beispiel für den Einsatz eines Typumwandlungskonstruktors wird im nächsten Abschnitt gezeigt.

4.4 Beispiel: Rationale Zahlen

4.4.1 Aufgabenstellung

Es folgt ein vollständiges Beispiel für eine Klasse, mit deren Objekten gerechnet werden kann. Es soll eine Bibliothek zum Rechnen mit rationalen Zahlen programmiert werden. Rationale Zahlen sind Zahlen, die durch einen Bruch darstellbar sind, wobei Zähler und Nenner ganzzahlig sein müssen. Dabei soll ein Anwender rationale Zahlen mit dem Datentyp `Rational` definieren können. Mit diesen Zahlen sollen alle Grundrechenarten ohne Genauigkeitsverlust durchführbar sein. Alle Ergebnisse von Rechnungen mit rationalen Zahlen sollen in bereits gekürzter Form vorliegen. Negative Zahlen sind durch einen negativen Zähler zu repräsentieren.

Benutzungsschnittstelle

Ein die Klasse benutzendes Programm muss die Datei *rational.h* per `#include`-Anweisung einlesen. Die Objektdatei *rational.o* muss eingebunden (»gelinkt«) werden. Die Definition einer rationalen Zahl `r` soll wie eine übliche Variablendeklaration geschrieben werden, zum Beispiel: `Rational r;`.

Die folgenden Funktionen sollen von der Bibliothek bereitgestellt werden, wobei die Operanden `a` und `b` vom Typ `Rational`, `int` oder `long int` sein können. Im Kapitel 9 wird gezeigt, wie man diese Funktionen direkt durch die üblichen mathematischen Operatoren ersetzen kann.

```
r.eingabe();      Dialogeingabe der rationalen Zahl r
r.ausgabe();      Dialogausgabe der rationalen Zahl im Format Zähler/Nenner
r.getZaehler();   Zähler zurückgeben
r.getNenner();    Nenner zurückgeben
r.set( z, n);     Setzen der Werte für Zähler und Nenner
r.kehrwert();     r enthält danach den Kehrwert
```

Rechenfunktionen:

```
r = add(a, b);    r = a + b
r = sub(a, b);    r = a - b
r = mult(a, b);   r = a * b
r = div(a, b);    r = a / b
```

Kurzformoperationen:

```
r.add(a);         r += a
r.sub(a);         r -= a
r.mult(a);        r *= a
r.div(a);         r /= a
```

Beschränkungen und Hinweise

Nach jeder Operation sollen die Zahlen gekürzt werden. Eine Bereichsüberprüfung ist der Einfachheit halber nicht vorgesehen. Anwender sind demnach selbst dafür verantwortlich, dass der für den Datentyp `long` zutreffende Bereich ihres Rechners in Zwischenrechnungen nicht überschritten wird. In einer kommerziellen Version sollte dieser Fall we-

nigstens zu einer Fehlermeldung führen. Falls der Nenner null wird, soll das Programm mit einer Fehlermeldung abbrechen.

4.4.2 Entwurf

In diesem Abschnitt werden die Algorithmen, die die mathematische Grundlage des Programms bilden, sowie einige Überlegungen zur Implementierung dargestellt. Für die folgenden arithmetischen Ausdrücke gilt :

$x.z$ ist der Zähler der rationalen Zahl x

$x.n$ ist der Nenner der rationalen Zahl x

$$\text{Addition} \quad r = a + b : \quad r = \frac{a.z * b.n + b.z * a.n}{a.n * b.n}$$

$$\text{Subtraktion} \quad r = a - b : \quad r = \frac{a.z * b.n - b.z * a.n}{a.n * b.n}$$

$$\text{Multiplikation} \quad r = a * b : \quad r = \frac{a.z * b.z}{a.n * b.n}$$

$$\text{Division} \quad r = a / b : \quad r = \frac{a.z * b.n}{a.n * b.z}$$

Kehrwert bilden $1/r$: Vertauschen von Zähler und Nenner

Die Rechenregeln für die Kurzformoperatoren sind entsprechend. Im Programm können Vereinfachungen vorgenommen werden etwa der Art, dass die Subtraktion auf die Addition einer negativen Zahl zurückgeführt wird.

Für den Datentyp der rationalen Zahl wird die Klasse `Rational` deklariert, die als private Attribute nur den Zähler und den Nenner hat. Die in der Anforderungsdefinition vorgeschriebenen Methoden werden in die Klasse übernommen. Hinzu kommt noch die Methode `kuerzen()`, die intern benötigt wird. Beim Betrachten der Operationen ist festzustellen:

- Eine Operation wie `r.add(a)` ändert das Objekt `r`, aber nicht das Objekt `a`. Weil Objekte nur über Methoden geändert werden können (Abstrakter Datentyp!), ist `add()` eine Methode oder Elementfunktion. Der Parameter `a` kann per Wert oder per Referenz auf `const` übergeben werden. Letztes spart das Erzeugen einer lokalen Kopie, lohnenswert bei großen Objekten. Ein Objekt des Typs `Rational` ist jedoch relativ klein, weswegen dieser Punkt hier keine große Rolle spielt. Die Funktion `add()` gibt nichts zurück, der Rückgabetypp ist `void`. Eine andere Möglichkeit für den Rückgabetypp wird in Punkt 6 auf Seite 559 diskutiert.
- Eine Operation wie `r = add(a, b)` ändert nicht die Parameter `a` und `b`. Für die Parameterübergabe gelten daher die Argumente des vorhergehenden Punktes. Die Funktion `add(a, b)` erzeugt eine rationale Zahl als Ergebnis, die der Variablen `r` zugewiesen wird. Der Rückgabetypp ist also `Rational`. Da die Funktion wegen der Funktionen `getZaehler()` und `getNenner()` nicht auf private Attribute zugreifen muss, kann sie global sein.

Weil Operationen mit gemischten Datentypen möglich sein sollen, muss ein Mechanismus dafür bereitgestellt werden. Zwei Möglichkeiten sind üblich:

1. Die arithmetischen Methoden und Funktionen können überladen werden, sodass für die beiden Operanden folgende Kombinationen erlaubt sind:

Methode, zum Beispiel

```
void Rational::add(const Rational&)
```

```
void Rational::add(long)
```

und globale Funktion, zum Beispiel

```
Rational add(const Rational&, const Rational&)
```

```
Rational add(const Rational&, long)
```

```
Rational add(long, const Rational&)
```

int-Werte würden implizit nach long konvertiert werden. Pro globaler Rechenoperation würden drei Methoden anstatt einer benötigt, die allerdings teilweise aufeinander zugreifen könnten.

2. Wenn die Anzahl der Methoden und Funktionen klein gehalten werden soll, muss dem Compiler eine Möglichkeit zur Typumwandlung zur Verfügung gestellt werden, wenn er die Daten an die Rechenfunktionen übergibt. Dies geschieht am günstigsten durch einen Typumwandlungskonstruktor, wie er im letzten Abschnitt behandelt wurde. Er darf hier natürlich *nicht* explicit sein!

In diesem Fall wurde die Entscheidung für die zweite Variante getroffen. Die Klassendeklaration in der Datei *rational.h* lautet:

Listing 4.6: Klasse Rational

```
// cppbuch/k4/ratio/rational.h Klasse für rationale Zahlen
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
public:
    Rational();
    Rational(long z, long n); // allgemeiner Konstruktor
    Rational(long);           // Typumwandlungskonstruktor

    // Abfragen
    long getZaehler() const;
    long getNenner() const;

    // arithmetische Methoden für +=, -=, *=, /=,
    // (werden später durch überladene Operatoren ergänzt)
    void add(const Rational& r);
    void sub(const Rational& r);
    void mult(const Rational& r);
    void div(const Rational& r);

    // weitere Methoden
    void set(long zaehler, long nenner);
    void eingabe();
    void ausgabe() const; // siehe dazu Seite 170
    void kehrwert();
```

```

    void kuerzen();
private:
    long zaehler, nenner;
};

// inline-Methoden
inline Rational::Rational() : zaehler(0), nenner(1) {}

inline Rational::Rational(long z, long n)
: zaehler(z), nenner(n) {}

inline Rational::Rational(long ganzeZahl)
: zaehler(ganzeZahl), nenner(1) {}

inline long Rational::getZaehler() const {return zaehler;}
inline long Rational::getNenner() const {return nenner;}

// globale Funktionsprototypen
const Rational add(const Rational& a, const Rational& b);
const Rational sub(const Rational& a, const Rational& b);
const Rational mult(const Rational& a, const Rational& b);
const Rational div(const Rational& z, const Rational& n);
#endif

```

Die Methode `kuerzen()` wird implizit von den anderen Methoden aufgerufen und könnte daher privat sein. Die `public`-Eigenschaft schadet aber nicht.



Hinweis

Warum haben die globalen Funktionsprototypen einen `const`-Spezifizierer? Wenn `const` fehlte, könnte das zurückgegebene Objekt als L-Wert (englisch *lvalue*) benutzt werden, weil es veränderbar ist. Mit anderen Worten, es könnte auf der linken Seite einer Zuweisung stehen (vgl. Seite 62), etwa so: `add(a, b) = c;`. Natürlich ist die Zuweisung von `c` zu dem temporären Ergebnis unsinnig, der Compiler würde die Anweisung aber akzeptieren. Mit `const` gibt der Compiler an dieser Stelle eine Fehlermeldung aus.

Fehlerbetrachtung

Ein Überlauf des Zahlenbereichs soll nicht geprüft werden, wohl aber der Fall, dass ein Nenner 0 wird. Dies kann in den Methoden `eingabe()`, `kehrwert()`, `set()` und `div()` geschehen, wobei Letztere nicht betrachtet werden muss, wenn man die Division durch die Multiplikation mit dem Kehrwert implementiert. Mit Hilfe von `assert()` wird geprüft, ob der Nenner 0 ist. Bei der Eingabe ist dieses Vorgehen sicher nicht sehr benutzerfreundlich und könnte daher modifiziert werden.

4.4.3 Implementation

Die Implementation der Methoden, globalen Funktionen einschließlich der Hilfsfunktion `ggt()` zum Finden des größten gemeinsamen Teilers¹, um einen Bruch zu kürzen, sind in der Datei *rational.cpp* abgelegt.

Listing 4.7: Implementation der Klasse Rational

```
// cppbuch/k4/ratio/rational.cpp (Definition der Methoden und globalen Funktionen)
#include "rational.h"
#include <iostream>
#include <cassert>
using namespace std;

void Rational::set(long z, long n) {
    zaehler = z;
    nenner = n;
    assert(nenner != 0);
    kuerzen();
}

void Rational::eingabe() {
    // Bildschirmausgabe nur zu Demonstrationszwecken.
    // cerr wird gewählt, damit die Abfragen auch dann auf dem Bildschirm erscheinen,
    // wenn die Standardausgabe in eine Datei zur Dokumentation geleitet wird.
    cerr << "Zähler :";
    cin >> zaehler;
    cerr << "Nenner :";
    cin >> nenner;
    assert(nenner != 0);
    kuerzen();
}

void Rational::ausgabe() const {
    cout << zaehler << '/' << nenner << endl;
}

void Rational::kehrwert() {
    long temp = zaehler;
    zaehler = nenner;
    nenner = temp;
    assert(nenner != 0);
}

void Rational::add(const Rational& r) {
    zaehler = zaehler*r.nenner + r.zaehler*nenner;
    nenner = nenner*r.nenner;
    kuerzen();
}

void Rational::sub(const Rational& s) {
    Rational r = s;
```

¹ Die erste, langsamere Fassung des Algorithmus wird auf Seite 70 gezeigt.

```

    // Das temporäre Objekt r wird benötigt, weil s wegen der const-Eigenschaft nicht
    // verändert werden kann (und darf). Die eigentlich bessere Alternative wäre die Übergabe
    // per Wert – dann könnte mit der lokalen Kopie gearbeitet werden (Begründung siehe
    // nachfolgender Text). Die Subtraktion kann durch Addition des negativen Arguments
    // erreicht werden.
    r.zaehler *=-1;
    add(r);
}

void Rational::mult(const Rational& r) {
    zaehler = zaehler*r.zaehler;
    nenner = nenner *r.nenner;
    kuerzen();
}

void Rational::div(const Rational& n) {
    Rational r = n; // siehe Diskussion bei sub()
    r.kehrwert();
    mult(r); // Division = Multiplikation mit dem Kehrwert
}

// Die globale Funktion ggt() wird zum Kürzen benötigt. Sie berechnet den größten
// gemeinsamen Teiler. Verwendet wird ein modifizierter Euklid-Algorithmus, in dem
// Subtraktionen durch die schnellere Restbildung ersetzt werden.
long ggt(long x, long y) {
    long rest;
    while(y > 0) {
        rest = x % y;
        x = y;
        y = rest;
    }
    return x;
}

void Rational::kuerzen() {
    // Vorzeichen merken und Betrag bilden
    int sign = 1;
    if(zaehler < 0) { sign = -sign; zaehler = -zaehler; }
    if(nenner < 0) { sign = -sign; nenner = -nenner; }
    long teiler = ggt(zaehler, nenner); // siehe oben
    zaehler = sign*zaehler/teiler; // Vorzeichen restaurieren
    nenner = nenner/teiler;
}

// Es folgen die globalen arithmetische Funktionen für die
// Operationen mit 2 Argumenten (binäre Operationen)

const Rational add(const Rational& a, const Rational& b) {
    // Die eigentliche Berechnung muss hier nicht wiederholt werden, sondern die bereits
    // vorhandenen Funktionen für die Kurzformen der Addition usw. können vorteilhaft
    // wiederverwendet werden. Dazu wird ein mit a initialisiertes temporäres Objekt
    // erzeugt, auf das das Argument b addiert und das dann als Ergebnis
    // zurückgegeben wird. Zum temporären Objekt r siehe auch die Diskussion bei der

```

```

// Elementfunktion sub() oben.
Rational r = a;
r.add(b);
return r;
}

const Rational sub(const Rational& a, const Rational& b) {
    Rational r = a;
    r.sub(b);
    return r;
}

const Rational mult(const Rational& a, const Rational& b) {
    Rational r = a;
    r.mult(b);
    return r;
}

const Rational div(const Rational& z, const Rational& n) {
    Rational r = z;
    r.div(n);
    return r;
}

```

Warum ist die Übergabe per Wert bei der Funktion `sub()` besser? Antwort: Falls der Parameter ein temporäres Objekt ist, hat der Compiler die Chance, den Aufruf des Kopierkonstruktors zu eliminieren (siehe Seite 159 unten). So kann sich gelegentlich erst bei dem Entwurf der Implementierung herausstellen, dass eine andere Schnittstelle günstiger ist, hier `sub(Rational)` statt `sub(const Rational&)`. `Rational`-Objekte sind klein; insofern macht der Unterschied hier nicht viel aus und die Schnittstelle wird beibehalten.

Der Einsatz des Typumwandlungskonstruktors vereinfacht das Programm durch die Reduktion der Methodenanzahl, führt aber zu einem kleinen Effizienzverlust, weil zur Laufzeit einige Schritte mehr ausgeführt werden müssen. Solange ein Programm nicht zeitkritisch ist, ist es immer sinnvoll, der Einfachheit den Vorzug zu geben und nicht der Geschwindigkeit.



Übungen

4.1 Schreiben Sie die Funktionen `add(long a, Rational b)` und `add(Rational a, long b)`, die bei *Abwesenheit* des Typumwandlungskonstruktors erforderlich wären.

4.2 Schreiben Sie eine Funktion, die ein Objekt vom Typ `Rational` übergeben bekommt und dasselbe tut wie die Funktion `Rational::ausgabe()`, ohne auf private Daten zuzugreifen.

Testdokumentation

Um die Klasse `Rational` zu testen, wurde die Datei `cppbuch/k4/ratio/main.cpp` geschrieben, die `rational.h` einbindet. Die aus `rational.cpp` durch Compilation entstandene Datei `rational.o` muss dazu gelinkt werden. Das ausführbare Programm `a.out` bringt die Test-

ausgaben auf den Bildschirm. Mit der Anweisung `a.out > test.erg` werden alle Ausgaben in die Datei `test.erg` geschrieben.

Ergebnisse des Testprogramms

Die Testergebnisse werden hier aus Platzgründen und weil sie überaus langweilig zu lesen sind, nicht abgedruckt. Probieren Sie das Testprogramm aus und erweitern Sie es um zusätzliche Prüfungen, um Fehlern auf die Spur zu kommen!

Listing 4.8: Testprogramm

```
// cppbuch/k4/ratio/main.cpp   Testprogramm für Klasse Rational (Auszug)
#include "rational.h"
#include <iostream>
using namespace std;

// alle 4 Operationen für a und b
void druckeTestfall(const Rational& a, const Rational& b) {
    Rational erg;
    cout << "a = "; a.angabe();
    cout << "b = "; b.angabe();
    // Die Elementfunktionen werden implizit mitgetestet.
    erg = add(a,b);
    cout << "+ : "; erg.angabe();
    erg = sub(a,b);
    cout << "- : "; erg.angabe();
    erg = mult(a,b);
    cout << "* : "; erg.angabe();
    erg = div(a,b);
    cout << "/ : "; erg.angabe();
    cout << endl;
}

int main() {
    Rational a,b;
    cout << "Test der Eingabe\n";
    a.eingabe();
    b.eingabe();
    druckeTestfall(a,b);
    cout << "\n Test mit verschiedenen Vorzeichen\n";
    a.set(3,7);
    b.set(6,13);
    druckeTestfall(a,b);
    a.set(3,-7);
    druckeTestfall(a,b);
    //...und so weiter
    cout << "\n Test mit gemischten Datentypen\n";
    a.set(2301,77777);
    druckeTestfall(a,17);
    druckeTestfall(17, a);
    //...und so weiter
    cout << "\n Test mit Nullwerten\n"; // ...und noch mehr
}
```

4.5 const-Objekte und Methoden

Objekte können wie einfache Variablen als *konstant* deklariert werden. Um jegliche Änderungen zu vermeiden, dürfen Methoden von konstanten Objekten nicht aufgerufen werden (ausgenommen Konstruktoren und Destruktoren), es sei denn, sie sind als konstante Elementfunktionen deklariert und definiert. Nehmen wir an, wir würden Bezugnehmend auf das Beispiel auf den Seiten 162 ff. eine konstante rationale Zahl definieren: `const Rational CR;`. Nehmen wir ferner an, wir hätten nach der Deklaration der Methode `ausgabe()` das Wort `const` vergessen. Der Aufruf `CR.ausgabe()`; rief dann eine Warnung oder Fehlermeldung des Compilers hervor. Wenn aber in der Deklaration und Definition das Schlüsselwort `const` angegeben wird, erhält die damit ausgezeichnete Methode das Privileg, für konstante (*und* nichtkonstante) Objekte aufgerufen werden zu können:

```
void ausgabe() const;           // Deklaration
```

```
void Rational::ausgabe() const { // Definition
    //.... wie vorher
}
```

Ein konstantes Objekt kann nicht durch `const`- oder andere Funktionen geändert werden, selbst dann nicht, wenn es per Referenz übergeben wird. Von dieser Regel gibt es zwei Ausnahmen:

1. Die `const`-Eigenschaft kann durch eine explizite Typumwandlung umgangen werden (englisch *casting the const away*) (siehe dazu Abschnitt 7.9).
2. In einer Klasse können Variablen mit dem Schlüsselwort `mutable` versehen werden. Es ist erlaubt, diese Attribute durch eine Methode zu ändern, auch wenn das Objekt konstant ist, zu dem das Attribut gehört. Der Sinn des Schlüsselworts liegt darin, dass eine Implementation sicherstellen will, dass einerseits Objekte nicht geändert werden, andererseits ein schneller Zugriff möglich sein soll, was die Änderung interner Verwaltungsinformationen erfordert. Beispielsweise könnte man sich in einer konstanten Liste die zuletzt benutzte Position für einen Zugriff merken (Stichwort »cache«), um beim nächsten Zugriff schnell zu sein.

Ein `const`-Qualifizierer wird auch beim Überladen von Methoden ausgewertet. Man kann *zwei* Methoden mit gleicher Parameterliste, aber verschiedener Wirkung, schreiben, die sich nur durch `const` unterscheiden:

```
void ausgabe();           // Methode 1
void ausgabe() const;     // Methode 2
```

In der Anwendung ruft der Compiler je nach Eigenschaft des Objekts die eine oder die andere Methode auf:

```
Rational r(6, 7);
const Rational CR(8, 9);
r.ausgabe();           // ruft Methode 1
CR.ausgabe();          // ruft Methode 2
```



Übung

4.3 Schreiben Sie eine Klasse `IntMenge`, bestehend aus den zwei Dateien `IntMenge.h` und `IntMenge.cpp`, sowie ein Testprogramm `main.cpp` entsprechend den Regeln dieses Kapitels. Die Klasse soll als mathematische Menge für ganze Zahlen nachbilden. Es sollen nur die folgenden einfachen Funktionen möglich sein, auf Operationen mit zwei Mengen wie Vereinigung und Durchschnitt werde verzichtet:

- `void hinzufuegen(int el)`: Element `el` hinzufügen, falls es noch nicht existiert, andernfalls nichts tun.
- `void entfernen(int el)`: Element `el` entfernen, falls es vorhanden ist, andernfalls nichts tun.
- `bool istMitglied(int el)`: Gibt an, ob `el` in der Menge enthalten ist.
- `size_t size()`: Gibt die Anzahl der gespeicherten Elemente zurück.
- `void anzeigen()`: Gibt alle Elemente auf der Standardausgabe aus.
- `void loeschen()`: Alle Elemente löschen.
- `int getMax()` und `int getMin()`: Geben das größte bzw. kleinste Element zurück.

Benutzen Sie intern zum Speichern der Werte ein `vector<int>`-Objekt (vgl. Abschnitt 1.9.2). Ein Auszug einer Anwendung könnte etwa wie folgt aussehen:

```
IntMenge menge;
menge.hinzufuegen(2); // ok
menge.hinzufuegen(-9); // ok
menge.hinzufuegen(2); // keine Wirkung, 2 gibt es schon
menge.entfernen(99); // keine Wirkung, nicht vorhanden
menge.entfernen(-9); // ok
menge.anzeigen();
menge.loeschen();
for(int i=17; i < 33; ++i) {
    menge.hinzufuegen(i*i);
}
cout << "Anzahl=" << menge.size() << " Minimum=" << menge.getMin();
if(menge.istMitglied(-11)) {
    // ... usw.
```

Diese Aufgabe ist eine Vorübung für das Thema einer Klasse »Menge«. Die C++-Bibliothek stellt für Mengen die Klasse `set` zur Verfügung, die in Abschnitt 28.3.3 beschrieben wird.

4.6 Destruktoren

Destruktoren dienen dazu, Aufräumarbeiten für nicht mehr benötigte Objekte zu leisten. Wenn Destruktoren nicht vorgegeben werden, werden sie vom System automatisch erzeugt (implizite Deklaration). Der häufigste Zweck ist die Speicherfreigabe, wenn der Gültigkeitsbereich eines Objekts verlassen wird. Konstruktoren haben die Aufgabe, Res-

sources zu beschaffen, Destruktoren obliegt es, sie wieder freizugeben. Die Reihenfolge des Aufrufs der Destruktoren ist *umgekehrt* wie die der Konstruktoren.

Destruktoren haben keine Argumente und keinen Rückgabotyp. In der Deklaration wird eine Tilde ~ vorangestellt. Im Beispiel werden nummerierte Testobjekte erzeugt. Um den Ablauf verfolgen zu können, sind Konstruktor und Destruktor mit Ausgabeanweisungen versehen. Die Gültigkeit oder Lebensdauer eines Objekts endet, wie schon aus Abschnitt 1.7 bekannt, an der durch eine schließende geschweifte Klammer markierten Grenze des Blocks, in dem das Objekt definiert wurde. Genau dann wird das Objekt zerstört, das heißt, dass der von diesem Objekt belegte Speicherplatz freigegeben wird.

Daraus folgt, dass durchaus außerhalb von `main()` einige Aktivitäten stattfinden können:

- Falls es globale Objekte gibt, wird ihr Konstruktor *vor* der ersten Anweisung von `main()` aufgerufen.
- Innerhalb des äußersten Blocks von `main()` definierte Objekte werden erst beim Verlassen von `main()` freigegeben.
- Wegen der umgekehrten Reihenfolge der Destruktoraufrufe werden globale Objekte zuletzt freigegeben.

Die Ausgabe des Programms belegt, dass die Objekte *nach* der letzten Anweisung ihres Blocks zerstört werden.

Listing 4.9: Wirkung des Destruktors

```
// cppbuch/k4/destrukt.cpp
#include<iostream>
using namespace std;

class Beispiel {
    int zahl;                // zur Identifizierung

public:
    Beispiel(int i = 0);      // Konstruktor
    ~Beispiel();              // Destruktor
};

Beispiel::Beispiel(int i)    // Konstruktor
: zahl(i) {
    cout << "Objekt " << zahl << " wird erzeugt.\n";
}

Beispiel::~~Beispiel() {     // Destruktor
    cout << "Objekt " << zahl << " wird zerstört.\n";
}

// globale Variable, durch Vorgabewert mit 0 initialisiert
Beispiel ein_globales_Beispiel;

int main() {
    cout << "main wird begonnen\n";
    Beispiel einBeispiel(1);
    {
        cout << "    neuer Block\n    ";
```

```
        Beispiel einBeispiel(2);  
        cout << "    Block wird verlassen\n    ";  
    }  
    cout << "main wird verlassen\n";  
}
```

Die Ausgabe des Programms ist:

```
Objekt 0 wird erzeugt.  
main wird begonnen  
Objekt 1 wird erzeugt.  
    neuer Block  
    Objekt 2 wird erzeugt.  
    Block wird verlassen  
    Objekt 2 wird zerstört.  
main wird verlassen  
Objekt 1 wird zerstört.  
Objekt 0 wird zerstört.
```

Der Destruktor statischer Objekte (static oder globale Objekte) wird nicht nur beim Verlassen eines Programms mit `return`, sondern auch bei Verlassen mit `exit()` aufgerufen. Im Gegensatz zum normalen Verlassen eines Blocks wird der Speicherplatz bei `exit()` jedoch nicht freigegeben.

4.7 Wie kommt man zu Klassen und Objekten? Ein Beispiel

Es kann hier keine allgemeine Methode gezeigt werden, wie man von einer Aufgabe zu Klassen und Objekten kommen kann. Es wird jedoch anhand eines Beispiels ein erster Eindruck vermittelt, wie man von einer Problemstellung zum objektorientierten Programm kommen kann.

Simulation der Benutzung eines einfachen Getränkeautomaten

In der Kantine stehen Getränkeautomaten, unter anderem einer mit *ObjektCola*. Der Automat sei so eingestellt, dass eine Dose dieses Getränks 2 € kostet. Der Automat nimmt einen beliebigen Geldbetrag an. Wenn zu wenig eingeworfen wird, wird das eingeworfene Geld wieder herausgegeben. Wenn zu viel eingeworfen wird, wird eine Dose *ObjektCola* sowie der Restbetrag herausgegeben. Nach Geldeinwurf löst ein Knopfdruck die Prüfung des Geldbetrags und gegebenenfalls die Ausgabe einer Dose aus. Wenn keine Dose mehr vorrätig ist, ist der Automat gesperrt, das heißt, dass jeder eingeworfene Geldbetrag vollständig zurückgegeben wird. Das folgende, bewusst einfach gehaltene Szenario soll mit einem Programm simuliert werden:



Szenario

Der Automat wird anfangs mit 50 Dosen befüllt. Zum Automaten gehen Personen, die eine Dose aus dem Automaten ziehen wollen. Sie tragen unterschiedliche Geldbeträge bei sich. Wenn der Betrag ausreicht, werfen sie Münzen in den Automaten, wobei manche nicht genau zählen und zufällig zu viele oder zu wenige Münzen einwerfen. Anschließend drücken sie auf den Ausgabeknopf mit der Wirkung, dass gegebenenfalls eine Dose und Rückgeld ausgegeben werden. Das Szenario endet, wenn der Automat gesperrt wird, weil er leer ist.

Das Programm soll zur Kontrolle die einzelnen Schritte auf dem Bildschirm dokumentieren. Der Einfachheit halber genügt es, einheitliche Münzen anzunehmen, zum Beispiel 1-€-Stücke.

4.7.1 Einige Analyse-Überlegungen

Um die Problemstellung zu verdeutlichen, wird sie aus verschiedenen Blickwinkeln betrachtet. Es handelt sich dabei nur um *Möglichkeiten*, nicht um den einzig wahren Lösungsansatz (den es nicht gibt).

1. In der Analyse geht es zunächst einmal darum, Prozesse *in der Sprache des (späteren Programm-) Anwenders* zu beschreiben. Dabei sind typische Abläufe, Szenarien genannt, ein gutes Hilfsmittel. Die Aufgabenstellung ist als Szenario formuliert.
2. Im zweiten Schritt wird versucht, beteiligte Objekte, ihr Verhalten und ihr Zusammenwirken zu identifizieren. Dies ist nicht unbedingt einfach, weil spontan identifizierte Beziehungen zwischen Objekten im Programm nicht immer die wesentliche Rolle spielen. Auf einem geeigneten Abstraktionsgrad muss entschieden werden, was zum »System« und was zur »Außenwelt« gehören soll.

Strukturierung des Ablaufs

Das Szenario beschreibt einen *Vorgang*, dessen einzelne Schritte als Pseudocode strukturiert dargestellt werden können.

Anfang des Szenarios

- 01 Automat mit 50 Dosen füllen.
- 02 Solange der Automat nicht gesperrt, das heißt nicht leer ist, wiederhole:
- 03 Eine durstige Person (z.B. eine Studentin) kommt vorbei, sie hat X € dabei.
- 04 X steht für eine zufällige Zahl.
- 05 Falls sie genug Geld hat (d.h. mindestens den Preis pro Dose).
- 06 steckt sie eine Anzahl Y Münzen in den Münzschlitz
- 07 und drückt auf den Knopf.
- 08 Falls Rückgeld ausgegeben wurde,
- 09 nimmt sie es.
- 10 Falls eine Dose ausgegeben wurde,
- 11 nimmt sie sie und
- 12 trinkt sie aus.

- 13 Andernfalls stellt sie fest: »Zu wenig Geld!«.
- 14 Die Person verlässt die Szene
(gegebenenfalls nicht mehr durstig und mit weniger Geld).

Ende des Szenarios

Objekte und Operationen identifizieren

Im nächsten Schritt wird versucht, die beteiligten Objekte und damit ihre Klassen zu identifizieren und eine Beschreibung ihres Verhaltens zu finden. Dazu wird auf die einzelnen nummerierten Zeilen des Szenarios Bezug genommen, die *kursiv* dargestellt sind. Bezeichnungen in dieser Schrift deuten auf *mögliche* Klassen- und Methodennamen. Es ist nicht nur das aktive Verhalten wichtig, sondern manchmal benötigt man außerhalb des Objekts Informationen über seinen inneren Zustand.

01 Automat mit 50 Dosen füllen.

Der Automat ist ein *GetraenkeAutomat*. Er enthält Dosen, deren Anzahl uns interessiert. Die Anzahl ist eine Eigenschaft des Automaten, nicht der Dose. Andere Automaten können gleichartige Dosen in anderer Stückzahl enthalten. Unklar ist hier, *wer* den Automaten füllt. Da wir uns in diesem Szenario nicht dafür interessieren, liegt die Antwort außerhalb unseres »Systems«. Wir müssen nur dafür sorgen, dass der Automat am Anfang des Szenarios ein paar Dosen enthält.

02 Solange der Automat nicht gesperrt, das heißt leer ist, wiederhole:

Der Automat hat ein Attribut *gesperrt*, das anzeigt, dass keine Dosen mehr da sind.

03 Eine Person (z.B eine Studentin) kommt vorbei, sie hat $X \in$.

Anstelle einer Studentin könnte natürlich eine andere Person mit etwas Geld kommen. Die Aktivität hier ist *kommen*.

05 Falls sie genug Geld hat (d.h. mindestens den Preis pro Dose),

Hat sie genug Geld, verglichen mit dem Preis pro Dose, der in dem Automaten eingestellt ist? Die Menge des Geldes ist ein Attribut der Person, der Dosenpreis ein Attribut des Automaten.

06 steckt sie eine Anzahl Y Münzen in den Münzschlitz

Das *GeldEinwerfen* ist eine Aktivität der Studentin. Dabei ist wichtig, wie viele Muenzen in welchen *Getraenkeautomat* (hier gibt es nur einen) gesteckt werden. Der Automat muss die Muenzen akzeptieren und die Anzahl der eingeworfenen Muenzen kennen.

07 und drückt auf den Knopf.

Knopf druecken ist eine Aktivität der Studentin, die sich auf einen bestimmten Automaten bezieht. Gleichzeitig ist dies eine Aufforderung an den Automaten, das Geld zu pruefen und eine Dose herauszugeben.

08 Falls Rückgeld ausgegeben wurde,

09 nimmt sie es.

Nur wenn *Rueckgeld* vorhanden ist, kann sie Geld entnehmen. Rückgeld kann nur vorhanden sein, wenn es vorher eine *Geldrueckgabe* gab.

10 Falls eine Dose ausgegeben wurde,

11 nimmt sie sie und

12 trinkt sie aus.

Falls der Automat eine Dose herausgegeben hat, kann die Studentin die Dose entnehmen.
Dann trinkt sie.

13 Andernfalls stellt sie fest: »Zu wenig Geld!«.

Die Studentin sagt....

14 Die Person verlässt die Szene (hoffentlich nicht mehr durstig und mit weniger Geld).

Die Aktivität ist hier verlassen.

Ende des Szenarios

Analyse einiger Aktivitäten

In der objektorientierten Programmierung kommunizieren Objekte durch Botschaften (englisch *message*). Das Wort »Botschaft« ist eine häufig benutzte, aber ungenaue Übersetzung, weil der Aufforderungscharakter unter den Tisch fällt. Besser ist es, nur von Aufforderungen zu sprechen, die an ein Objekt gerichtet sind. Die Notation ist *Objekt.Aufforderung(Daten)*. In den oben beschriebenen Aktivitäten stecken Aufforderungen: zu 06:

Die Daten, die zu der Aktivität *Geld einwerfen* gehören, sind der Automat, in den die Münzen gesteckt werden, sowie die Anzahl der Münzen. In der obigen Notation geschrieben: *dieStudentin.GeldEinwerfen(derAutomat, Münzenanzahl)*

Die Aufforderung an den Automaten, die *innerhalb* der Aktivität *Geld einwerfen* steckt, ist es, die Münzen zu akzeptieren. Das Akzeptieren der Münzen beinhaltet die Kenntnis, wie viele Münzen eingeworfen wurden. In der obigen Notation geschrieben: *derAutomat.akzeptieren(Münzenanzahl)*

zu 07:

Ähnliche Überlegungen sind für die Aktivität *Knopf druecken* anzustellen, weil diese Aktivität den Automaten dazu veranlasst, das eingeworfene Geld zu prüfen und eine Dose herauszugeben.

Erster Klassenentwurf

In der Analyse können Objekte identifiziert und damit schon erste Klassen gebildet werden. Die Aktivitäten oder das Verhalten der Objekte sind nach außen sichtbar und erscheinen deshalb im Programm als öffentliche Schnittstelle. Im Design werden die Klassen näher untersucht und genauer formuliert. Insbesondere ergeben sich aus den Aktivitäten, welche Informationen über ein Objekt benötigt werden bzw. welche *Attribute* es hat.

Die Klassen ergeben sich aus den handelnden Objekten mit ihren Attributen und Aktivitäten. Der Ansatz, zunächst die Substantive mit Objekten gleichzusetzen und die Verben mit Methoden, kann irreführend sein, weil beides austauschbar ist. Beispiel: »Die Entnahme der Dose wird von der Studentin durchgeführt.« Entnahme als Objekt? Durchführen als Aktivität? – etwas zweifelhaft. »Die Studentin entnimmt die Dose« ist erheblich klarer. Passivkonstruktionen sollten also vor der Analyse der Substantive und Verben stets in Aktivkonstruktionen verwandelt werden. Ebenso ist das Subjekt genau zu identifizieren.

»Die Messung erfolgt um 13 Uhr.« ist in diesem Sinne ein unbrauchbarer Satz, weil die Frage nach dem »wer?« nicht beantwortet werden kann.

Objekte sind aus dem vorherigen Abschnitt identifizierbar. Es gibt Objekte, die jedoch passiv sind: Dosen und Münzen. Zu diesen Objekten gibt es keine Attribute, es interessiert hier nur die Anzahl. Wenn zu einem Objekt keine Attribute und keine Aktivitäten (passiv!) angebbbar sind, ist es im Allgemeinen nicht notwendig, es als Klasse zu formulieren. Die *aktiven*, handelnden Objekte sind die Studentin und der Automat. Da anstelle der Studentin (Spezialfall) auch andere Personen kommen können, ist es sinnvoll, dafür eine Klasse *Person* zu erfinden. Die Klasse für den Automaten nennen wir *GetraenkeAutomat*. Die *vorläufig* gefundenen Attribute und Aktivitäten sind in Tabelle 4.1 zusammengefasst.

Tabelle 4.1: Vorläufige Kandidaten für Klassen, Attribute und Aktivitäten

Klasse	Attribute	Aktivitäten/Zustandsabfragen
Person	Geldmenge	kommen hat genug Geld? Geld einwerfen Knopf drücken Geld entnehmen Dose entnehmen trinken gehen / Szene verlassen
GetränkeAutomat	Anzahl Dosen gesperrt Preis pro Dose Rückgeld eingeworfene Münzen	befüllt werden (von wem?) Geld prüfen Dose herausgegeben? Rückgeld vorhanden? Münzen akzeptieren Dose herausgeben ist gesperrt? Geldrückgabe

4.7.2 Formulierung des Szenarios in C++

Bei den identifizierten Objekten mit ihren Methoden handelt es sich *zunächst um eine erste Näherung*, die weiter verfeinert werden muss. Weil nur ein erster Eindruck vermittelt werden soll, wird auf eine vollständige objektorientierte Analyse (OOA) und ein entsprechendes Design (OOD) verzichtet und auf die Literatur verwiesen, die die OOA/D-Thematik ausführlich behandelt, zum Beispiel [Oe] oder [Bal]. Hier wird *als Starthilfe* das oben strukturiert beschriebene Szenario in einer möglichen C++-Darstellung formuliert. Es gibt nur einige Besonderheiten:

- `main()` beschreibt im Ablauf Aktivitäten einer Person, die ihrerseits Aktivitäten beim Getränkeautomaten auslösen.
- Der Automat wird mit Anfangswerten initialisiert (Konstruktor). Damit ist er am Anfang mit Dosen gefüllt, ohne dass wir uns Gedanken machen müssen, wer ihn gefüllt hat.
- Die Methoden *kommen* und *gehen* werden in C++ geeignet durch Konstruktor und Destruktor beschrieben.

Listing 4.10: Simulation des Getränkeautomaten

```
// /cppbuch/loesungen/k4/4/main.cpp
#include "person.h"
#include "automat.h"
using namespace std;

int zufall(int x) { // gibt eine Pseudo-Zufallszahl zwischen 0 und x zurück
    static long r = 1;
    r = (125 * r) % 8192;
    return (x+1)*r/8192;
}

int main() {
    const int DOSEANZAHL = 50,
            DOSENPREIS = 2, // in €
            MAX_GELD = 20; // in €

    // Szenario:
    // Der Konstruktor initialisiert den Automaten mit der
    // gewünschten Anzahl von Dosen und dem Dosenpreis.
    GetraenkeAutomat objektColaAutomat(DOSEANZAHL, DOSENPREIS);
    while(!objektColaAutomat.istGesperrt()) {
        // eine Person betritt die Szene:
        // Der Konstruktor erzeugt ein Objekt der Klasse Person (siehe Text) mit Namen
        // einePerson, wobei das Objekt (das heißt sein Geldvorrat) mit einer zufälligen
        // Zahl zwischen 0 und MAX_GELD initialisiert wird.
        Person einePerson(zufall(MAX_GELD));
        if(einePerson.genugGeld( objektColaAutomat.getraenkePreis())) {
            // eine zufällig gewählte Anzahl Münzen wird eingeworfen,
            // aber nicht mehr als vorhanden:
            int wieviel = zufall(einePerson.wievielGeld());
            einePerson.geldEinwerfen(objektColaAutomat, wieviel);
            einePerson.knopfDruecken(objektColaAutomat);
            if(objektColaAutomat.rueckgeldVorhanden())
                einePerson.geldEntnehmen( objektColaAutomat.geldRueckGabe());
            if(objektColaAutomat.doseHerausgegeben()) {
                einePerson.doseEntnehmen(objektColaAutomat);
                einePerson.trinkt();
            }
            if(objektColaAutomat.istGesperrt())
                cout << "Automat gesperrt! (leer)" << endl;
        }
        else einePerson.sagt(" Leider zu wenig Geld.");
    } // Blockende: die Studentin verlässt die Szene:
    // hier automatisch realisiert durch den Destruktor
}
```



Übungen

4.4 Versuchen Sie, die einzelnen Schritte nachzuvollziehen. Stellen Sie fest, wo es Unzulänglichkeiten und Unvollständigkeiten gibt. Entwerfen Sie die nötigen Klassen in C++, vervollständigen Sie das Programm und bringen Sie es zum Laufen. Das Programm sollte verschiedene Fälle abdecken (verschiedene, zufällig gewählte anfängliche Geldmengen),

die in der zu erzeugenden Testdokumentation aufgeführt werden. Die Testdokumentation erhält man am einfachsten durch Umleiten der Bildschirmausgaben in eine Datei.

Falls der Automat einer Methode als Parameter übergeben wird, die ihrerseits eine Methode des Automaten aufruft, welche den Zustand des Automaten ändert, ist nur die Übergabe per Referenz sinnvoll. Änderungen des Zustands des Automaten müssen im Original wirksam werden, nicht in einer Kopie, die am Ende der Methode weggeworfen wird. Dies kann erzwungen werden, wenn man den Kopierkonstruktor privat deklariert. Eine Definition ist nicht erforderlich, weil kein Aufruf erfolgt.

4.5 Wie können Sie mit C++ erreichen, dass ein Attribut *direkt*, also ohne Einsatz einer Methode, zwar gelesen, aber nicht verändert werden kann? Beispiel:

```
int main() {
    MeineKlasse objekt;
    objekt.readonlyAttribut = 999; // Fehler! nicht erlaubte Aktion
    // erlaubter direkter lesender Zugriff:
    cout << "objekt.readonlyAttribut="
          << objekt.readonlyAttribut << endl;    // ok
}
```

Wie sieht die Realisierung in der Klasse *MeineKlasse* aus?

4.6 Schreiben Sie auf Basis der Lösung der Taschenrechner-Aufgabe von Seite 134 eine Klasse *Taschenrechner*, die einen eingegebenen String verarbeitet. Die Anwendung könnte wie folgt aussehen:

```
int main() {
    while(true) {
        // Abbruch mit break
        cout << "Bitte einen mathematischen Ausdruck eingeben, z.B. 4*(12+3)"
              << "\n(Abbruch durch Eingabe einer Leerzeile) : ";
        string anfrage;
        getline(cin, anfrage);
        if(anfrage.length() > 0) {
            Taschenrechner tr(anfrage);
            cout << "Das Ergebnis der Anfrage '"
                  << tr.getAnfrage() << "' ist " << tr.getErgebnis() << endl;
        }
        else break;
    }
}
```

Der auf den Seiten 119-120 verwendete Funktionsaufruf `cin.get(c)`, muss dabei durch den Aufruf einer Funktion ersetzt werden, die das jeweils nächste Zeichen des übergebenen Anfrage-Strings holt.

4.8 Gegenseitige Abhängigkeit von Klassen

In der Lösung der Aufgabe des vorhergehenden Abschnitts werden in den inline-Funktionen der Klasse `Person` teilweise Methoden der Klasse `GetraenkeAutomat` benutzt, weswegen *automat.h* von *person.h* eingeschlossen wird. Hingegen benutzt die Klasse `GetraenkeAutomat` keinerlei Eigenschaften oder Methoden der Klasse `Person`. Was ist aber, wenn jede der beiden Klassen Methoden der jeweils anderen Klasse benutzt? Es nutzt nichts, gegenseitig die Header-Datei der jeweils anderen Klasse einzuschließen, weil der Compiler die nötigen Informationen nicht bekommt. Betrachten wir zwei Header-Dateien, die sich aufeinander beziehen:

```
// Datei A.h
#ifndef A_h
#define A_h
#include "B.h"
.... usw.
#endif
```

```
// Datei B.h
#ifndef B_h
#define B_h
#include "A.h"
.... usw.
#endif
```

Wenn *A.h* zuerst gelesen wird, wird bei Ausführung der dritten Zeile *B.h* eingelesen. Die Ausführung der dritten Zeile von *B.h* scheitert jedoch, weil *A.h* nun definiert ist und der Rest von *A.h* nicht zur Kenntnis genommen wird. Die Lösung des Problems besteht in der *Vorwärtsdeklaration*:

```
// Datei A.h
#ifndef A_h
#define A_h

class B; //Vorwärtsdeklaration

class A {
public:
    void benutzeB(const B&);
    void eineAMethode();
    // ... usw.
};
#endif
```

```
// Datei B.h
#ifndef B_h
#define B_h

class A; //Vorwärtsdeklaration

class B {
public:
    void machWasMitA(A*);
    void eineBMethode() const;
    // ... usw.
};
#endif
```

Die Notation `A*` bedeutet »Zeiger² auf Objekt der Klasse `A`«. In den Header-Dateien werden gegenseitig nur die Klassennamen bekannt gemacht, und die Kenntnisnahme der Methoden wird auf die Implementierungsdateien verschoben. Dies funktioniert dann, wenn die Header-Dateien ausschließlich Zeiger oder Referenzen der jeweils anderen Klasse enthalten, aber keine Methodenaufrufe. Dies kann leicht erreicht werden, wenn auf inline-Methoden verzichtet wird, die Methoden der anderen Klasse benutzen. Die dazu notwendige Struktur wird für zwei Klassen gezeigt, eine Erweiterung auf die gegenseitige

² Zeiger werden in Kapitel 5 besprochen, aber hier der Vollständigkeit halber mit erwähnt.

Abhängigkeit mehrerer Klassen ist nach diesem Muster leicht möglich. Die Implementierungsdateien schließen die Header-Dateien auch der anderen Klasse ein, wobei die Reihenfolge der Include-Anweisungen keine Rolle spielt. Nun können Methoden der jeweils anderen Klasse problemlos in den Implementierungsdateien aufgerufen werden.

```
// Datei A.cpp
#include "A.h"
#include "B.h"

void A::benutzeB(const B& b) {
    b.eineBMethod();
}

void A::eineAMethode() {
    .... usw.
```

```
// Datei B.cpp
#include "B.h"
#include "A.h"

void B::machWasMitA(A* pA) {
    pA->eineAMethode();
}

void B::eineBMethod() const {
    .... usw.
```

Es gibt natürlich auch Fälle, wo einer Klasse die Größe eines Objekts einer anderen Klasse bekannt sein *muss*, wenn zum Beispiel die Klasse A ein Objekt der Klasse B aggregiert. In diesem Fall hilft die Vorwärtsdeklaration, unsymmetrisch angewendet, ebenfalls weiter, wie das folgende Listing zeigt. Damit kann natürlich nicht der unwahrscheinliche Fall gelöst werden, dass die Klasse A ein Objekt der Klasse B aggregieren *und* Klasse B ein Objekt der Klasse A einschließen soll. Wer das unbedingt möchte, sollte seinen Entwurf noch einmal überdenken. Wenn es denn sein muss, kann dieser Fall mit als Zeiger auf A bzw. B realisierten Attributen gelöst werden, an die der Konstruktor zur Laufzeit dynamisch erzeugte Objekte hängt.

```
// Datei A.h
#ifndef A_h
#define A_h
// Klasse B einschließen:
#include "B.h"

class A {
public:
    void benutzeB(const B&);
    void eineAMethode();
private:
    B einB; // aggregiertes Objekt
    // .... usw.
};
#endif
```

```
// Datei B.h
#ifndef B_h
#define B_h
// Vorwärtsdeklaration:
class A;

class B {
public:
    void machWasMitA(A*);
    void eineBMethod() const;
    // .... usw.
};
#endif
```

4.9 Konstruktor und mehr vorgeben oder verbieten³

Der Compiler erzeugt automatisch einen Konstruktor ohne Argumente, einen Kopierkonstruktor, einen Destruktor und einen Zuweisungsoperator, falls eine Klasse diese nicht zur Verfügung stellt. Um dem Programmierer mehr Kontrolle darüber zu erlauben, sind die syntaktischen Konstruktionen `= default` und `= delete` eingeführt worden. Beispiel:

```
class X {
public:
    X() = default; // Compiler-generierten Konstruktor erlauben, obwohl es schon
                  // X(int) gibt.

    X(int i);
    X(const X&) = delete;           // Kopieren und
    X& operator=(const X&) = delete; // Zuweisung verbieten
    void* operator new(std::size_t) = delete; // Erzeugung mit new verbieten
    void f(int);
    void f(double) = delete; // Aufruf von f(double) verbieten
};
```

Ohne die `f(double)`-Deklaration wäre wegen der automatischen Typumwandlung ein Aufruf von `f(int)` mit einem `double`-Argument möglich. Alternativ kann man die zu verbietenden Methoden in den `private`-Bereich legen. Dabei genügen die Prototypen; eine Implementation wird nicht gebraucht.

4.10 Delegierender Konstruktor⁴

Auf Seite 156 wird die Initialisierung eines Objekts mit Listen in der Konstruktordefinition gezeigt. Eine nach [ISOC++] neue Möglichkeit ist der Aufruf eines anderen Konstruktors derselben Klasse in der Initialisierungsliste. Der Konstruktor *delegiert* so seine Aufgabe an einen anderen Konstruktor. Der Sinn besteht darin, die Initialisierung von Attributen ohne Code-Duplikation zu erreichen. Ein Beispiel noch ohne delegierenden Konstruktor:

```
// cppbuch/k4/delegierenderKonstruktor/klasse1.h
class Klasse {
public:
    Klasse(int a, int b) // Konstruktor 1
    : attr1(a), attr2(b) {
        weitereInitialisierungen();
    }
};
```

³ Dieser Abschnitt sollte beim ersten Lesen übersprungen werden. Zu seinem Verständnis wird die Kenntnis der Kapitel 5 und 9 vorausgesetzt.

⁴ Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

```

    Klasse()                // Konstruktor 2
    : attr1(1), attr2(42) { // vorgegebene Werte
        weitereInitialisierungen();
    }
private:
    void weitereInitialisierungen() {
        // Code für weitere Initialisierungen
    }
    int attr1;
    int attr2;
};

```

In diesem Beispiel ist ein Teil der Initialisierung in eine Funktion ausgelagert worden, die Initialisierung der Attribute `attr1` und `attr2` jedoch nicht. Im Fall von Anpassungen sind ggf. beide Konstruktoren zu ändern, was fehleranfällig sein kann. Eine Delegation der Konstruktoraufgaben würde Letzteres vermeiden helfen. Der Programmcode der Funktion `weitereInitialisierungen()` wird dabei in den Konstruktor verlegt, den der andere aufruft:

```

// cppbuch/k4/delegierenderKonstruktor/klasse2.h
class Klasse {
public:
    Klasse(int a, int b) // Konstruktor 1
    : attr1(a), attr2(b) {
        // Code für weitere Initialisierungen
    }
    Klasse()            // Konstruktor 2
    : Klasse(1, 42) {    // Delegation an Konstruktor 1
    }
private:
    int attr1;
    int attr2;
};

```

Die Klasse ist kürzer und lesbarer geworden, weil der zweite Konstruktor nunmehr fast leer ist. Das folgende Beispiel zeigt, welcher Konstruktor aufgerufen wird:

```

Klasse k1(9); // Konstruktor 1: Daten: 9, 17
Klasse k2(5, 20); // Konstruktor 1: Daten: 5, 20
Klasse k3; // Konstruktor 2, Konstruktor 1 rufend: Daten: 1, 42

```


5

Intermezzo: Zeiger

Dieses Kapitel behandelt die folgenden Themen:

- Was sind Zeiger und wie benutze ich sie?
- C-Arrays und C-Strings
- Dynamisches Erzeugen von Objekten
- Parameterübergabe mit Zeigern
- Mehrdimensionale C-Arrays
- Binärdaten in Dateien schreiben und lesen
- Zeiger auf Funktionen
- Unterschied zwischen Wert- und Referenzsemantik

Zeiger sind unverzichtbare Elemente in C++. Nach der Einführung von Zeigern werden C-Arrays beschrieben und ihre Anwendung gezeigt. Verwandt mit Zeigern und C-Arrays sind C-Strings, eine andere Form von Zeichenketten. Mit Hilfe von Zeigern werden dynamisch, das heißt zur Laufzeit, erzeugte Objekte verwaltet. Auch die binäre Ein- und Ausgabe mit Dateien setzt die Begriffe Zeiger und Adresse voraus. Abschließend werden Zeiger im Zusammenhang mit Funktionen behandelt.

Viele Klassen sind in C++ ohne Zeiger nicht realisierbar. In den folgenden Kapiteln werden häufig Zeiger und C-Arrays verwendet, weswegen dieses Kapitel an dieser Stelle notwendig ist. Zeiger erlauben große Freiheiten, haben aber auch ihre Tücken.

5.1 Zeiger und Adressen

Zeiger sind ähnlich wie andere Variablen: Sie haben einen Namen und einen Wert, und sie können mit Operatoren verändert werden. Der Unterschied besteht darin, dass der Wert als *Adresse* behandelt wird. Ein Beispiel dafür sind Seitenangaben in einem Inhaltsverzeichnis, die »Adressen« für verschiedene Kapitel darstellen.

Die Namen können konventionell ein p oder ptr (für »pointer«) enthalten. Zeiger werden in C++ wie in C sehr häufig verwendet, weil sie eine große Flexibilität gestatten. Mit Hilfe von Zeigern kann dynamisch, das heißt zur Laufzeit eines Programms, Speicher beschafft werden. Anwendungen werden Sie noch kennenlernen. In Deklarationen bedeutet ein * »Zeiger auf«:

```
int *ip;
```

ip ist ein Zeiger auf einen int-Wert oder anders ausgedrückt: In der Speicherzelle, deren Adresse in ip gespeichert ist, befindet sich ein int-Wert. In anderen Anweisungen bedeutet * eine *Dereferenzierung*, das heißt, dass der Wert an der Stelle betrachtet wird, auf die der Zeiger verweist. *ip = 100; setzt den Wert der Speicherzelle, auf die ip zeigt, auf 100. Insofern könnte man die obige Deklaration lesen als: (*ip) ist vom Datentyp int.

Zeiger erhalten bei der Deklaration zunächst eine *beliebige* Adresse, genau wie andere nicht-initialisierte Variable zunächst beliebige Werte annehmen (Ausnahme: static-Variablen, siehe Seite 105). Daher muss vor Benutzung des Zeigers in einem Ausdruck erst eine sinnvolle Adresse zugewiesen werden, um nicht den Inhalt anderer Speicherzellen zu zerstören! Zur Verdeutlichung des Prinzips betrachten wir der Reihe nach verschiedene Deklarationen. Zunächst definieren und initialisieren wir eine Variable i mit der Anweisung int i = 99;. Wir legen damit einen Speicherplatz mit dem symbolischen Namen i an und tragen die Zahl 99 ein. Die uns unbekannte, vom Compiler für i festgelegte Speicherplatzadresse sei 10125.

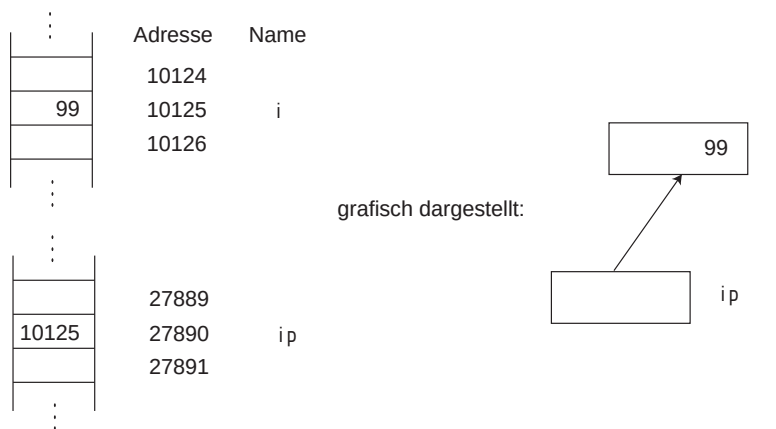


Abbildung 5.1: Zeiger

Nun werde mit `int *ip`, ein Zeiger `ip` definiert, aber nicht initialisiert. Der Wert des Zeigers ist jetzt rein zufällig. Die Speicherplatzadresse des Zeigers selbst sei 27890, und er zeigt auf eine unbekannte Adresse. Jetzt wird `ip` die Adresse von `i` zugewiesen. Dabei kommt der Operator `&` zur Anwendung, der hier als Adressoperator wirkt. In einem anderen Kontext hatten wir das Zeichen `&` bereits als Operator für die bitweise UND-Operation kennengelernt. Weisen wir nun `ip` die Adresse von `i` zu:

```
ip = &i;
```

Jetzt zeigt `ip` auf `i`. Das heißt nichts anderes, als dass die Adresse von `i`, hier 10125, bei `ip` eingetragen wird (Abbildung 5.1). Als Nächstes definieren wir einen Zeiger `ip2`, der ebenfalls auf `i` gerichtet wird. Durch die Initialisierung gleichzeitig mit der Definition durchläuft `ip2` keinen undefinierten Zustand:

```
int *ip2 = &i;
```

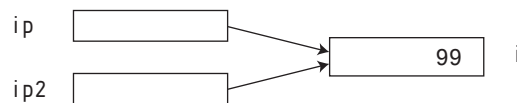


Abbildung 5.2: Zwei Zeiger zeigen auf ein Objekt.

Der Wert von `i` ist jetzt über mehrere Namen beziehungsweise Zeiger zugreifbar (Abbildung 5.2). Die Anweisungen

```
i = 99;
*ip = 99;
*ip2 = 99;
```

bewirken alle dasselbe. `*ip` und `*ip2` sind *Alias-Namen* für `i`.

Hinweis zur Schreibweise von Zeigerdeklarationen

Die folgenden drei Schreibweisen sind äquivalent:

```
int* ip, x;    // 1
int * ip, x;   // 2
int *ip, x;    // 3

// Nur eine Variable pro Deklaration: Verwechslung nicht möglich
int *ip;       // 4
int x;

int* ip;       // 5
int x;
```

Die Variable `x` ist vom Typ `int`, die Variable `ip` ist vom Typ »Zeiger auf `int`«. Der Stern `*` bezieht sich also nur *auf den direkt folgenden Namen*. Um die Zuordnung beim Lesen leichter treffen zu können, sollte deshalb nur die vierte oder fünfte Schreibweise benutzt werden. Ein Zeiger auf eine Referenz ist nicht möglich.

Null-Zeiger / nullptr

In C gibt es einen speziellen Zeigerwert, nämlich *NULL*. *NULL* ist ein im Header `<stddef>` definiertes Makro. Ein mit *NULL* initialisierter Zeiger zeigt nicht irgendwohin, sondern definitiv auf »nichts«. *NULL* ist als logischer Wert abfragbar. Um sich zu merken, dass ein Zeiger noch nicht oder nicht mehr auf ein definiertes Objekt zeigt, kann ein Zeiger auf *NULL* gesetzt werden: `int *iptr = NULL;`. In C++ wird *NULL* als Zahlenwert 0 oder 0L (long) dargestellt. Um einen Null-Zeiger von einer Zahl unterscheiden zu können, ist im neuen C++-Standard das Schlüsselwort `nullptr` vorgesehen. Die Verwendung von *NULL* oder `nullptr` erhöht die Lesbarkeit an den Stellen, wo nicht unbedingt klar ist, ob ein Zeiger oder eine Zahl gemeint ist, zum Beispiel in der Parameterliste eines Funktionsaufrufs.

Typprüfung

Im Gegensatz zu C wird der Typ eines Zeigers in C++ geprüft. Gegeben seien die Definitionen:

```
char *cp;      // Zeiger auf char
void *vp;      // Zeiger auf void
```

`void` hat die Bedeutung »undefinierter Datentyp«. `vp` ist also ein Zeiger auf ein Objekt, über dessen Typ nichts gesagt wird. Eine Deklaration für ein `void`-Objekt kann es nicht geben, weil der benötigte Speicherplatz nicht angebbbar ist, sehr wohl aber die Deklaration eines Zeigers, der auf ein Objekt unbekannten Typs gerichtet werden soll. Aus dem Typ des Zeigers geht der Typ des Objekts nicht hervor. Ein Zeiger auf `void` hat den Typ `void*`. Ein `void*`-Zeiger kann zum Beispiel auf ein `char`-Objekt gerichtet werden. Das Umgekehrte geht nicht.

```
vp = cp;       // möglich
cp = vp;       // nicht möglich, Fehlermeldung des Compilers!
```

Die Zuweisung kann jedoch durch eine Typumwandlung ermöglicht werden:

```
cp = static_cast<char*>(vp);
```

Solche Typumwandlungen sollte man nicht ohne wichtigen Grund benutzen, weil die Typkontrolle des Compilers umgangen wird.

Kein Zeigerzugriff auf Block-lokale Variablen!

Der Speicherplatz für die innerhalb eines Blocks deklarierten Variablen wird bei Verlassen des Blocks wieder freigegeben. Der nachträgliche Zugriff auf diese Speicherplätze kann zu Dateninkonsistenzen und zum Systemabsturz führen:

```
int i = 9;
int *ip = &i;           // Zeiger ip zeigt auf i
*ip = 8;                // i erhält den Wert 8
{ // neuer Block beginnt
    int j = 7;
    ip = &j;            // Zeiger ip zeigt auf j
    // weiterer Programmcode
}                       // Blockende, j wird ungültig
*ip = 8;                // gefährlich!
```

Die letzte Anweisung versucht, `j` über den Alias-Namen `ip` den Wert 8 zuzuweisen. Da `j` aber nicht mehr existiert, ist die Speicherstelle, auf die `ip` zeigt, möglicherweise vom Betriebssystem bereits für andere Zwecke vergeben worden. Durch diese Zuweisung werden daher gegebenenfalls andere Daten zerstört. Es gibt keine Warnung durch den Compiler oder das Laufzeitsystem! Daher sollten Sie darauf verzichten, Zeiger auf lokale Objekte mit gegenüber den Zeigern eingeschränkter Gültigkeit zu verwenden. Stattdessen sollten Sie lieber Zeiger desselben Gültigkeitsbereichs benutzen. In diesem Beispiel wäre als letzte Anweisung `j = 8;` »günstiger« gewesen, die vom Compiler sofort als falsch erkannt worden wäre und somit einen Hinweis auf Fehler im Programm gegeben hätte.

Konstante Zeiger auf konstante Werte

Die Bedeutung von `const char*` ist »Zeiger auf konstante Zeichen« und ist nicht zu verwechseln mit »konstantem Zeiger auf Zeichen« (`char *const`). Solche Deklarationen sind von rechts nach links zu lesen, wobei die Reihenfolge (`const char`) auch (`char const`) lauten kann. Ein konstanter Zeiger auf konstante Zeichen ist demnach vom Typ `const char * const` oder gleichwertig `char const * const`.

5.2 C-Arrays

Vektoren sind von Seite 81 bekannt. C-Arrays genannte Felder sind etwas Ähnliches, nur viel primitiver: Es sind Bereiche im Speicher, die (eingeschränkt) ähnlich wie ein Vektor benutzt werden können. Der Zugriff auf ein einzelnes Element eines C-Arrays geht über den von den Vektoren bekannten Indexoperator `[ ]`. Bei einer zweidimensionalen Tabelle ist zusätzlich die Spaltennummer anzugeben, zum Beispiel `[6][3]`. *AnzahlDerElemente* in Abbildung 5.3 muss eine Konstante sein oder ein Ausdruck, der ein konstantes Ergebnis hat.



Abbildung 5.3: Syntax der Definition einer eindimensionalen Tabelle

```
const int ANZAHL = 5;
int Tabelle[ANZAHL]; // Beispiel einer eindimensionalen Tabelle
```

Es ist guter Programmierstil, die Größe eines C-Arrays als *Konstante* zu deklarieren und die Konstante in der Arraydeklaration und im restlichen Programm zu verwenden. Dadurch kann ein Programm an eine andere Arraygröße angepasst werden, indem nur der Wert der Konstanten geändert wird. Es kann auch direkt eine Zahl eingetragen werden. `char a[10];` bezeichnet zum Beispiel ein Feld namens `a` mit 10 Zeichen.

Der Compiler reserviert für alle Elemente ausreichend Speicherplatz, die Anzahl der Tabellenelemente ist danach während des Programmlaufs nicht veränderbar. Die Anzahl

der Elemente ist daher problemabhängig ausreichend groß zu wählen, auch wenn einige Tabellenplätze möglicherweise nicht ausgenutzt werden. Arrays, deren Größe erst zur Laufzeit festgelegt wird, lassen sich auch konstruieren, doch davon später mehr. Die Abbildung 5.4 zeigt ein Array mit 5 ganzen Zahlen.

⋮	Index	
17	0	← Tabelle
35	1	
112	2	
-3	3	
1000	4	
⋮		

Abbildung 5.4: int-Array Tabelle

Der *Name des Feldes* zeigt auf die *Startadresse* des Feldes, d.h. auf das erste Element, und ist wie ein Zeiger einsetzbar. Anders ausgedrückt: Der Feldname ist wie ein Zeiger auf das erste Element (das heißt das Element mit dem Index 0) des Arrays. Ein Unterschied zu Zeigern besteht dennoch. Da dem Array bereits fest Speicherplatz zugewiesen ist, würde eine Änderung dieses »Zeigers« den Speicherplatz unzugänglich machen, weil die Information über die Adresse verloren geht. Daher kann ein Feldname kein sogenannter *L-Wert* sein, siehe auch Seite 953. Der Begriff L-Wert bezeichnet eine Größe, die auf der linken Seite einer Zuweisung stehen darf. Das Gegenstück ist der *R-Wert*. Also: Der Feldname ist ein R-Wert und *konstanter* Zeiger auf das erste Element des Feldes (Arrays). Der Zugriff auf ein Element ist durch den *Indexoperator* `[ ]` oder durch Zeigerarithmetik möglich, wie die Zuweisung eines Fragezeichens an das sechste Element zeigt (die Nummerierung beginnt bei 0!). Zwischen den eckigen Klammern wird die relative Tabellenposition eingetragen.

```
a[5]    = '?';    // gleichwertig:
*(a+5)  = '?';    // = Wert an der Stelle a + 5
```

Zeigerarithmetik ist hier die Addition von 5 zu einem Zeiger mit der Bedeutung, dass das Ergebnis als ein um 5 Positionen verschobener Zeiger aufgefasst wird. Einzelheiten zur Zeigerarithmetik folgen auf Seite 192.

```
char c = a[9]; // 10. Element
char* cp;     // Zeiger auf char
cp = &c;      // möglich

cp = a; // cp zeigt auf den Feldbeginn, d.h. cp == &a[0] bzw. *cp == a[0]

a = cp;       // Fehler : a ist kein L-Wert
a = &c;       // Fehler : a ist kein L-Wert
```

Zeiger und Arrays

Zeiger und Arrays sind im Gebrauch sehr ähnlich. Deswegen werden die Unterschiede hier zusammengefasst:

- Ein Zeiger ist der symbolische Name für einen Speicherplatz, der einen Wert enthält, der als Adresse benutzt werden kann.
- Ein Array (d.h. der Feldname) besitzt *in diesem Sinne keinen* Speicherplatz. Genau wie einer Konstanten keine Speicherzelle zugeordnet sein muss, weil der Compiler den Wert jedesmal direkt verwenden kann, ist keine Speicherzelle notwendig, die die Adresse des Arrays enthält. Ein Array ist vielmehr ein symbolischer Name für die Adresse (= den Anfang) eines Bereichs im Speicher. Der Name wird *syntaktisch* wie ein konstanter Zeiger behandelt.

5.2.1 C-Arrays und sizeof

C-Arrays sind als »roher Speicher« (englisch *raw memory*) *keine* Objekte im bisherigen Sinn. Es gibt keine Methoden, die verwendet werden könnten:

```
vector<double> kosten(12);
// ... berechnen
for(size_t i = 0; i < kosten.size(); ++i) // nicht bei C-Arrays!
    cout << i << ": " << kosten[i] << endl;
```

Die Größe eines C-Arrays kann nicht von ihm erfragt werden, sie muss vielmehr vorher bekannt oder mit `sizeof` ermittelt worden sein:

```
const int ANZAHL = 5;
int tabelle[ANZAHL];           // C-Array-Definition
// ... Berechnungen
for(size_t i = 0; i < ANZAHL; ++i)
    cout << i << ": " << tabelle[i] << endl; // oder:
for(size_t i = 0; i < sizeof tabelle / sizeof tabelle[0]; ++i)
    cout << i << ": " << tabelle[i] << endl;
```

`sizeof` ist ein Operator, der den Platzbedarf eines Ausdrucks oder Typs in Bytes zurückgibt. Ein Ausdruck kann, ein Typ muss in runden Klammern eingeschlossen sein. Die Anzahl der Elemente eines C-Arrays ergibt sich einfach durch Division des Platzbedarfs für das ganze Feld durch den Platzbedarf für ein einzelnes Element, in diesem Fall das erste. Wenn der Datentyp wie hier eindeutig bekannt ist, kann statt `sizeof tabelle[0]` auch `sizeof(int)` geschrieben werden. `sizeof` funktioniert jedoch nicht bei C-Arrays, die als Parameter einer Funktion übergeben werden, weil C-Array-Parameter innerhalb der Funktion nur als Zeiger interpretiert werden. Einzelheiten sind auf Seite 211 zu finden.

5.2.2 Indexoperator bei C-Arrays

Der Zugriff über den Index-Operator `[ ]` wird nicht auf seine Grenzen überprüft! Er kann durch das entsprechende Zeigeräquivalent ersetzt werden. Man kann genausogut `*(tabelle + i)` anstatt `tabelle[i]` verwenden. Der Grund dafür liegt darin, dass der Compiler ohnehin *jeden* Zugriff über `[ ]` in die Zeigerform übersetzt, solange nicht ein benutzerdefinierter Operator `[ ]` verwendet wird. Letzterer ist nur bei Klassen möglich, siehe Kapitel 9. Mit `(tabelle+i)` ist die Adresse gemeint, die um `i` Positionen (nicht Bytes, siehe Abschnitt 5.2.4) weiter liegt als der Feldanfang, auf den `tabelle` zeigt. Der Stern `*` sorgt dafür, dass der Wert genommen wird, der an dieser Adresse eingetragen ist.

5.2.3 Initialisierung von C-Arrays

Ein C-Array kann bei der Definition bereits initialisiert werden. Trotz des '='-Zeichens in der Initialisierungszeile darf das Array auf der linken Seite stehen. Begründung: Eine Initialisierung ist keine Zuweisung. In C++ darf im Gegensatz zu C das '='-Zeichen entfallen. Der Compiler entnimmt die Anzahl der Feldelemente aus der Initialisierungsliste, eine Zahlenangabe kann deshalb entfallen. Die Anzahl kann mit `sizeof` ermittelt werden:

```
int feld[ ] = { 1, 777, -500};
const int ANZAHL = sizeof feld/ sizeof feld[0];
```

Falls weniger Elemente in der Initialisierungsliste als vorhanden angegeben sind, werden die restlichen Elemente mit 0 initialisiert. `int feld[3] = {1};` ist identisch mit `int feld[3] = {1, 0, 0};`.

5.2.4 Zeigerarithmetik

Wenn ein Zeiger inkrementiert oder dekrementiert wird, zeigt er nicht auf die nächste Speicheradresse, sondern auf die Adresse des nächsten Werts. Der Abstand der Speicheradressen ist gleich der Größe, die der Wert beansprucht, zum Beispiel 8 Byte bei einem Datentyp `double`. Gegeben seien folgende Deklarationen (hier gleichzeitig Definitionen):

```
double d[10];      // Array d
double *dp1 = d;
double *dp2;
```

Wenn nun `dp2 = dp1 + 1;` gesetzt wird (oder `dp2 = dp1; ++dp2;`), ergibt sich als Differenz `dp2 - dp1` der Wert 1. `dp2` zeigt also auf das nächste `double`-Element des Arrays. Aufgrund des notwendigen Platzbedarfs für eine `double`-Zahl im Speicher von angenommen 8 Byte (die Zahl mag in Ihrem System anders sein) ist die nächste `double`-Zahl also 8 Byte entfernt. Das wird ermittelt, indem man die Zeigerwerte in `long` umwandelt und dann die Differenz berechnet:

```
cout << dp2-dp1 << endl;           // ergibt 1
cout << reinterpret_cast<long>(dp2)
    - reinterpret_cast<long>(dp1) << endl; // ergibt 8
```

Zur Wandlung des Zeigers wird der `reinterpret_cast`-Operator verwendet. Dieser Operator verzichtet im Gegensatz zum `static_cast`-Operator auf jegliche Typ-Verträglichkeitsprüfung. Nur beim Datentyp `char` wären beide Werte identisch, weil eine Variable vom Typ `char` eben genau ein Byte belegt. Mit Zeigern kann also gerechnet werden, um Adressen oder Adressendifferenzen zu ermitteln. Die Einheit ist dabei nicht Byte, sondern ergibt sich aus dem Datentyp des Zeigers.

An dieser Stelle wird die Suche eines Werts in einer Tabelle von Seite 84 aufgegriffen, nur dass jetzt ein C-Array mit Zeigeroperationen anstelle eines Vektors eingesetzt wird. Diese Variante erlaubt die kürzeste Formulierung der Schleife, setzt aber wie die 4. Variante in Abschnitt 1.9.2 voraus, dass das Feld um einen Eintrag erweitert wird, der als »Wächter« (englisch *sentinel*) dazu dient, die Schleife abubrechen.

```
// Definitionen
const int N = ...
int a[N+1];      // C-Array
```

```
int key = ...    // gesuchtes Element
int i;          // Laufvariable
               // Ergebnis: i = 0..N - 1 : gefunden, i = N : nicht gefunden!
```

Das (N+1). Element dient wie gesagt als »Wächter«. Der Zeiger `p` wird auf das Array gerichtet. Der Abbruch der Schleife ist durch `a[N]` als »Wächterelement« garantiert.

```
a[N] = key;
int *p = a;
while(*p++ != key);
i = p-a-1;          // Zeigerarithmetik
```

`*p++` bedeutet, erst den Wert an der Stelle `p` zu nehmen und dann `p` hochzuzählen. `*p++` ist also identisch mit `*(p++)`, wohingegen `(*p)++` den Wert an der Stelle `p` anstelle des Zeigers `p` hochzählt. Diese sehr kompakte Schreibweise trägt nicht unbedingt zum schnellen Verständnis eines Programms bei, insbesondere bei fehlenden erläuternden Kommentaren. Machen Sie sich die Wirkungsweise jedoch klar, weil viele Programmierer diese Schreibweise bevorzugen und man deren Programme schließlich auch verstehen sollte.

5.3 C-Zeichenketten

Eine *C-Zeichenkette* (englisch *string*) ist ein Spezialfall eines Arrays. Mit C-String meint man eine Folge von Zeichen des Typs `char`, die mit `'\0'` abgeschlossen wird. Diese C-Strings sind nicht zu verwechseln mit den String-Objekten von Seite 86, deren Basis sie bilden. Umgangssprachlich kann »String« sowohl ein C++-Stringobjekt wie auch einen C-String meinen. `'\0'` ist das ASCII-Zeichen mit dem Wert 0, nicht das Ziffernzeichen '0'. Der Datentyp für einen C-String ist `char*` und stellt einen Zeiger auf den Beginn der Zeichenfolge dar, über den auf den C-String zugegriffen wird. Bei der Ausgabe einer Zeichenkette »weiß« der zu `cout` gehörende Ausgabeoperator `<<`, dass `char*` *nicht* als Zeiger, sondern als mit `'\0'` terminierter String aufzufassen ist:

```
const char * str = "ABC"; // const: siehe unten
cout << str;
```

Hier wird `str` gleichzeitig definiert und initialisiert. Ein Programmierer muss an dieser Stelle `'\0'` nicht hinschreiben, weil es vom Compiler ergänzt wird.

Der Compiler erkennt einen C-String am Datentyp `char*`. Eine Zeichenkettenkonstante, auch *Literal* genannt, erkennt er daran, dass sie in Anführungszeichen eingeschlossen ist. Der vom Compiler für den String »ABC« reservierte Speicherplatz beträgt 4 Byte, ein Byte pro Zeichen und ein Byte für `'\0'`. Der Zugriff auf einzelne Zeichen ist auf die Arten möglich, die wir schon von den Arrays her kennen. `cout << str[0];` zeigt das erste Zeichen, welches an der Position 0 steht (die interne Zählung läuft auch hier ab 0). Das Zeichen mit der Nummer `i` ist ansprechbar mit `str[i]` oder `*(str+i)`. Im Gegensatz zu `char`-C-Arrays werden Stringliterals bei vielen Systemen im schreibgeschützten Speicher angelegt: Eine Änderung wie `str[0] = 'X';` führt dann zu einer Laufzeitfehlermeldung.

Deswegen: Zeiger auf Textlitterale stets als `const char*` deklarieren! Eine neue Zuweisung `str = "neuer Text";` ist möglich, wobei gleichzeitig die Information über die vorherige Stelle verloren geht (siehe Abbildung 5.5).

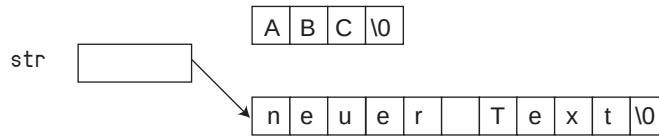


Abbildung 5.5: Neuzuweisung eines Strings

Zur Bearbeitung von C-Strings gibt es eine Menge vordefinierter Funktionen, die in der Datei `string.h` (Header `<cstring>`) deklariert sind. `#include<cstring>` veranlasst den Compiler, diese Datei zu lesen, anschließend kennt er die Funktionen. Die Funktionen sind in Abschnitt 35.9 beschrieben. Wegen der C++-Standardklasse `string` haben diese Funktionen stark an Bedeutung verloren. Hier sei beispielhaft nur die Funktion `strlen()` gezeigt, die die Anzahl der Zeichen eines Strings (ohne '\0') zurückgibt. Zeichen wie '\n' zählen als einzelne Zeichen mit, wie auch '\', wodurch das Anführungszeichen als zum String zugehörig erscheint und eine vorzeitige Interpretation als Stringende verhindert wird.

Listing 5.1: Zusammengesetzter C-String

```
// cppbuch/k5/langstr.cpp
#include<cstring>
#include<iostream>
using namespace std;

int main() {
    const char* const TEXT= "Bei Initialisierung, Zuweisung, "
        "oder Ausgabe kann ein Stringliteral wie hier auch "
        "aus einzelnen Teilstrings zusammengesetzt sein.";
    cout << TEXT << endl
        << "enthält " << strlen(TEXT) << " Zeichen\n";
}
```

Der Speicherplatz für ein Literal, also eine durch Anführungszeichen begrenzte Zeichenfolge, wird zur Compilerzeit festgelegt und kann während der Ausführung des Programms nicht verändert werden. *Ein nicht initialisierter String besitzt keinen Speicherplatz außer für den Zeiger selbst* – dies wird von Anfängern häufig vergessen. Bei der Eingabe mit `cin` muss darauf geachtet werden, dass genügend Platz zur Verfügung steht, andernfalls ist das Programmverhalten unvorhersehbar.

```
// 1.
char* x;
cin >> x; // Fehler!
```

```
// 2.
x = "12345678";
cin >> x; // Fehler!
```

Im ersten Fall zeigt `x` auf eine undefinierte Stelle irgendwo im Speicher, an die die eingelesenen Zeichen geschrieben werden, wobei die vorher dort gespeicherten Informationen vernichtet werden! Im zweiten Fall wäre das Einlesen von maximal 8 Zeichen nur möglich, wenn das C++-System das Literal nicht in einem read-only-Bereich abgelegt haben sollte – dies ist normalerweise nicht der Fall. Dabei werden jedoch führende sogenannte *Zwischenraumzeichen* (englisch *whitespace*) von dem `>>`-Operator ignoriert. Zwischenraumzeichen ist ein Sammelbegriff für Leerzeichen, Tabulatorzeichen usw. Mit Zwischenraumzeichen wird die Eingabe für `cin` beendet, Einzelheiten dazu siehe in Abschnitt 2.1. Wenn wir mehr Platz bereitstellen wollen, sollte ein `char`-Array (siehe eine halbe Seite weiter) ausreichender Größe als Eingabepuffer deklariert werden:

```
// 3.
char bereich[1000];
cin >> bereich; // relativ sicher, liest bis zum 1. Zwischenraumzeichen
```

Wie eine ganze Zeile inklusive aller Zwischenraumzeichen und gleichzeitig sehr sicher eingelesen wird, sehen Sie im 4. Beispiel:

```
// 4.
const int ZMAX = 100;
char zeile[ZMAX];
// sicher, liest eine Zeile, aber maximal (ZMAX - 1) Zeichen:
cin.getline(zeile, ZMAX);
```

Die Konstante `ZMAX` muss nur groß genug gewählt werden. Günstiger ist es in der Regel, `string`-Objekte einzulesen, wie auf Seite 95 beschrieben. Weitere Einzelheiten zur Methode `getline()` sind in Abschnitt 10.2 zu finden.

char-Arrays

Eine Art Zwitterstellung nehmen die `char`-Arrays ein. Genau wie bei Arrays wird Speicherplatz zur Compilierzeit reserviert. Wie Arrays können `char`-Arrays keine L-Werte sein. Wie bei Strings nimmt der Ausgabeoperator `<<` bei `cout` an, dass mit dem `char`-Array eine Zeichenkette gemeint ist, die mit `'\0'` abschließt. Die Initialisierung kann wie bei C-Strings geschehen oder wie bei Arrays vorgenommen werden. Im Folgenden werden einige Beispiele für die verschiedenen Fälle aufgelistet.

```
// Definition und Initialisierung
char str_a [ ] = "noch ein String";
char str_b [9];           // 9 Byte reservieren, d.h. 8 Zeichen 0 bis 7 sowie '\0'
char str_c [9] = "ABC";   // 9 Byte, aber nur die ersten 4 haben definierte Werte
cin >> str_b;             // Eingabe wie bei C-Strings. Länge beachten!

// Aliasing, d.h. mehr als einen Namen für nur eine Sache
const char * str1 = "Guten Morgen!\n";
const char * str2 = str1; // str2 zeigt auf dieselbe Stelle wie str1.
cout << str2;             // Beweis: Guten Morgen!

// erlaubte Zuweisung
char charArray1[ ] = "hallo\n";
str1 = charArray1;        // Zeiger str1 zeigt nun auf Array-Anfang
++str1;                  // str1 zeigt jetzt auf das nächste Zeichen
```



```
// nicht erlaubt, weil ein Array kein L-Wert ist:
charArray1 = str1;      // Fehler!
++charArray1;          // Fehler!

// Definition
char buchstaben[3] = "abc"; // Fehler! (kein Platz für '\0')
// tolerante Compiler ignorieren die '3'. Besser:
char buchstaben[4] = "abc"; // oder
char buchstaben[ ] = "abc"; // Compiler zählen lassen, oder
char buchstaben[3] = {'a', 'b', 'c'}; // ohne '\0'-Terminierung!
```

Beispiele: Schleifen mit Strings

Trotz vorhandener Funktionen für verschiedene Zwecke der Stringbearbeitung werden im Folgenden zur Vertiefung einige Operationen mit Strings ausführlich diskutiert, weil sie in ihrer Art typisch sind und unterschiedliche Gestaltungsmöglichkeiten für Schleifen aufzeigen. Als Erstes soll die Länge einer Zeichenkette auf verschiedene Weisen bestimmt werden. Nach jeder Version enthält `sl` die Stringlänge.

Stringlänge berechnen

Version 1

```
const char *str1 = "pro bonum, contra malum";
const char *temp = str1;
int sl = 0;
while(*temp) {
    ++sl;
    ++temp;
}
cout << "Stringlänge von " << str1 << '=' << sl << endl;
```

Die Variable `temp`, die ebenfalls auf die Zeichenfolge zeigt, ist notwendig, damit `str1` nicht geändert zu werden braucht und am Ende weiterhin auf den Beginn der Zeichenkette zeigt. Die mit 0 initialisierte Variable `sl` steht für die Stringlänge.

Was geschieht nun in der Schleife? Zunächst wird der Wert an der Stelle `temp` geprüft. Die Auswertung der Bedingung ergibt *wahr*, weil der erste Wert `*temp` mit dem ersten Zeichen 'p' der Zeichenkette identisch und damit ungleich 0 ist. `sl` wird daher um 1 erhöht, und `temp` wird auf das nächste Zeichen ('r') gerichtet. Dieser Ablauf wird so lange wiederholt, bis `*temp == '\0'` gilt, also `temp` auf das Ende des Strings zeigt. `sl` enthält jetzt die Anzahl der Zeichen in `str1`.

Version 2

```
const char * str2 = "Lieber reich und gesund als arm und krank";
const char *temp = str2;
int sl = 0;
while(*temp++)
    ++sl;
```

Die `while`-Schleife ist etwas kürzer durch den Seiteneffekt, dass `temp` nach Auswerten *in* der Bedingung inkrementiert wird. Zur Bedeutung von `*temp++` siehe die Erläuterung am Ende von Abschnitt 5.2.4.

Version 3

```
const char * str3 = "Morgenstund ist aller Laster Anfang";
int sl = 0;
while(str3[sl++]);
--sl;
```

Diese `while`-Schleife hat keine Anweisung mehr im Schleifenkörper, weil alles Nötige bereits *innerhalb der Bedingung* getan wird. Ein temporärer Zeiger ist nicht notwendig, weil über den Indexoperator auf die Elemente der Zeichenkette zugegriffen wird. Weil `sl` in jedem Fall beim Auswerten der Bedingung inkrementiert wird, also auch am Ende des Strings, muss die letzte Inkrementierung wieder rückgängig gemacht werden. Durch eine Inkrementierung noch *vor* Auswertung der Bedingung lässt sich das vermeiden:

```
int sl = -1;
while(str3[++sl]);
```

Version 4

```
const char * str4 = "letztes Beispiel zur Stringlängenberechnung";
const char *temp = str4;
int sl;
while(*temp++);
sl = temp-str4-1;
```

Diese ebenfalls sehr kurze Formulierung lässt einfach den Zeiger `temp` bis zum terminierenden Zeichen `'\0'` laufen. Die Stringlänge entspricht dann der Differenz der Zeiger, korrigiert um das `'\0'`-Zeichen.

Nachdem nun bekannt ist, wie die Länge einer C-Zeichenkette ermittelt wird, empfiehlt sich für den weiteren Gebrauch die schon aus dem Beispiel von Seite 194 bekannte Funktion `strlen(const char*)`.

Strings kopieren

Nehmen wir an, wir hätten zwei Strings deklariert, und der zweite soll ein Duplikat des ersten werden.

```
const char* original = "Ich verstehe mich! (G. Ch. Lichtenberg, geb. 1742)";
char* duplikat;
```

Die Zuweisung `duplikat = original;`, wie sie in anderen Programmiersprachen wie Pascal benutzt wird, hätte hier nicht den gewünschten Effekt, denn es wird *nur der Zeiger kopiert*, das heißt, `duplikat` zeigt auf denselben Speicherbereich wie `original`. Wenn eine C-Zeichenkette dupliziert werden soll, muss man sie *elementweise* in einen vordefinierten Speicherbereich kopieren. In diesem Fall ist keine Zieladresse definiert und nicht genügend Speicher vorhanden, weil `duplikat` bei der Definition nicht entsprechend initialisiert wurde. Es muss also zusätzlich Sorge getragen werden, dass der `duplikat` zur

Verfügung stehende Speicherplatz *mindestens* so groß ist wie der von `original`. Es gibt für Kopierzwecke ebenfalls *vordefinierte* Funktionen, auf die später noch eingegangen wird.

Das Kopieren eines Strings wird auf vier Arten mit `while` gezeigt (do `while`-Varianten sind natürlich auch möglich). Vorangestellt sind die allen Versionen gemeinsamen Definitionen, wobei sichergestellt sein muss, dass der Speicherplatz des Zielbereichs `dest` ausreichend ist.

Definitionen:

```
const char * source = "Unter allen Tieren steht der Mensch dem "
                    "Affen am nächsten.";
char dest[80];      // Platz muss reichen!
```

Version 1

```
int i = 0;
while(source[i] != '\0') { dest[i] = source[i]; ++i;}
dest[i] = '\0';
```

In der Schleife wird jedem Element von `dest` das entsprechende von `source` zugewiesen. Weil die Zuweisung wegen der Schleifenbedingung nicht mehr für `'\0'` durchgeführt wird, wird die Endekennung anschließend eingetragen.

Version 2

```
int i = -1;
while(source[++i]) dest[i] = source[i];
dest[i] = '\0';
```

Dieses Beispiel unterscheidet sich vom vorhergehenden nur dadurch, dass die Inkrementierung von `i` als Seiteneffekt in die Bedingung verlegt und auf den Vergleich mit `'\0'` verzichtet wird. Es wird der Wert von `source[++i]` ausgewertet.

Version 3

```
const char *s = source;
char *d = dest;
while(*s) {*d = *s; ++s; ++d;}
*d = '\0';
```

Die Hilfszeiger `s` und `d` werden auf die Anfänge des Quell- und Zielbereichs gesetzt. In der Bedingung wird das Zeichen, auf das `s` zeigt, geprüft. Die Bedingung ist so lange wahr, bis `s` auf `'\0'` zeigt. Wenn die Bedingung wahr ist, wird jedesmal mit `*d=*s`; der Wert an der Stelle `d` gleich dem Wert an der Stelle `s` gesetzt, anschließend werden `s` und `d` um 1 weitergezählt. Auch hier wird die Zuweisung von `'\0'` nicht mehr innerhalb der Schleife vorgenommen, sodass sie nachgeholt wird. `s` wird eingeführt, damit `source` nicht verändert werden muss. Der Hilfszeiger `d` ist notwendig, weil `dest` als Array nicht veränderbar ist.

Version 4

```
const char *s = source;
char *d = dest;
while(*d++ = *s++);
```

Noch kürzer geht es nicht! Der Unterschied zur vorangehenden Version liegt darin, dass sämtliche Aktivitäten in die Bedingung verlegt werden. Alle Seiteneffekte werden ausgeführt einschließlich der Bedingung, die die Schleife zum Abbruch bringt. Die Anweisung `*d = '\0';` ist nun nicht mehr gesondert notwendig, weil sie die letzte während der Bedingungsauwertung ist. Am Ende der Schleife zeigen `s` und `d` auf die Stellen direkt nach den `'\0'`-Zeichen.

Die sehr kompakte Schreibweise ist für Anfänger auf den ersten Blick oft schwer zu verstehen. Andererseits ist diese Art der Formulierung ein *Idiom* in C und C++, das jeder erfahrene C/C++-Programmierer kennt, weswegen man sich damit vertraut machen sollte. Die Anweisung `while(*d++=*s++)` kann als Abkürzung einer `do while`-Schleife aufgefasst werden. Zur Erläuterung wird der Ablauf dieser kurzen Anweisung in Einzelschritte aufgelöst formuliert:

```
bool x;           // Hilfsvariable
do {
    *d = *s;       // Zuweisung eines Zeichens
    x = (*d != '\0'); // Ergebnis des Vergleichs in x merken
    ++d;
    ++s;
} while(x);
```

Nachdem nun bekannt ist, wie eine C-Zeichenkette kopiert wird, empfiehlt sich für den weiteren Gebrauch die Funktion `strcpy(char* ziel, const char* quelle)`.

C-String-Arrays

Die Elemente eines Arrays können auch C-Strings sein. Abbildung 5.6 und das Beispielprogramm verdeutlichen die Datenstruktur. Die Abbildung zeigt ein String-Array `sa` und einen Zeiger `sp` auf den Anfang des Feldes. `sa` ist ein symbolischer Name für den Feldanfang und wird syntaktisch wie ein nichtveränderbarer Zeiger auf den Feldanfang behandelt. Der Zugriff auf Feldelemente über Array-Indizes oder Zeiger ist äquivalent.

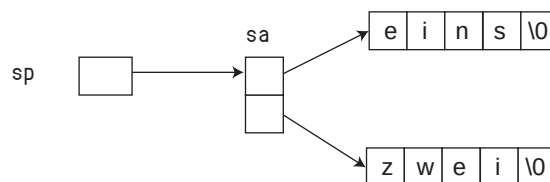


Abbildung 5.6: String-Array

Listing 5.2: C-String-Array

```
// cppbuch/k5/strarray.cpp  Bezug: Abbildung 5.6
#include<iostream>
using namespace std;

int main() {
    const char* sa[] = {"eins\n", "zwei\n"}; // Array
    const char** sp = sa; // Zeiger auf const char*.
                                // Programmausgabe:
    cout << sa[0] << endl; // eins
    cout << *sa << endl; // eins
    cout << sa[1] << endl; // zwei
    cout << sa[1][0] << endl; // z
    cout << *sp << endl; // eins
    cout << sp[1] << endl; // zwei
}
```

5.4 Dynamische Datenobjekte

Bisher wurden nur Datentypen behandelt, deren Speicherplatzbedarf bereits zur Compilierzeit berechnet und damit vom Compiler eingeplant werden konnte. Nicht immer ist es jedoch möglich, den Speicherplatz exakt vorherzuplanen, und es ist unökonomisch, jedesmal mit großen Arrays sicherheitshalber den maximalen Speicherplatz zu reservieren. C++ bietet daher die Möglichkeit, mit dem Operator `new` Speicherplatz genau in der richtigen Menge und zum richtigen Zeitpunkt bereitzustellen und diesen Speicherplatz mit `delete` wieder freizugeben, wenn er nicht mehr benötigt wird. Damit unterliegen die mit `new` erzeugten Objekte *nicht* den Gültigkeitsbereichsregeln für Variablen. `new` erkennt die benötigte Menge Speicher am Datentyp, sie muss also nicht explizit angegeben werden. Es gibt einen vom Betriebs- oder Laufzeitsystem verwalteten großen zusammenhängenden Adressbereich, der von [Str] *free store*, *dynamic store* oder *heap*¹ genannt wird. Ich verwende im Folgenden durchgängig die letzte Bezeichnung, die deutsch *Halde* oder *Haufen* bedeutet. Innerhalb des Heaps wird mit `new` der Platz für ein Objekt reserviert. Der Zugriff auf die neu auf dem Heap erzeugten Objekte geschieht ausschließlich über Zeiger. Abbildung 5.7 visualisiert das nachfolgende Code-Beispiel.

```
int *p; // Zeiger auf int
p = new int; // int-Objekt erzeugen
*p = 15; // Wert zuweisen
cout << *p << endl; // 15
```

Nur der Platz für `p` wird zur Compilierzeit eingeplant. Mit `p = new int;` wird Speicherplatz in der Größe von `sizeof(int)` Byte erst *zur Laufzeit* des Programms bereitgestellt, und `p`

¹ Manche unterscheiden zwischen Heap und Free Store, weil auch mit der C-Funktion `malloc` Speicher beschafft werden kann und es sein kann, dass `new` und `malloc` verschiedene Speicherbereiche nutzen. Empfehlung: Verzicht auf `malloc`, wenn es keinen zwingenden Grund für die Verwendung gibt.



Abbildung 5.7: Erzeugen eines `int`-Datenobjekts mit `new`

zeigt nach dieser Operation auf diesen Platz. `*p` (Wert an der Stelle `p`) kann als Name für das Objekt interpretiert werden. Mit `new` kann dynamisch ein Array erzeugt werden, wobei dann eckige Klammern `[ ]` angegeben werden. Der Operator `new [ ]` zur Erzeugung von Arrays wird in C++ vom `new`-Operator für einzelne Objekte unterschieden. Hier wird ein Feld von 4 `int`-Zahlen bereitgestellt (siehe auch Abbildung 5.8).

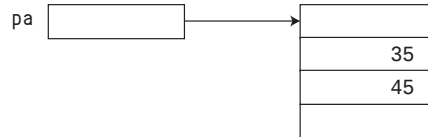


Abbildung 5.8: Erzeugen eines `int`-Arrays mit `new [ ]`

```

// Operator new [ ]
int *pa = new int[4]; // Array von int-Zahlen
pa[1] = 35;
pa[2] = 45;
cout << pa[1] << endl; // 35
  
```

Der Zugriff auf die Elemente geschieht wie bei dem statisch deklarierten Array. Im Beispiel enthalten die Arrayelemente 0 und 3 undefinierte Werte. Das Programmstück zeigt beispielhaft einige Möglichkeiten des Gebrauchs von Zeigern für Objekte, die mit `new` erzeugt wurden:

```

// etwas komplizierteres Beispiel zur Übung
int **pp = new int*[4]; // Array von Zeigern auf int-Zahlen
pp[0] = p; // pp[0] zeigt auf *p
pp[1] = &pa[2]; // pp[1] zeigt auf pa[2] (s.o.)
cout << *pp[0] << endl; // 15
cout << **pp << endl; // 15
cout << *pp[1] << endl; // 45
  
```

Das zweifache Sternchen bei der Deklaration mag etwas ungewohnt erscheinen. `int **pp` ist gleichbedeutend mit `(int*)* pp`, das heißt, `pp` ist ein »Zeiger auf Zeiger auf `int`«, weil die Array-Elemente selbst Zeiger sind. Sie sind nach der Deklaration alle noch undefiniert, verweisen also nicht auf bestimmte Speicherplätze. Das ändert sich nur für die ersten beiden Elemente, die nach den Zuweisungen auf die oben definierten Plätze `*p` und `pa[2]` verweisen (siehe Abbildung 5.9).

Dynamisch erzeugte Struktur

Die Struktur `test_struct` enthält eine `int`- und eine `double`-Variable und außerdem einen Zeiger `next` auf ein gleichartiges Objekt.

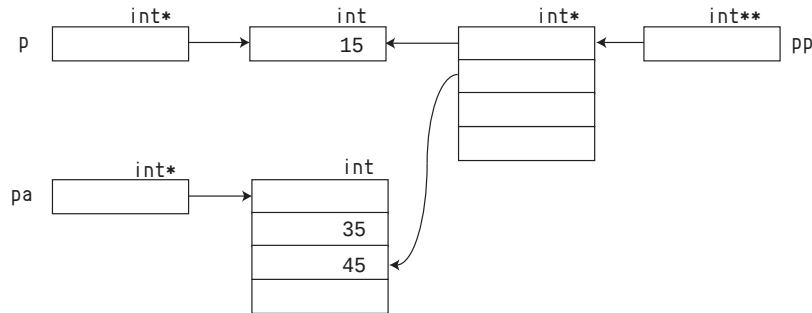


Abbildung 5.9: Array von int-Zeigern zum Programmbeispiel

```
struct test_struct {
    int a;
    double b;
    test_struct* next;
};

test_struct *sp= new test_struct;
```

Der Zeiger `sp` verweist auf ein neu erzeugtes Objekt von der Größe der Struktur `sizeof(test_struct)`, der Summe `sizeof(int) + sizeof(double) + sizeof(test_struct*)` entsprechend. Der Zugriff auf die Elemente der Struktur geschieht über den Pfeil-Operator `->`:

```
sp->a = 12;      // entspricht (*sp).a
sp->b = 17.4;
```

Dem Zeiger `sp->next` wird mit `sp->next = new test_struct;` ein weiteres neu erzeugtes Objekt zugewiesen, sodass jetzt zwei miteinander verkettete Objekte vorliegen, die beide über den Zeiger `sp` erreichbar sind (Abbildung 5.10).

```
// weiteres Objekt erzeugen
sp->next = new test_struct;
// Zugriff auf Element a des neuen Objekts
sp->next->a = 144;
```

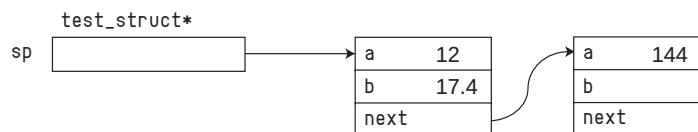


Abbildung 5.10: Struktur mit Verweis auf zweite Struktur

Interne Datenelemente des zweiten Objekts können durch entsprechende Anweisungen der Art `sp->next->a = 144;` erreicht werden. Der beschriebene Verkettungsmechanismus wird zum Aufbau von Listen verwendet.

5.4.1 Freigeben dynamischer Objekte

Der `delete`-Operator gibt den reservierten Platz wieder frei, damit er von Neuem belegt oder anderen Programmen zur Verfügung gestellt werden kann. Nach `delete` wird der Zeiger angegeben, der auf das zu löschende Objekt verweist. Im Folgenden werden alle oben erzeugten Objekte der Reihe nach gelöscht. Dabei könnte man anstatt `delete p` auch `delete pp[0]` schreiben, weil `pp[0]` auf dasselbe Objekt verweist, aber nicht beides. Das Löschen des Objekts, auf das `sp->next` zeigt, muss vor dem Löschen von `*sp` erfolgen, weil sonst mit der Vernichtung von `*sp` die Information über den Ort des über `next` verketteten Objekts verloren wäre.

In Entsprechung zu den Operatoren `new` und `new [ ]` wird zwischen den Operatoren `delete` und `delete [ ]` unterschieden.

```
// Freigaben zu Abbildung 5.9
delete p;
delete [ ] pa;           // Array löschen
delete [ ] pp;           // Array löschen
```

```
// Freigaben zu Abbildung 5.10. Reihenfolge beachten!
delete sp->next;          // muss zuerst kommen
delete sp;
```

Falls es nicht gelingt, mit `new` Speicher zu beschaffen, zum Beispiel weil schon zu viel verbraucht und nicht wieder freigegeben wurde, wird das Programm mit einer Fehlermeldung abgebrochen (Einzelheiten siehe in Abschnitt 8.2).

Einige Dinge sollten bei der Verwendung von `new` und `delete` beachtet werden, deren Missachtung oder Unkenntnis in manchen Fällen ein unvorhersehbares Programmverhalten nach sich zieht, leider *ohne* Fehlermeldung seitens des Compilers oder des Laufzeitsystems.

- `delete` darf *ausschließlich* auf Objekte angewendet werden, die mit `new` erzeugt worden sind.

```
int i;
int *iptr = &i;
delete iptr;          // Fehler! (Absturzgefahr)
iptr = new int;
delete iptr;           // ok!
```

- `delete` darf nur *einmal* auf ein Objekt angewendet werden. Der Wert eines Zeigers, auf den `delete` angewendet wurde, ist danach *undefiniert*, er ist also leider nicht gleich `NULL` bzw. `0`. Falls zwei oder mehr Zeiger auf ein Heap-Objekt zeigen, bewirkt ein `delete` auf nur einen von diesen Zeigern die Zerstörung des Objekts. Die anderen Zeiger verweisen danach nicht mehr auf einen Speicherbereich mit Bedeutung, sie heißen dann »hängende Zeiger« (englisch *dangling pointer*).
- `delete` auf einen Null-Zeiger angewendet, bewirkt nichts und ist unschädlich.
- Wenn ein mit `new` erzeugtes Objekt mit dem `delete`-Operator gelöscht wird, wird automatisch der Destruktor für das Objekt aufgerufen.
- Mit `new` erzeugte *Objekte* unterliegen *nicht* den Gültigkeitsbereichsregeln für Variablen. Sie existieren so lange, bis sie mit `delete` gelöscht werden oder das Programm

beendet wird, unabhängig von irgendwelchen Blockgrenzen. Dies gilt nicht für die statisch deklarierten *Zeiger* auf diese Objekte, für die die normalen Gültigkeitsbereichsregeln gelten. Als Konsequenz ist darauf zu achten, dass ein Zeiger auf ein Heap-Objekt mindestens bis zu dessen Löschung existiert, damit überhaupt eine Chance besteht, das Objekt zu löschen. Das folgende Programmfragment verdeutlicht das Problem:

```
{ // Blockanfang
  // ungünstig
  int *p = new int;
  *p = 30000;
  // mehr Programmcode
}
```

Nach Verlassen des Blocks existiert *p* nicht mehr; das Objekt mit dem ehemaligen Namen **p* existiert noch, ist aber nicht mehr erreichbar. In Schleifen angewendet, kann diese Methode schnell zur Speicherknappheit führen. Es kommt vor, dass ein rund um die Uhr laufendes Programm nach einer Woche plötzlich ohne erkennbare Ursache stehenbleibt, wofür wir nun einen möglichen Grund kennen. Ein nicht mehr zugänglicher Bereich wird »verwitwetes« Objekt genannt, im Englischen »Speicherleck« *memory leak*. Richtig ist:

```
{
  int *p = new int;
  *p = 30000;
  // mehr Programmcode
  delete p;           // *p wird nicht mehr gebraucht
}
```

oder:

```
int *p;                // außerhalb des Blocks deklariert
{
  p = new int;
  *p = 30000;
  // mehr Programmcode
}
cout << *p;           // weitere Verwendung von p
// mehr Programmcode
delete p;
```

- Die Freigabe von Arrays erfordert die Angabe der eckigen Klammern (siehe obiges Beispiel). Ohne `[]` gäbe `delete pa;` nur ein einziges Element frei! Der Rest des Arrays wäre danach nicht mehr zugänglich (verwitwetes Objekt). *Merkregel:* `delete []` dann (und *nur* dann) benutzen, wenn die Objekte mit `new []` erzeugt worden sind.



Hinweis

Wenn statt `delete []` nur `delete` geschrieben wird, sich aber aus dem Kontext ergibt, dass es um die Löschung eines Arrays gehen soll, wird von manchen Compilern die fehlerhafte Schreibweise stillschweigend korrigiert. Nach dem C++-Standard resultiert

eine fehlerhafte `delete`-Anweisung für ein Array (d.h. ohne die eckigen Klammern) aber in »undefiniertem Verhalten« (englisch *undefined behaviour*).

Das bedeutet nicht unbedingt Programmabsturz, sondern dass der Hersteller des Compilers selbst entscheiden kann, was zu tun ist: Fehler selbsttätig korrigieren, Warnung ausgeben, Speicherleck akzeptieren oder was auch immer. Das bedeutet aber auch, dass ein Programm der Art

```
while(true) {                // Programm soll ständig laufen
    int* arr = new int[100000];
    // ... mit dem Array arbeiten
    delete arr; // Klammern [ ] fehlen!
}
```

nicht portabel ist! Bei einem System wird der Fehler toleriert, beim anderen kracht es, weil der Speicher ausgeht. Beides wäre im Sinn des C++-Standards »legal«. Manche Programmiersprachen, wie etwa Java, kennen kein `delete`. Dafür gibt es ein parallel laufendes Programm zur Speicherbereinigung, *garbage collector* genannt (englisch für: »Müllsammler«). Die Speicherverwaltung vieler C++-Systeme enthält aus Effizienzgründen keine Speicherbereinigung – es gibt ja `delete`. C++ erlaubt es jedoch, eine eigene Speicherverwaltung mit diesen und anderen Eigenschaften zu schreiben. Es gibt auch entsprechende Bibliotheken – ein einfacherer Weg, die Möglichkeit zur *garbage collection* zu erhalten.

Neben mangelndem Speicher liegt ein weiterer Grund für das plötzliche unerwartete Stehenbleiben von sehr lange laufenden Programmen in der Zerstückelung (Fragmentierung) des Speichers durch viele `new`- und `delete`-Operationen. Damit ist gemeint, dass sich kleinere belegte und freie Plätze abwechseln, sodass die plötzliche Anforderung eines größeren *zusammenhängenden* Bereichs für ein großes Datenobjekt nicht mehr erfüllbar ist, obwohl die Summe aller einzelnen freien Plätze ausreichen würde. Dieses Problem verlangt ein Zusammenschieben aller belegten Plätze, sodass ein großer freier Bereich entsteht. Programme zur *garbage collection* erledigen diese Aufgabe meistens gleich mit.

5.5 Zeiger und Funktionen

5.5.1 Parameterübergabe mit Zeigern

Wenn ein übergebenes Objekt modifiziert werden soll, kann die Übergabe *per Zeiger* auf das Objekt geschehen. Dies ist ein Spezialfall der Übergabe *per Wert*, es wird nämlich mit einer *Kopie* des Zeigers weitergearbeitet, die auf dasselbe Objekt wie das Original zeigt. Die Übergabe *per Zeiger* ist kein Sprachmittel von C++, sondern zeigt nur eine andere Art der Benutzung der Übergabe *per Wert*. Eine Modifikation des Zeigers in der Funktion ändert also nicht den Wert des Zeigers für den Aufrufer.

Das Objekt, auf das der Zeiger verweist, kann in der Funktion gleichwohl geändert werden, sodass der Zeiger beim Aufrufer zwar auf dieselbe Adresse zeigt, unter dieser Adresse

jedoch ein verändertes Objekt zu finden ist. Im Beispiel ist beides zu sehen: Die lokale Kopie `s` des Zeigers `str` wird verändert, ohne dass `str` verändert wird, aber das Objekt an der Stelle `str` ändert sich, weil alle Klein- durch Großbuchstaben ersetzt werden. Abbildung 5.11 zeigt in Ergänzung zum Programmbeispiel, dass ein Objekt über die Kopie eines Zeigers zugreifbar und modifizierbar ist.

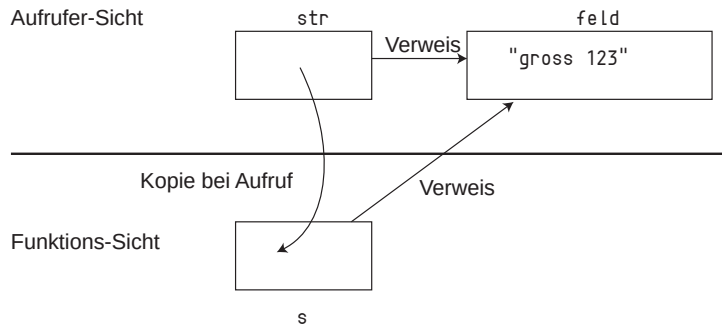


Abbildung 5.11: Parameterübergabe per Zeiger (Bezug: Beispielprogramm)

Listing 5.3: Parameterübergabe per Zeiger

```
// cppbuch/k5/perzeigISO8859.cpp
#include<iostream>
using namespace std;
void upcase(char *); // Prototyp

int main( ) {
    char feld[] = "gross 123"; // Array
    char* str = feld;          // Zeiger
    upcase(str);                // upcase(feld); ist ebenso möglich
    cout << str << endl;      // GROSS 123
}

void upcase(char* s) {
    const int DIFFERENZ = 'a' - 'A'; // In der ASCII-Tabelle sind die Platznummern
    // der Kleinbuchstaben um 'a'-'A' = 32 gegenüber den Großbuchstaben verschoben.
    while(*s) {
        if(*s>='a' && *s<='z') {
            *s -= DIFFERENZ;
        }
        else { // Umlaute
            // Lesbar, aber nicht auf jedem System: Umlaute sind nicht portabel! Der
            // folgende Teil funktioniert nur dann richtig, wenn dieses Programm vor der
            // Compilation dieselbe Zeichenkodierung hat wie das Terminal und die
            // Umlaute als ein Byte darstellbar sind, wie etwa bei der Zeichenkodierung
            // ISO8859-1. Weitere Informationen finden Sie im Abschnitt 31.
            switch(*s) {
                case 'ä' : *s = 'Ä'; break;
```

```

        case 'ö' : *s = 'Ö'; break;
        case 'ü' : *s = 'Ü'; break;
        default:; // nichts tun
    }
}
s++;
}
}

```

Weil ein `char` vom Compiler zwecks Umwandlung wie eine Ein-Byte-`int`-Zahl interpretiert wird, ist das Rechnen ohne explizite Typumwandlung möglich. Um von der internen Darstellung der Zeichen unabhängig zu sein, wird `'a'-'A'` anstatt 32 benutzt. Auch ist es schwierig für einen Leser, bei fehlendem Kommentar die Bedeutung der Zahl 32 zu erraten. Die Umlaute und andere Sonderzeichen werden auf verschiedenen Maschinen mit anderen Betriebssystemen durch andere Codierungen repräsentiert, darunter auch Multi-Byte-Codierungen wie Unicode. Alle Zeichen außerhalb des ASCII-Bereichs sind nicht portabel und müssen ihrer Codierung entsprechend verarbeitet werden.

const und Zeiger-Parameter

Hier seien verschiedene Möglichkeiten der Verwendung von `const` in einer Parameterliste aufgezeigt (vergleiche Seite 189). Es liegt ein C-Array `int tabelle[100]` zugrunde.

- `void tabellenfunktion(int* tabelle)`
Nichts ist konstant, das heißt, in der Funktion können sowohl die Tabellenwerte wie auch der Zeiger `tabelle` geändert werden. Ersteres wirkt sich beim Aufrufer aus, Letzteres nicht, weil in der Funktion eine Kopie des Zeigers angelegt wird (Übergabe des Zeigers per Wert).
- `void tabellenfunktion(const int* tabelle)`
`tabelle` zeigt auf nicht veränderbare Elemente. Eine Anweisung etwa der Art `tabelle[0] = 3`; innerhalb der Funktion würde vom Compiler nicht akzeptiert werden, `++tabelle` wäre hingegen ok.
- `void tabellenfunktion(int* const tabelle)`
`tabelle` ist selbst konstant. Eine Anweisung `tabelle = NULL`; würde vom Compiler nicht akzeptiert werden, `tabelle[10] = 17`; schon.
- `void tabellenfunktion(const int* const tabelle)`
kombiniert die beiden vorhergehenden Möglichkeiten: Ein konstanter Zeiger zeigt auf unveränderliche Werte.

Ein konstanter Zeiger verbietet eine Änderung des Zeigers innerhalb der Funktion. Diese Änderung hätte aber ohnehin keine Auswirkung auf das aufrufende Programm, wie oben erläutert. In der Praxis sind daher nur die ersten beiden Fälle von Bedeutung.

5.5.2 Parameter des main-Programms

`main()` kann über Parameter verfügen, siehe Seite 116. Sie sind innerhalb des Programms auswertbar, wie es in vielen Dienstprogrammen gehandhabt wird. Die Parameter werden beim Aufruf des Programms auf Betriebssystemebene nach dem Programmnamen angegeben, daher der Name »Kommandozeilenparameter«. Der Compilerhersteller kann

weitere Parameter nach `char* argv[]` zulassen, üblich ist ein weiteres String-Array `char* env[]` zum Abfragen von Umgebungsvariablen (englisch *environment variables*):

```
int main( int argc, char* argv[], char* env[]) {
    // ...
}
```

`argc` ist die Anzahl der Kommandozeilenparameter einschließlich des Programmaufrufs, `argv[]` ein String-Array mit den Kommandozeilenparametern. Das letzte Element von `env[]` ist 0, und es gilt `argv[argc] == 0` (Null-Pointer). `argv[0]` enthält den Programmaufruf. Das Programm *mainpar.cpp* demonstriert die Benutzung der Parameter. Auf Betriebssystemebene kann es zum Beispiel mit *mainpar 1 par2 /3 5 5 A6* aufgerufen werden. Die Parameter haben hier keine Bedeutung und dienen nur zur Demonstration.

Listing 5.4: Kommandozeilenparameter anzeigen

```
// cppbuch/k5/mainpar.cpp
#include<iostream>
using namespace std;

int main(int argc, char* argv[], char *env[]) {
    cout << "Aufruf des Programms = " << argv[0] << endl;
    cout << (argc-1) << " weitere Argumente wurden main() übergeben:\n";
    int i = 1;
    while(argv[i]) {
        cout << argv[i++] << endl;
    }
    cout << "\n*** Umgebungs-Variablen: ***\n";
    i = 0;
    while(env[i]) {
        cout << env[i++] << endl;
    }
}
```

5.5.3 Gefahren bei der Rückgabe von Zeigern

Wie bei der Rückgabe von Referenzen (Seite 112) muss auch bei Zeigern darauf geachtet werden, dass sie nicht auf lokale Objekte verweisen, die nach dem Funktionsaufruf verschwunden sind.

Negativ-Beispiel

```
#include<iostream>

char* murks(const char * text) {
    char neu[100];           // Speicherplatz besorgen
    char* n = neu;
    while(*n++ = *text++);   // text wird nach neu kopiert
    return neu;              // Fehler!
}
```

```
int main() {
    char *sp3 = murks("Oh je!");
    std::cout << sp3;      // nicht existierendes Objekt!
}
```

Alternative

Die Funktion `murks()` soll ein Duplikat des übergebenen Strings erzeugen und den Zeiger darauf zurückgeben. Der Speicherplatz für das lokale Feld `neu` wird jedoch bei Verlassen der Funktion freigegeben! Die Kopie wird zerstört. Richtig wäre es, den Speicherplatz mit `new` zu besorgen und den Zeiger `neu` zurückzugeben:

```
char* kein_murks(const char * text) {
    char *neu = new char[strlen(text) + 1]; // strlen() erfordert #include<cstring>
    char* n = neu;
    while(*n++ = *text++);                // text wird nach neu kopiert
    return neu;
} // Der Aufrufer muss für die Speicherfreigabe sorgen!
```

Ferner soll kein Objekt per `return` zurückgegeben werden, das mit dem `new`-Operator in einer Funktion erzeugt wurde. Der Grund liegt darin, dass bei der Rückgabe mit `return` ein Objekt *kopiert* wird. Das Original wäre für ein notwendiges `delete` nicht mehr erreichbar. Es darf also nur der Zeiger auf ein mit `new` erzeugtes Objekt zurückgegeben werden, wobei der Aufrufer die Verantwortung hat, das Objekt irgendwann zu löschen.

5.6 this-Zeiger

`this` ist ein Schlüsselwort, das innerhalb einer Elementfunktion einen *Zeiger auf das Objekt* darstellt. Weil `this` der Zeiger auf das Objekt ist, wird das Objekt selbst durch `*this` benannt, ganz in Analogie zu den Zeigern, die wir schon kennen: `*ptr = 3`; weist den Wert 3 dem Objekt zu, auf das `ptr` zeigt. Innerhalb einer Methode bezeichnet `this` einen Zeiger auf das aktuelle Objekt und `*this` das Objekt selbst, für das die Methode aufgerufen wird. `*this` ist nur ein anderer Name (Alias-Name) für das Objekt, der innerhalb der Elementfunktionen benutzt werden kann.

5.7 Mehrdimensionale C-Arrays

5.7.1 Statische mehrdimensionale C-Arrays

Gelegentlich hat man es mit mehrdimensionalen Feldern zu tun, am häufigsten mit zweidimensionalen. Eine zweidimensionale Tabelle besteht aus Zeilen und Spalten, eine dreidimensionale aus mehreren zweidimensionalen Tabellen. In C++ wird eine Tabelle linear

auf den Speicher abgebildet. Im Fall der zweidimensionalen Tabelle kommen zunächst alle Elemente der ersten Zeile, dann alle Elemente der zweiten Zeile usw. Ein zweidimensionales Array ist ein Array von Arrays. Das folgende Beispiel zeigt eine statisch angelegte zweidimensionale Tabelle.

Listing 5.5: Zweidimensionale Matrix

```
// cppbuch/k5/matrix2d.cpp
#include<iostream>
using namespace std;

int main() {
    const size_t DIM1 = 2;
    const size_t DIM2 = 3;
    int matrix [DIM1] [DIM2] = { {1,2,3}, {4,5,6} };
    for (size_t i = 0; i < DIM1; ++i) {
        for (size_t j = 0; j < DIM2; ++j) {
            cout << matrix[i][j] << ' ';
        }
        cout << endl;
    }
}
```

Die Matrix besteht aus zwei Zeilen und drei Spalten. Die Struktur wird auch in der Initialisierungsliste deutlich. Das Weglassen der inneren geschweiften Klammern, das heißt {1, 2, 3, 4, 5, 6}, hätte die gleiche Wirkung. Auch hier gilt, dass nicht aufgeführte Elemente der Liste mit 0 initialisiert werden. {{1}, {4}} ist gleichbedeutend mit {{1, 0, 0}, {4, 0, 0}}, also Initialisierung der ersten Spalte und Nullsetzen des Rests.

Die Konstanten DIM1 und DIM2 müssen zur Compilationszeit bekannt sein. In Abschnitt 5.7.2 werden Sie sehen, wie die Arraygröße erst zur Laufzeit des Programms definiert werden kann. Mehrdimensionale Arrays haben für *jede Dimension* ein Klammernpaar []. In C++ darf der Zugriff auf ein Element eines mehrdimensionalen Arrays nicht mit einem Komma abgekürzt werden, wie es in anderen Programmiersprachen manchmal der Fall ist. Es darf nicht `a[2, 3]` statt `a[2][3]` geschrieben werden. Der Compiler meldet die falsche Schreibweise *nicht* unbedingt, weil das Komma einen besonderen Operator darstellt und der aus semantischer Sicht falsche Ausdruck syntaktisch erlaubt sein kann.

Der Kommaoperator gibt eine Reihenfolge von links nach rechts vor. Der Ausdruck `sum = (total = 3) + (++total)` von Seite 57, in dem die Auswertungsreihenfolge der Klammerausdrücke nicht definiert ist, kann mit dem Kommaoperator in einen definierten Ausdruck verwandelt werden: `sum = (total = 3, total + ++total)`. Das Ergebnis des gesamten Ausdrucks ist das Ergebnis des Teilausdrucks nach dem letzten Komma. Im obigen Fall würde fälschlicherweise `a[2, 3]` als `a[3]` interpretiert werden.



Tipp

Große Arrays sollten *dynamisch*, wie in Abschnitt 5.7.2 beschrieben, angelegt werden, weil auf dem Heap im Allgemeinen mehr Platz als auf dem Stack ist.

Array als Funktionsparameter

Wie schon in Abschnitt 5.2 erwähnt, sind C-Arrays syntaktisch mit konstanten Zeigern gleichzusetzen. Die Parameterliste (`int tabelle[]`) einer Funktion ist daher dasselbe wie (`int* tabelle`). Damit kann `sizeof` nicht zur Größenermittlung eines Arrays benutzt werden, wie das folgende Beispiel zeigt:

```
// fehlerhaftes Beispiel
void tabellenausgabe(int tabelle[]) { // C-Array-Deklaration
    int anzahlBytes = sizeof tabelle; // Fehler! Grund: dasselbe wie sizeof(int*)!
    int wieoft = anzahlBytes/sizeof(int);
    for(int i = 0; i < wieoft; ++i) {
        cout << tabelle[i] << endl;
    }
}

int main() {
    const int ANZAHL = 5;
    int tabelle[ANZAHL];          // C-Array-Definition
    // ... Berechnungen
    tabellenausgabe(tabelle);    // leider falsch ...
}
```

`sizeof tabelle` innerhalb der Funktion `tabellenausgabe` gibt nur den Platzbedarf für einen Zeiger zurück, weil syntaktisch `int tabelle[]` dasselbe wie `int *tabelle` ist! `sizeof` kann den benötigten Speicherplatz nur im Sichtbarkeitsbereich der *Definition* eines C-Arrays erkennen, also dort, wo Speicherplatz tatsächlich beschafft wird (wie etwa auf Seite 191). Jede andere Stelle, wo ein C-Arrayname eingeführt wird, ist eine *Deklaration* ohne gleichzeitige Definition (vergleiche die »one definition rule« auf Seite 127).



Merke:

Die Anzahl der Array-Elemente muss einer Funktion übergeben werden! Dies gilt auch für mehrdimensionale Arrays.

```
// korrigiertes Beispiel
void tabellenausgabe1D(int tabelle[], size_t n) {
    for(size_t i = 0; i < n; ++i) {
        cout << tabelle[i] << endl;
    }
}
```

Wie sieht es bei einem zwei- oder mehrdimensionalen Array aus? Das ist ein Array von Arrays, das heißt, hier gibt es eine zu übergebende Anzahl von Arrays (statt Werten wie im eindimensionalen Fall). Wie erfährt die Funktion von der Größe der Arrays? Bei statischen, also nicht mit `new` erzeugten Arrays, geht die Größeninformation mit in den Typ ein (zu den dynamischen Arrays siehe unten). Beispiel:

```
int feld1[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

ist ein Array, das 2 Elemente enthält, nämlich zwei Arrays zu je 3 `int`-Werten (2 Zeilen, 3 Spalten). Der Typ der zuletzt genannten Arrays ist *Bestandteil des Arraytyps* von `feld1`. Dies gilt entsprechend für Zeiger:


```
// kompatibler Zeiger
int (*pFeld)[3] = feld1; // zeigt auf Zeile feld1[0]
```

Die Funktion zur Ausgabe dieses Feldes benötigt diese Typinformation in der Parameterliste:

```
void tabellenausgabe2D(int (*T)[3], size_t n) {
    // alternativ: void tabellenausgabe(int T[][3], size_t n)
    for(size_t i = 0; i < n; ++i) {
        for(size_t j = 0; j < 3; ++j) {
            cout << T[i][j] << ' ';
        }
        cout << endl;
    }
    cout << endl;
}
```

Mit dieser Typinformation weiß der Compiler, wo jede Zeile anfängt. Innerhalb der Funktion ist `sizeof T[0]` gleich 3 mal `sizeof(int)`. Die Funktion kann für das Array oder den Zeiger gleichermaßen aufgerufen werden:

```
tabellenausgabe2D(feld1, 2); // 1 2 3
                             // 4 5 6
++pFeld;                     // zeigt jetzt auf Zeile feld1[1]
tabellenausgabe2D(pFeld, 1); // 4 5 6
```

Dieses Schema setzt sich für mehrere Dimensionen fort. Die Information `[3][4]` gehört zum Typ einer dreidimensionalen Matrix `int Matrix3D[2][3][4]`. Die Funktion zur Ausgabe dieser Matrix hat die Schnittstelle

```
void tabellenausgabe3D(int (*T)[3][4], size_t n) ; // oder
void tabellenausgabe3D(int T[][3][4], size_t n) ;
```

Wie schreibt man eine Funktion, die für statische zwei-dimensionale Arrays geeignet ist, egal wie groß die Spaltenanzahl ist? Die Lösung ist ein Template. Der Compiler leitet dann den Feldtyp bei Aufruf der Funktion aus dem Parameter ab.

```
template<typename Feldtyp>
void tabellenausgabe2D(Feldtyp tabelle, size_t n) {
    const size_t SPALTEN = sizeof tabelle[0] / sizeof tabelle[0][0];
    for(size_t i = 0; i < n; ++i) {
        for(size_t j = 0; j < SPALTEN; ++j) {
            cout << tabelle[i][j] << ' ';
        }
        cout << endl;
    }
    cout << endl;
}
```

Interpretation von `[ ]`, `[ ][ ]` usw.

Der Compiler wandelt die Indexoperatoren mehrdimensionaler Arrays wie bei den eindimensionalen Arrays in die Zeigerdarstellung um. Wenn wir mit `x` alles bezeichnen, was

vor dem letzten Klammerspaar steht, und mit γ den Inhalt des letzten Klammerspaars, so wird vom Compiler $X[\gamma]$ in $*(X) + (\gamma)$ umgesetzt. Auf X wird das Verfahren wiederum angewendet, bis alle Indexoperatoren aufgelöst sind. Man kann daher statt `matrix[i][j]` ebenso `*(matrix[i]+j)` oder `*(*(matrix+i)+j)` schreiben. `matrix[i]` ist als Zeiger auf den Beginn der i -ten Zeile zu interpretieren. Durch die Zeigerarithmetik wird die dahinterstehende Berechnung der tatsächlichen Adresse verborgen, die ja noch die Größe der Datenelemente eines Arrays berücksichtigen muss. Die Position `(matrix + i)` liegt daher $(i \text{ mal } \text{sizeof}(\text{matrix}[0]))$ Bytes von der Stelle `matrix` entfernt. Dies wird bei der Ermittlung der Anzahl der Spalten im obigen Funktions-Template ausgenutzt.



Übungen

5.1 Auf Seite 191 wird über die Äquivalenz von `*(kosten+i)` und `kosten[i]` gesprochen. Anstatt `(kosten+i)` könnte man genauso gut `(i+kosten)` schreiben, das Ergebnis der Addition wäre das gleiche. Ist es dann richtig, dass die Schreibweise `i[kosten]` äquivalent ist zu `kosten[i]`?

5.2 Geben Sie den für `matrix[2][3]` benötigten Speicherplatz in Bytes an, wenn `sizeof(int)` als 4 angenommen wird. An welcher Bytenummer beginnt das Element `matrix[i][j]` relativ zum Beginn des Arrays?

5.3 Schreiben Sie die Multiplikation zweier Matrizen `a[n][m] * b[p][q]`. Das Ergebnis soll in einer Matrix `c[r][s]` stehen. Welche Voraussetzungen gelten für die Zeilen- und Spaltenzahlen n, m, p, q, r, s ?

5.4 Schreiben Sie zur Ausgabe von dreidimensionalen Arrays eine Template-Funktion `tabellenausgabe3D(Feldtyp T, size_t n)` entsprechend dem obigen Muster für zwei Dimensionen.

5.7.2 Dynamisch erzeugte mehrdimensionale Arrays

Mehrdimensionale Arrays mit konstanter Feldgröße

Auf Seite 209 haben Sie die statische Deklaration von mehrdimensionalen Feldern gesehen. Im Abschnitt 5.4 wurden ein Array von `int`-Zahlen und ein Array von Zeigern auf `int` dynamisch erzeugt, also zur Laufzeit des Programms.

Wie erzeugt man nun dynamisch mehrdimensionale Arrays, zum Beispiel eine zwei- oder dreidimensionale Matrix? Betrachten wir die Anweisung `int *pa = new int[4];`, die eine Zeile aus `int`-Elementen anlegt, so können wir feststellen, dass `new` einen Zeiger vom Datentyp »Zeiger auf Typ eines Arrayelements« zurückgibt. Eine zweidimensionale Matrix ist aber nichts anderes als ein Array von Arrays, wie oben beschrieben. Ein Element dieses Arrays ist eine Zeile mit zum Beispiel 7 Elementen, und daher sollte `new` einen Zeiger vom Datentyp »Zeiger auf eine Zeile mit 7 Elementen« zurückgeben:

```
int (* const p2)[7] = new int [5][7];
```

`p2` ist ein konstanter Zeiger, der auf die Zeile 0 einer zweidimensionalen Matrix mit 5 Zeilen zu je 7 `int`-Zahlen verweist. Der Zugriff auf ein einzelnes Element kann anschließend mit einer Anweisung wie zum Beispiel `p2[1][6] = 66;` erreicht werden. Man beachte, dass die »5« nach `new` auch eine andere Zahl sein kann – die auf beiden Seiten auftretende »7« jedoch nicht, weil sie in den *Datentyp* eingeht. Der Zeiger `p2` wird als

const-Zeiger deklariert und initialisiert, damit eine weitere Zuweisung an p2 durch den Compiler verhindert wird. Eine weitere Zuweisung ohne vorheriges `delete []` hat den Verlust von Speicherplatz zur Folge, wie das Beispiel zeigt:

```
int *pTest = new int[7];
pTest     = new int [10]; // 7 Elemente nicht mehr zugreifbar!
int *const pC = new int[3];
pC[1]      = 123;         // ok, die Elemente von pC sind nicht konstant
pC         = new int [33]; // Fehlermeldung des Compilers:
                        // konstantes Objekt pC kann nicht geändert werden!
```

Die 7 Elemente können nicht mehr per `delete []` freigegeben werden, weil die Information über ihren Ort verloren gegangen ist. Im zweiten Fall ist zu unterscheiden, dass pC zwar konstant ist, nicht aber der Speicherbereich, auf den pC zeigt. Ein konstanter Zeiger auf ein Objekt ist etwas anderes als ein Zeiger auf ein konstantes Objekt. In Analogie zu zweidimensionalen Matrizen wird konsequenterweise eine dreidimensionale Matrix als Array von zweidimensionalen Matrizen formuliert:

```
int (*const p3)[5][7]= new int [44][5][7];
```

p3 ist ein Zeiger, der auf ein Array mit 5 Elementen zeigt, die wiederum aus einem int-Array mit 7 Elementen bestehen.² p3 zeigt auf das erste von 44 5x7-Teil-Arrays. Auch hier geht »[5][7]« in den Datentyp ein und muss daher links und rechts übereinstimmen. Allgemein dürfen anstelle der 5 und der 7 nur konstante Ausdrücke stehen, anstelle der 44 kann ein beliebiger ganzzahliger Ausdruck stehen.

Das Schema setzt sich für vier-, fünf-, ...-dimensionale Arrays fort – aber wer braucht die schon! Auch die Größe eines mit `new` erzeugten Arrays kann *nicht* mit `sizeof()` ermittelt werden! `sizeof(p3)` ergibt nur den Platzbedarf für den Zeiger selbst (zum Beispiel 4 Byte), nicht den mit `new` angelegten Platz.

Mehrdimensionale Arrays mit variabler Feldgröße

Mit variabler Feldgröße ist hier gemeint, dass die Größe des Arrays zur Compilierzeit nicht bekannt ist. Die Größe kann dann natürlich nicht als Teil des Datentyps aufgefasst werden. Hier wird nur kurz auf zweidimensionale Matrizen eingegangen, weil erstens drei- und mehrdimensionale Matrizen leicht daraus ableitbar sind und weil zweitens eine komfortablere Lösung in Abschnitt 9.8 vorgestellt wird. Eine mögliche Lösung besteht darin, zunächst ein Feld von Zeigern auf eindimensionale Arrays (Zeilen) anzulegen und jedem dieser Zeiger eine Zeile mit Spalten zuzuordnen (siehe Abbildung 5.12).

```
int z, s;           // Zeilen , Spalten
cout << "Zeilen und Spalten eingeben:";
cin >> z >> s;      // erst zur Laufzeit bekannt
// Feld von Zeigern auf Zeilen anlegen:
int **const mat = new int* [z]; // mat ist ein konstanter Zeiger auf Zeiger auf int
// jeder Zeile Speicherplatz zuordnen:
for(size_t i = 0; i < z; ++i) {
    mat[i] = new int [s];
}
```

² Eine Anleitung zum Lesen solcher Deklarationen ist auf Seite 226 zu finden.

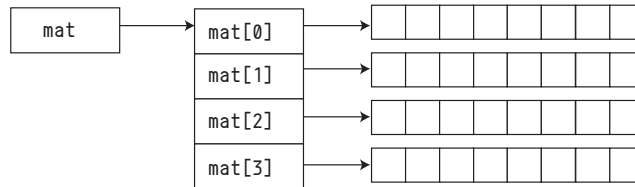


Abbildung 5.12: Zweidimensionales dynamisches Array

`mat` kann nun wie eine gewöhnliche Matrix benutzt werden. Der Zugriff auf ein Element der Matrix `mat` in Zeile `i` und Spalte `j` wird vom Compiler in die entsprechende Zeigerdarstellung umgewandelt, wie schon auf Seite 212 beschrieben.

```
// Beispiel für die Benutzung der dynamisch erzeugten Matrix
for(size_t iz = 0; iz < z; ++iz) {
    for(size_t is = 0; is < s; ++is) {
        mat[iz][is] = iz*s + is;
        cout << mat[iz][is] << '\t';
    }
    cout << endl;
}
```

Der für `mat` mit `new` angelegte Speicherbereich muss nach Gebrauch explizit wieder freigegeben werden, falls er nicht bis zum Programmende bestehen bleiben soll. Wir dürfen jedoch `delete` nicht unmittelbar auf `mat` anwenden, weil dann die einzelnen Zeilen nicht mehr freigegeben werden könnten. Damit ergibt sich folgender Ablauf:

```
// Matrix freigeben
for(size_t zeile = 0; zeile < z; ++zeile) {
    delete [] mat[zeile]; // zuerst Zeilen freigeben
}
delete [] mat; // Feld mit Zeigern auf Zeilenanfänge freigeben
```

5.7.3 Klasse für dynamisches zweidimensionales Array

Dynamische zweidimensionale Arrays sind vielseitig einsetzbar. Aber jedes Mal Speicher zuzuweisen und für ein korrektes `delete` zu sorgen, ist umständlich. Es bietet sich an, diese Vorgänge in einer Klasse zu kapseln mit dem Ziel, dass Objekte dieser Klasse ohne Performance-Verlust benutzt werden können, so wie hier gezeigt:

Listing 5.6: Anwendung für Array2d

```
// cppbuch/k5/array2d/main.cpp
#include "array2d.h" // Klasse Array2d
#include <iostream>
using namespace std;

int main() {
    Array2d<int> arr(5, 7);
    arr.init(0);
    printArray(arr); // Ausgabe mit Hilfsfunktion
    for(size_t z = 0; z < arr.getZeilen(); ++z) {
        for(size_t s = 0; s < arr.getSpalten(); ++s) {
```

```

        arr.at(z, s) = 10*z+s;    // Benutzung, schreibend und ...
        cout << arr.at(z, s) << " "; // lesend (siehe Text unten)
    }
    cout << endl;
}
Array2d<int> arr1(arr); // Kopierkonstruktor
printArray(arr1);
arr1.init(3);          // Neuinitialisierung
arr.assign(arr1);      // Zuweisung
printArray(arr);
Array2d<double> arrd(3, 4, 99.013); // double-Array
printArray(arrd);
Array2d<string> arrs(2, 5, "hello"); // string-Array
printArray(arrs);
}

```

Wie man sieht, dürfen die Array-Elemente nicht nur `int`- oder `double`-Zahlen sein, sondern auch Strings. In der Datei `cppbuch/k5/array2d/array2d.h` übernimmt der Konstruktor die Speicherallokation, der Destruktor sorgt für die Freigabe. Weil das Überladen von Operatoren erst im Kapitel 9 behandelt wird, gibt es hier noch einige Einschränkungen:

- Anstelle der Schreibweise `arr = arr1`; für die Zuweisung wird `arr.assign(arr1)`; geschrieben.
- Der Zugriff auf ein Array-Element über den Indexoperator wird durch einen Funktionsaufruf ersetzt, also etwa `arr.at(z, s) = wert`; statt `arr[z][s] = wert`;

Entwurfsüberlegungen

Da der schreibende Funktionsaufruf auf der *linken* Seite der Zuweisung steht, ergibt sich, dass er eine Referenz auf das zu ändernde Array-Element zurückgeben muss. Bei dem lesenden Funktionsaufruf, der das Array-Objekt `arr` nicht ändert, genügt eine Kopie des Werts oder eine `const`-Referenz.

Falls das Array innerhalb der Klasse so wie in Abbildung 5.12 (Seite 215) erzeugt wird, könnte `at(i, j)` wie folgt realisiert werden, wobei `T` der Typ der Array-Elemente und `ptr` der Zeiger auf den Beginn des internen Arrays ist:

```

T& at(size_t z, size_t s) {
    return ptr[z][s]; // ptr ist hier vom Typ T**.
}

```

Konstruktor und Destruktor wären wegen der Schleifen etwas umständlich. Aus diesem Grund kann die Beschaffung des Speichers als *ein* Block erwogen werden, zum Beispiel `T* ptr = new T[zeilen*spalten]`; `ptr` wäre dann vom Typ `T*` und nicht mehr `T**`. Im Destruktor genügte ein einziges `delete []`. Das bedeutet allerdings, dass ein Zugriff wie `ptr[z][s]` nicht möglich ist. Die Adresse müsste anders berechnet werden – wie, das zeigt die Lösung der Aufgabe 5.2 von Seite 213. Damit wird die Funktion `at()` wie folgt realisiert (`spalten` ist die Anzahl der Spalten):

```

T& at(size_t z, size_t s) {
    return ptr[z * spalten + s]; // ptr ist hier vom Typ T*.
}

```

Der Compiler würde im ersten Fall `ptr[z][s]` in `*(*(ptr+z)+s)` umwandeln, wie auf Seite 212 erläutert. Weil Konstruktor und Destruktor einfacher werden und die Berechnung der Adresse etwa ebenso schnell ist wie die Berechnung von `*(*(ptr+z)+s)`, wird der zweiten Variante, also der Beschaffung des Speichers in einem einzigen Schritt, der Vorzug gegeben. Das folgende Listing zeigt die dazu passende Klasse:

Listing 5.7: Klasse `Array2d`

```
// cppbuch/k5/array2d/array2d.h
#ifndef ARRAY2D_H
#define ARRAY2D_H
#include<cassert>
#include<iostream>

template<typename T>
class Array2d {
public:
    Array2d(size_t z, size_t s)
        : zeilen(z), spalten(s), ptr(new T[z*s]) {
    }

    Array2d(size_t z, size_t s, const T& wert)
        : zeilen(z), spalten(s), ptr(new T[z*s]) {
        init(wert);
    }

    Array2d(const Array2d& a)
        : zeilen(a.zeilen), spalten(a.spalten),
          ptr(new T[zeilen*spalten]) {
        size_t anzahl = zeilen * spalten;
        for(size_t i=0; i < anzahl; ++i) {
            ptr[i] = a.ptr[i];
        }
    }

    ~Array2d() {
        delete [] ptr;
    }

    Array2d& assign(Array2d tmp) { // siehe Text unten
        swap(tmp);
        return *this;
    }

    size_t getZeilen() const { return zeilen; }
    size_t getSpalten() const { return spalten; }

    void init(const T& wert) { // Alle Elemente mit wert initialisieren
        size_t anzahl = zeilen * spalten;
        for(size_t i=0; i < anzahl; ++i) {
            ptr[i] = wert;
        }
    }
}
```

```

const T& at(size_t z, size_t s) const { // für lesenden Zugriff
    assert(z < zeilen && s < spalten);
    return ptr[z * spalten + s];
}

T& at(size_t z, size_t s) { // für verändernden Zugriff
    assert(z < zeilen && s < spalten);
    return ptr[z * spalten + s];
}

void swap(Array2d& rhs) { // Daten von *this mit denen von rhs vertauschen, s.u.
    size_t temp = zeilen;
    zeilen = rhs.zeilen;
    rhs.zeilen = temp;
    temp = spalten;
    spalten = rhs.spalten;
    rhs.spalten = temp;
    T* tempPtr = ptr;
    ptr = rhs.ptr;
    rhs.ptr = tempPtr;
}

private:
    size_t zeilen;
    size_t spalten;
    T* ptr;
    Array2d& operator=(const Array2d& arr); // (noch) verbieten
};

// Globale Funktion zur Ausgabe
template<typename T>
void printArray(const Array2d<T>& a) {
    for(size_t z=0; z < a.getZeilen(); ++z) {
        for(size_t s=0; s < a.getSpalten(); ++s) {
            std::cout << a.at(z,s) << " ";
        }
        std::cout << std::endl;
    }
}

#endif

```

Die private Deklaration des Zuweisungsoperators verhindert, dass `arr = arr1;` geschrieben werden kann. Wie er realisiert werden kann, erfahren Sie im Kapitel 9. In der Funktion `assign(arr)` müsste der Ablauf die folgenden Schritte umfassen: 1. Speicher beschaffen, 2. Werte kopieren, 3. alten Speicher freigeben, 4. Verwaltungsdaten aktualisieren.

Zu den ersten beiden Schritten findet sich Code im Kopierkonstruktor, zum dritten Schritt im Destruktor. Um eine Codeduplikation zu vermeiden, wird in `assign()` eine lokale Kopie durch die Übergabe des Parameters per Wert angelegt. Durch die lokale Kopie statt einer Referenz wird erreicht, dass sie ohne Auswirkung auf das Original verändert werden kann. Die Funktion `swap()` sorgt für die Vertauschung der Objektinhalte (es gibt auch eine `swap()`-Bibliotheksfunktion). Danach hat `*this` den Inhalt von `arr` (= Sinn der Zuweisung) und `arr` den von `*this`. Letzteres wird zurückgegeben, und der Destruktor von

`tmp` sorgt für die Speicherfreigabe. Die Abbildungen 5.13 bis 5.15 zeigen die Schritte für eine hypothetische Anweisung `x.assign(y);`. Die Ellipsen stehen für die `Array2d`-Objekte. Das Objekt `x` kann innerhalb der Funktion mit `*this` angesprochen werden. Nach dieser Anweisung gilt `x == y`.

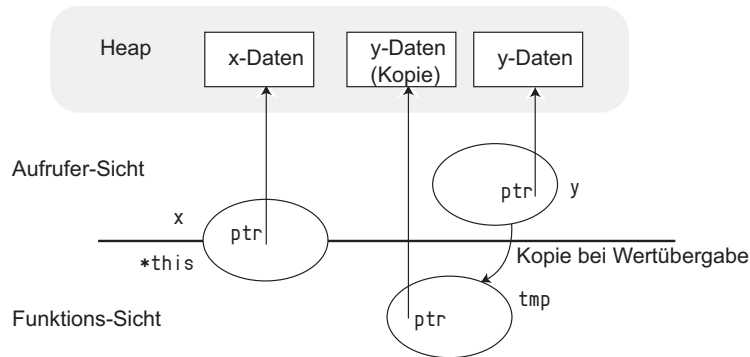


Abbildung 5.13: Zustand direkt nach Funktionsaufruf, aber vor `swap()`

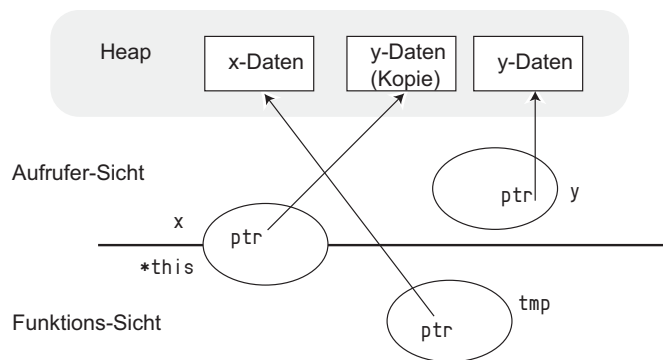


Abbildung 5.14: Zustand nach `swap()`

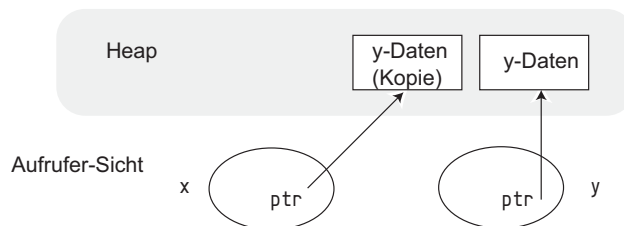


Abbildung 5.15: Zustand nach Funktionsende. Der Destruktor hat `tmp` zerstört.

5.8 Binäre Ein-/Ausgabe

Im Gegensatz zur formatierten Ein-/Ausgabe mit den Operatoren `<<` und `>>` ist der binäre Datentransfer *unformatiert*. Unformatiert heißt, dass die Daten in der *internen* Darstellung direkt geschrieben beziehungsweise gelesen werden, eine Umwandlung zum Beispiel einer `float`-Zahl in eine Folge von ASCII-Ziffernzeichen also unterbleibt. Auf diese Art beschriebene Dateien können nicht sinnvoll mit einem Texteditor bearbeitet oder direkt ausgedruckt werden. Die zum binären Datentransfer geeigneten Funktionen in C++ sind `read()` und `write()`.

Das Prinzip: Es wird ein Zeiger auf den Beginn des Datenbereichs angegeben, also die Adresse des Bereichs. Dabei wird der Zeiger in den Datentyp `char*` umgewandelt. `write()` verlangt an dieser Stelle `char*`, weil ein `char` einem Byte entspricht. Zusätzlich wird die Anzahl der zu transferierenden Bytes angegeben. Zur Wandlung des Zeigers wird der `reinterpret_cast`-Operator verwendet. Dieser Operator verzichtet im Gegensatz zum `static_cast`-Operator von Seite 53 auf jegliche Verträglichkeitsprüfung, weil hier Zeiger auf *beliebige*, das heißt auch selbst geschriebene Datentypen in den Typ `char*` umgewandelt werden sollen.

Das Beispiel zeigt das unformatierte Schreiben und Lesen von `double`-Zahlen als Vorlage, wie man es machen kann. Der Dateiname wurde nur deswegen fest vorgegeben, um die Programme nicht so lang werden zu lassen. Das Schema lässt sich sinngemäß auf beliebige Datentypen und Mengen übertragen. Die `for`-Schleife im ersten Beispielprogramm zeigt eine typische Anwendung des auf den Seiten 76 und 210 erwähnten Kommaoperators. Die zu schreibenden Zahlen werden nur als Beispiel nach der Vorschrift 1.1ⁱ, $i = 0..19$ berechnet.

Listing 5.8: Schreiben einer binären Datei

```
// cppbuch/k5/wdouble.cpp Erzeugen einer Datei double.dat mit 20 double-Zahlen
#include<cstdlib> // für exit( )
#include<fstream>
#include<iostream>
using namespace std;

int main( ) {
    ofstream ziel;
    // ios::binary mit ios::out verwenden (siehe S. 390).
    ziel.open("double.dat", ios::binary|ios::out);
    if(!ziel) {
        cerr << "Datei kann nicht geöffnet werden!\n";
        exit(-1);
    }
    double d = 1.0;
    for(int i = 0; i < 20; ++i, d *= 1.1) // Schreiben von 20 Zahlen
        ziel.write(reinterpret_cast<const char*>(&d), sizeof(d));
} // ziel.close() wird vom Destruktor durchgeführt
```

Listing 5.9: Lesen einer binären Datei

```
// cppbuch/k5/rdouble.cpp
```

```
// Lesen einer Datei double.dat mit double-Zahlen
#include<cstdlib> // für exit( )
#include<fstream>
#include<iostream>
using namespace std;
int main( ) {
    ifstream quelle;
    quelle.open("double.dat", ios::binary|ios::in);
    if(!quelle) { // muss existieren
        cerr << "Datei kann nicht geöffnet werden!\n";
        exit(-1);
    }
    double d;
    while(quelle.read(reinterpret_cast<char*>(&d), sizeof(d)))
        cout << " " << d << '\n';
} // quelle.close() wird vom Destruktor durchgeführt
```

Im zweiten Beispiel wird beim Lesen die Variable `d` verändert. Deswegen fehlt im Vergleich zum ersten Programm das `const` in der Typumwandlung.

ASCII oder binär?

Um den Unterschied zu verdeutlichen, folgt ein Beispiel, in dem eine Matrix einmal als ASCII-Datei und einmal als binäre Datei ausgegeben wird. Die Merkmale der Dateitypen sind im Wesentlichen:

ASCII-Datei

- ASCII-Dateien sind mit einem Texteditor lesbar.
- Schreiben und Lesen von ASCII-Dateien mit einem Programm dauern im Allgemeinen länger als binäres Schreiben oder Lesen von binären Dateien, weil implizit Umformatierungen vom internen Format nach ASCII (oder umgekehrt) vorgenommen werden.
- Anstelle der Funktionen `get()` und `put()` für zeichenweise Verarbeitung können auch die Operatoren `>>` und `<<` zur Ein- und Ausgabe verwendet werden. In der Anwendung besteht kein Unterschied zur Standardein- und -ausgabe.
- Die Dateigröße bestimmt sich aus der Anzahl der ausgegebenen Zeichen und hängt damit von der Formatierung ab. Die Ausgabe sollte so formatiert sein, dass ein problemloses Einlesen möglich ist, zum Beispiel durch Angabe einer genügenden Ausgabeweite oder Einfügen von Leerzeichen zur Trennung.
- Durch die Formatierung kann es zu Wertänderungen kommen, zum Beispiel durch Abschneiden von Nachkommastellen.

Listing 5.10: Dateiausgabe eines Arrays

```
// cppbuch/k5/wmatrix.cpp
// Schreiben einer Matrix als ASCII- und als binäre Datei
// (siehe auch Übungsaufgabe)
#include<cstdlib>
#include<fstream>
#include<iostream>
```

```

using namespace std;

int main() {
    const int ZEILEN = 10, SPALTEN = 8;
    double matrix[ZEILEN][SPALTEN];
    for(int i = 0; i < ZEILEN; ++i) // Matrix mit Werten füllen
        for(int j = 0; j < SPALTEN; ++j)
            matrix[i][j] = i+1 + (j+1)/1000.0;

    // Schreiben als ASCII-Datei (lesbar mit Editor)
    ofstream ziel;
    ziel.open("matrix.asc");
    if(!ziel) {
        cerr << "Datei kann nicht geöffnet werden!\n";
        exit(-1);
    }
    // formatiertes Schreiben
    for(int i = 0; i < ZEILEN; ++i) {
        for(int j = 0; j < SPALTEN; ++j) {
            ziel.width(8);    // siehe auch Seite 378
            ziel << matrix[i][j];
        }
        ziel << endl;
    }
    // Datei schließen, damit ziel wieder verwendet werden kann:
    ziel.close();

    // Schreiben als binäre Datei
    ziel.open("matrix.bin", ios::binary|ios::out);
    if(!ziel) {
        cerr << "Datei kann nicht geöffnet werden!\n";
        exit(-1);
    }
    ziel.write(reinterpret_cast<const char*>(matrix),
               sizeof(matrix));
} // automatisches close()

```

Binärdatei

- Binärdateien sind *nicht* mit einem Texteditor lesbar und auch nicht druckbar.
- Schreiben und Lesen von Binärdateien ist schnell, weil Umformatierungen nicht notwendig sind. Voraussetzung des Vergleichs ist natürlich, dass die Ein- und Ausgabe beider Arten von Dateien auf dieselbe Weise vom Betriebssystem gepuffert wird.
- Die Dateigröße bestimmt sich aus der Größe und Anzahl der ausgegebenen Daten und hat nichts mit ihrer ASCII-Darstellung zu tun, die je nach Formatierung auf verschiedene Art möglich ist. Im Programm oben ist die Dateigröße in Bytes gleich der Matrixgröße `sizeof(double)·(Anzahl der Zeilen)·(Anzahl der Spalten)`.
- Große Dateistrukturen können ohne Benutzung von Schleifen mit einer *einzig* Anweisung ausgegeben werden (siehe `matrix` im Beispiel).

5.9 Zeiger auf Funktionen

Zeiger auf Funktionen ermöglichen es, erst zur Laufzeit zu bestimmen, welche Funktion ausgeführt werden soll (englisch *late binding*, *dynamic binding*) – die Grundlage für den noch zu besprechenden Polymorphismus. Im Beispielprogramm wird entweder das Maximum oder das Minimum der Zahlen 1700 und 1000 ausgegeben. Welche Funktion tatsächlich gewählt wird, entscheidet sich erst *zur Laufzeit*, sodass zur Compilationszeit nicht bekannt sein *kann*, ob mit `(*fp)(a,b)` im Beispielprogramm `max()` oder `min()` aufgerufen wird.

Zur Deklaration: `fp` ist ein Zeiger auf eine Funktion, die einen `int`-Wert zurückgibt und zwei `int`-Parameter verlangt. Dabei ist `(*fp)` statt `*fp` notwendig, um eine Verwechslung mit einer Funktionsdeklaration `int *fp (int, int)` zu vermeiden.

Zeiger auf Funktionen können wie andere Variablen anderen Funktionen als Parameter übergeben werden. Zum Beispiel könnte man einer Funktion, die mathematische Graphen zeichnet, wahlweise einen Zeiger auf die Sinus-Funktion oder auf eine Hyperbel-Funktion übergeben. Das folgende Beispiel zeigt, wie wir eine vordefinierte Funktion mithilfe von Zeigern auf Funktionen nutzen.

Listing 5.11: Funktionszeiger

```
// cppbuch/k5/funkptr.cpp
#include<cstdint>
#include<iostream>
using namespace std;

int max(int x, int y) { return x > y ? x : y;}
int min(int x, int y) { return x < y ? x : y;}

int main() {
    int a = 1700, b = 1000;
    int (*fp)(int, int); // fp ist Zeiger auf eine Funktion
    do {
        char c;
        cout << "max (1) oder min (0) ausgeben (sonst = Ende)?";
        cin >> c;
        // Zuweisung von max() oder min()
        switch(c) {
            case '0': fp = &min; break; // Funktionsadresse zuweisen
            // Ohne den Adressoperator & wandelt der Compiler den
            // Funktionsnamen automatisch in die Adresse um:
            case '1': fp = max; break;
            default : fp = NULL;
        }
    }
    if(fp) { // d.h if(fp != 0)
        // Dereferenzierung des Funktionszeigers und Aufruf
        cout << (*fp)(a, b) << endl;
        // oder direkt Funktionszeiger als Name verwenden:
        // (implizite Typumwandlung des Zeigers in die Funktion)
        cout << fp(a, b) << endl;
    }
}
```

```

    }
  } while(fp);
}

```

Wenn wir ein C-Array sortieren wollen, müssen wir keine eigene Funktion schreiben, sondern können die Funktion `qsort()` verwenden, die in jedem C++-System vorhanden und deren Prototyp im Header `<cstdlib>` deklariert ist. Die hier vorgestellte Bibliotheksfunktion `qsort()` unterscheidet sich von der Funktion `quicksort()` von Seite 134 dadurch, dass die Allgemeinheit der Funktion über ihre Realisierung mit Zeigern auf `void` hergestellt wird und nicht über ein Template. Außerdem sortiert sie C-Arrays und keine `vector`-Objekte.

```

void qsort(void *feldname,
           size_t zahlDerElemente,
           size_t bytesProElement,
           int (*cmp)(const void *a, const void *b));

```



Hinweis

Hier geht es um die Wirkungsweise von Funktionszeigern, nicht um das Sortieren. Wenn es nur um Sortieren geht, wird am besten die Funktion `std::sort()` benutzt, die an anderer Stelle beschrieben wird (Abschnitt 24.4.2).

Der letzte Parameter ist ein Zeiger auf eine Funktion, die einen `int`-Wert zurückgibt und zwei Zeiger auf `const void` als Parameter verlangt, hier `a` und `b` genannt. Der Name `cmp` steht für »compare« und ist hier nur zu Dokumentationszwecken vorhanden; er kann auch weggelassen werden.

`qsort()` soll als Standardfunktion für beliebige Arrays geeignet sein, daher der Datentyp `void*`. Die Funktion mit dem Platzhalternamen `cmp` ist von uns zu schreiben. Sie hat die Aufgabe zu entscheiden, welches von den beiden Arrayelementen, auf die die Zeiger `a` und `b` zeigen, größer ist. Als Ergebnis für eine aufsteigende Sortierung wird verlangt, dass ein Wert kleiner 0 zurückgegeben wird, falls `*a < *b` ist, ein Wert größer 0, falls `*a > *b` ist, und 0 bei `*a == *b`.

Bei absteigender Sortierung sind die Vorzeichen zu vertauschen. Die Funktion zum Vergleich und ihr Aufruf sind unten am Beispiel des Sortierens eines `int`-Feldes beschrieben. Die Schnittstelle muss zu `qsort()` passen, deshalb wird in der Funktion eine Typumwandlung vorgenommen. Dies geschieht dadurch, dass der Zeiger auf `void` in einen Zeiger auf `int` umgewandelt wird, über den durch Dereferenzierung (*) die zu vergleichenden Zahlenwerte `ia` und `ib` gewonnen werden. Ein Umweg der Art

```

int (*vergleich)(const void*, const void*);
vergleich = icmp;
qsort(iefeld, anzahlDerElemente, sizeof(iefeld[0]), vergleich);

```

ist nicht notwendig; der Name der Funktion `icmp` kann direkt eingesetzt werden. Die Bedeutung der Parameterliste im Aufruf ergibt sich durch Vergleich mit dem oben angegebenen Prototypen. Den Quellcode von `qsort()` kennen wir nicht und brauchen ihn auch nicht zu kennen, um `qsort()` benutzen zu können. Es genügt ausschließlich die Kenntnis der Schnittstelle und natürlich die Kenntnis dessen, was `qsort()` tut. Wie in

`qsort()` die Funktion `icmp` angesprochen wird, ist letztlich nicht wichtig zu wissen und wird hier nur zur Erläuterung beschrieben. Innerhalb von `qsort()` steht an irgendeiner Stelle etwa sinngemäß (wobei die Namen im uns unbekannten tatsächlichen Quellcode natürlich ganz anders lauten können):

```
if(icmp(feld+i, feld+j) < 0) {
    ...
}
else {
    // ...usw.
}
```

Durch die Übergabe des Funktionszeigers »weiß« `qsort()`, welche Funktion zu nehmen ist, ganz in Analogie zum oben aufgeführten `min-max`-Beispiel. Dieser Mechanismus wird *callback* genannt und immer dann eingesetzt, wenn ein Server eine Dienstleistung erbringen soll und dazu eine Funktion oder Methode des Client benötigt, die er nicht automatisch zur Verfügung hat. Der Server ist in diesem Fall die Funktion `qsort()`, die die Dienstleistung (= das Sortieren) nur erbringen kann, wenn der Client mitteilt, wie die Objekte zu vergleichen sind. Der Server kann dann die ihm (hier über einen Funktionszeiger) mitgeteilte Funktion aufrufen (= *callback*), ohne ihre Details zu kennen.

Listing 5.12: Quicksort und Funktionszeiger

```
// cppbuch/k5/qsort.cpp
#include<iostream>
#include<cstdlib> // enthält Prototyp von qsort()
using namespace std;

// Definition der Vergleichsfunktion
int icmp(const void *a, const void *b) {
    // Typumwandlung der Zeiger auf void in Zeiger auf int
    // und anschließende Dereferenzierung (von rechts lesen).
    // Die Typumwandlung meint, dass der Speicherinhalt,
    // auf den a und b verweisen, als int zu interpretieren ist.
    int ia = *static_cast<const int*>(a);
    int ib = *static_cast<const int*>(b);
    // Vergleich und Ergebnisrückgabe (> 0, = 0, oder < 0)
    if(ia == ib)
        return 0;
    return ia > ib? 1 : -1;
}

int main() {
    int ifeld[] = {100, 22, 3, 44, 6, 9, 2, 1, 8, 9};
    // Anzahl der Elemente = sizeof(Feld) / sizeof(ein Element)
    size_t anzahlElemente = sizeof(ifeld)/sizeof(ifeld[0]);
    // Aufruf von qsort() mit Funktionszeiger icmp:
    qsort(ifeld, anzahlElemente, sizeof(ifeld[0]), icmp);
    for(size_t i = 0; i < anzahlElemente; ++i) // Ausgabe des sortierten Feldes
        cout << ' ' << ifeld[i];
    cout << endl;
}
```

Deklarationen mit Zeigern auf Funktionen sind manchmal etwas mühsam zu lesen. Hinweise zum Lesen komplexer Deklarationen werden auf Seite 226 gegeben.



Übungen

5.5 Schreiben Sie eine Funktion `strcpy(char* ziel, const char* quelle)`, die den Rückgabetypp `void` hat und die den Inhalt des Strings `quelle` in den String `ziel` kopiert, wobei der vorherige Inhalt von `ziel` dabei überschrieben wird. Es sei vorausgesetzt, dass `ziel` ausreichend groß ist, um `quelle` aufzunehmen.

5.6 Schreiben Sie eine Funktion `char* strduplikat(const char* s)`, die den String `s` dupliziert, indem neuer Speicherplatz mit `new` beschafft und `s` in diesen Bereich hineinkopiert wird. Ein Zeiger auf den Beginn des Duplikats soll zurückgegeben werden.

5.7 Auf Seite 224 wurde gezeigt, wie Quicksort auf ein Ganzzahlen-Feld angewendet werden kann. Gegeben sei nun ein alphabetisch zu sortierendes String-Array.

```
const char *sfeld[]={ "eins", "zwei", "drei", "vier", "fünf",
                      "sechs", "sieben", "acht", "neun", "zehn"};
```

Schreiben Sie ein Programm, das dieses Array mit `qsort()` sortiert. Wie sieht die benötigte Vergleichsfunktion aus, die hier `scmp()` genannt sei? Wie ist `qsort()` aufzurufen?

Hinweise: a) Diese Aufgabe ist nur für diejenigen, die ihr Wissen über den Einsatz von Funktionszeigern bei einer C-Bibliotheksfunktion vertiefen möchten – C ist schließlich eine Untermenge von C++. Die anderen benutzen besser `std::sort()`. Ein Beispiel für die Alternative finden Sie nach der Lösung zu dieser Aufgabe (siehe Seite 914). b) Innerhalb von `scmp()` kann `strcmp()` aus `<cstring>` benutzt werden. `strcmp(a,b)` gibt eine Zahl kleiner 0 zurück, falls der C-String `a` < der C-String `b` ist, größer 0, falls `a` > `b` ist, und 0 bei Gleichheit.

5.10 Komplexe Deklarationen lesen

Komplexe Deklarationen sind manchmal schwer zu lesen. Es gibt dafür ein Rezept, das hier etwas vereinfacht dargestellt wird: Man löst die Deklaration vom Namen her auf und schaut dann nach rechts und links in Abhängigkeit vom Vorrang der Syntaxelemente. Beginnend bei den wichtigen Elementen ist die Reihenfolge (nach [vdL]):

- runde Klammern `()`, die Teile der Deklaration zusammenfassen (gruppieren);
- runde Klammern `()`, die eine Funktion anzeigen;
- eckige Klammern `[]`, die ein Array anzeigen;
- Sternsymbol `*`, das »Zeiger auf« bedeutet.

Beispiel:

```
char *(*(*seltsam)(double, int))[3]
```

Name: `seltsam`

Klammer: Die Klammer dient zur Gruppierung, das nächste Zeichen ist daher `*`.

`*`: `seltsam` ist ein Zeiger. Nun haben wir die Auswahl zwischen einer Funktion (Klammer rechts) oder einem Zeiger (Sternchen links). Die Klammer hat Vorrang.

`(double, int)`: `seltsam` ist ein Zeiger auf eine Funktion, die über ein `double`- und ein `int`-Argument verfügt.

Klammer: Die Klammer dient zur Gruppierung, das nächste Zeichen ist `*`. Die Funktion gibt also einen Zeiger zurück.

eckige Klammer: Vorrang gegenüber `*`. Der Zeiger zeigt auf ein Array mit 3 Elementen, `*`, links: die `char`-Zeiger sind.

Zusammengefasst: `seltsam` ist ein Zeiger auf eine Funktion mit einem `double`- und einem `int`-Argument, die einen Zeiger auf ein Array von Zeigern auf `char` zurückgibt. Ein weiteres Beispiel: `int** was[2][3];`:

`was` ist ein Array von Arrays (= zweidimensionales Array), dessen Elemente Zeiger auf Zeiger auf `int` sind. Eine gut lesbare und ausführliche Behandlung des Themas ist in [\[vdL\]](#) zu finden.

typedef

Komplizierte Deklarationen sollten der schlechten Lesbarkeit wegen vermieden werden. Manchmal sind sie jedoch unumgänglich. Für diesen Fall bietet das Schlüsselwort `typedef` die Möglichkeit, die Lesbarkeit durch Strukturierung der Namensgebung zu verbessern. Betrachten wir zunächst ein einfaches Beispiel: In einem Programm wollen wir die Unterschiede in der Genauigkeit der Ergebnisse je nach Datentyp `float` oder `double` untersuchen. Mit `typedef` können wir uns einen anderen Namen für den Datentyp schaffen:

```
typedef float real;
int main() {
    real Zahl = 1.7353; // Zahl ist vom Typ float
    // ...
}
```

Wenn wir dasselbe Programm mit `double`-Zahlen rechnen lassen wollen, brauchen wir nur die `typedef`-Zeile durch

```
typedef double real;
```

zu ersetzen und neu zu compilieren. Alle Vorkommen von `real` bleiben unverändert, haben aber nun die Bedeutung von `double`.

Komplexe Deklarationen lassen sich durch `typedef` lesbarer gestalten, weil neue Typnamen zur Strukturierung benutzt werden können. Das obige `seltsame` Beispiel sei wieder aufgegriffen, indem ein neuer Datentyp `ArrayVon3CharZeigern` definiert wird:

```
typedef char* ArrayVon3CharZeigern [3];
// die Deklaration
ArrayVon3CharZeigern A; // ist identisch mit:
char* A[3];
```

Der Rückgabetyt der Funktion, die zum Funktionszeiger `seltsam` der Vorseite passt, ist ein Zeiger auf den neuen Datentyp:

```
typedef ArrayVon3CharZeigern *ZeigerAufArrayVon3CharZeigern;
```


Äquivalent dazu ist:

```
typedef char* (*ZeigerAufArrayVon3CharZeigern) [3];
```

Der Datentyp des Zeigers `seltsam` kann mit `typedef` beschrieben werden:

```
typedef char* (*seltsamerTyp) (double, int) [3];
```

oder erheblich lesbarer mit

```
typedef ZeigerAufArrayVon3CharZeigern (*seltsamerTyp) (double, int);
```

Damit ließe sich eine »seltsame Funktion« deklarieren, die über die Zeiger ausgeführt werden kann:

```
// Prototyp (Implementation ist sonstwo...)
ZeigerAufArrayVon3CharZeigern seltsameFunktion(double, int);
ZeigerAufArrayVon3CharZeigern z;
seltsam = seltsameFunktion;
z = seltsam(3.1, 2); // Ausführung über Zeiger
seltsamerTyp X;
X = seltsam;
z = X(3.1, 2);      // Ausführung über Zeiger
```

In [vdL] ist ein C-Programm abgedruckt, das in der Unix-Welt unter dem Namen *cdecl* bekannt ist. Es übersetzt komplizierte Deklarationen in verständlichen Text. Das Programm ist auch in den Beispielen der DVD vorhanden, siehe *cppbuch/k5/cdecl.cpp*. Das Programm wartet nach dem Start auf die Eingabe einer Deklaration, zum Beispiel `char* A[3]`, und gibt danach die Auswertung aus (»A is array 0..2 of pointer to char«).



Übungen

5.8 Schreiben Sie eine Funktion `void leerzeichenEntfernen(char* s)`, die alle Leerzeichen im `char`-Array `s` entfernt. Zum Beispiel soll aus »a bb ccc d« die Zeichenkette »abbcccd« werden. Verwenden Sie dabei Zeiger.

5.9 Schreiben Sie ein Programm, das Dateien, deren Namen in der Kommandozeile angegeben werden, auf der Standardausgabe ausgibt. Zum Beispiel könnte der Befehl

```
prog datei1.cpp XXX datei2.cpp
```

dazu führen, dass die Dateien *datei1.cpp* und *datei2.cpp* auf dem Bildschirm angezeigt werden. Wenn eine Datei `XXXX` nicht vorhanden ist, soll es eine Fehlermeldung »Datei `XXXX` nicht gefunden!« geben.

5.10 Schreiben Sie ein Programm, das alle in einer Datei vorkommenden Namen ausgibt. Ein Name ist dabei so definiert: Er beginnt mit einem Buchstaben, anschließend folgen beliebig viele Buchstaben und Ziffern. Der Unterstrich zählt auch als Buchstabe.

5.11 Standard-Typumwandlungen für Zeiger

- Ein integraler Ausdruck, der 0 ergibt, kann zu einem Zeiger auf ein Objekt eines beliebigen Typs `T` konvertiert werden: `T* ptr = 0;` (Null-Zeiger).
- Jeder Zeiger ist nach `bool` konvertierbar. Dabei wird ein Null-Zeiger stets zu `false` und ein anderer Zeiger stets zu `true` ausgewertet.
- Jeder Zeiger auf ein Objekt eines beliebigen Typs `T` kann in einen Zeiger auf `void` konvertiert werden:

```
T* ptr;
void* vptr = ptr;
```

Anmerkung: Die in Abschnitt 5.12 beschriebenen Zeiger auf Elementfunktionen und -daten sind keine Zeiger auf Objekte und können nicht nach `void*` gewandelt werden.

- Jeder Zeiger auf ein Objekt einer Klasse `B` kann in einen Zeiger auf Klasse `A` umgewandelt werden, wenn `A` eine Oberklasse von `B` ist und (bei Mehrfachvererbung) eindeutig ermittelt werden kann (die Begriffe Oberklasse und Vererbung werden in Kapitel 7 erläutert).
- Arrays können zu Zeigern konvertiert werden. Beispiel:

```
char dasArray[]="ABC";
const char* cpc = dasArray; // Array → Zeiger
char* cp = dasArray;       // Array → Zeiger
```

Funktionszeiger

Eine Funktion kann in einen Zeiger auf eine Funktion oder umgekehrt umgewandelt werden. Ein Beispiel verdeutlicht die beiden Fälle:

```
void drucke(int x) { cout << x << endl; } // Funktion

int main() {
    void (*f1)(int) = &drucke; // keine Umwandlung
    void (*f2)(int) = drucke;  // Funktion → Zeiger
    f1(3);                    // Zeiger → Funktion
    (*f1)(3);                 // explizit: Zeiger→ Funktion
}
```

Hinweis: Bei Zeigern auf Elementfunktionen ist keine implizite Typumwandlung vorgesehen. In der Zeile

```
int (0rt::*fp)() const = &0rt::X;
```

des Beispielprogramms im nächsten Abschnitts darf also der Adressoperator `&` nicht weggelassen werden.

5.12 Zeiger auf Elementfunktionen und -daten³

C++ erlaubt es, Zeiger auf Elementdaten (englisch *data members*) und -funktionen (englisch *member functions*) zu richten. Deklaration und Zugriff auf Elemente werden mit den Operatoren `::*`, `.*` und `->*` bewerkstelligt. Diese Zeiger unterliegen einer Typprüfung durch den Compiler, die die Klassenzugehörigkeit berücksichtigt, im Unterschied zu den Zeigern auf globale Funktionen des Abschnitts 5.9.

Eine sehr große Flexibilität wird erreicht, weil Zeiger zur Laufzeit auf verschiedene Elementfunktionen (bzw. seltener Attribute) gerichtet werden können. Diese Zeiger können wiederum mit Zeigern kombiniert werden, die zur Laufzeit auf verschiedene Objekte verweisen. Die verschiedenen Objekte können sogar unterschiedlichen Typs sein, wie noch gezeigt wird (Abschnitt 7.6).

5.12.1 Zeiger auf Elementfunktionen

Die von Seite 157 bekannte Klasse `Ort` hat einige Elementfunktionen, auf die im folgenden Beispielprogramm über Zeiger zugegriffen wird. Bei der Initialisierung von Zeigern auf Elementfunktionen und -daten darf im Gegensatz zu Zeigern auf andere Funktionen (siehe Beispiel auf Seite 223) der Adressoperator `&` nicht weggelassen werden.

Listing 5.13: Zeiger auf Elementfunktionen

```
// cppbuch/k5/elfunptr.cpp
#include "ort.h" // schließt <iostream> ein
using namespace std;

int main() {
    Ort einOrt(100, 200);
    // Zeiger auf Ort::getX() richten.
    int (Ort::*fp)() const = &Ort::getX;
    cout << (einOrt.*fp)() << endl; // dasselbe wie einOrt.getX()
    fp = &Ort::getY; // Funktionszeiger umschalten
    cout << (einOrt.*fp)() << endl; // jetzt dasselbe wie einOrt.getY()
    // Das Umschalten des Zeigers auf einen neuen Ort beeinflusst nicht die
    // Zuordnung zu der Methode:
    Ort * derZeiger = new Ort(300, 400);
    cout << (derZeiger->*fp)() << endl;
    // ist dasselbe wie derZeiger->getY()

    void (Ort::*elFktPtr)(int, int) = &Ort::aendern;
    (derZeiger->*elFktPtr)(500, 600);
    // ist dasselbe wie derZeiger->aendern(500, 600)

    anzeigen(*derZeiger); // geänderter Ort
    delete derZeiger;
}
```

³ Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

5.12.2 Zeiger auf Elementdaten

Sicher kommt es kaum vor, dass auf Elementdaten über Zeiger zugegriffen werden soll, weil Attribute in der Regel privat sind. Das folgende Beispielprogramm zeigt, wie der Zugriff bei öffentlichen Attributen möglich ist. Man sieht dabei, dass das Umschalten auf ein neues Objekt nicht die Zuordnung zum Attribut `b` zerstört.

Listing 5.14: Zeiger auf Elementdaten

```
// cppbuch/k5/eldatptr.cpp
#include<iostream>
using namespace std;

struct Datensatz {                // öffentliche Klasse
    Datensatz(int x, int y)       // Konstruktor
    : a(x), b(y) {}
    int a;
    int b;
};

int main() {
    // Zeiger auf Elementattribute
    Datensatz einDatensatz(1, 2);
    int Datensatz::*dp = &Datensatz::a;

    cout << einDatensatz.*dp << endl; // 1
    dp = &Datensatz::b;               // Zeiger umschalten
    cout << einDatensatz.*dp << endl; // 2

    // neues Objekt erzeugen
    Datensatz* dPtr = new Datensatz(1000, 2000);
    cout << dPtr->*dp << endl;        // 2000
    delete dPtr;
}
```


6

Objektorientierung 2

Dieses Kapitel behandelt die folgenden Themen:

- Wie funktioniert eine String-Klasse?
- Unterschied zwischen objekt- und klassenbezogenen Daten und Funktionen
- Klassen-Templates
- Template-Metaprogrammierung
- Templates mit variabler Anzahl von Parametern

6.1 Eine String-Klasse

Die Verarbeitung der Zeichenketten oder C-Strings in C++, wie sie in Abschnitt 5.3 beschrieben sind, ist ziemlich mühselig und fehleranfällig, weil stets der Speicherplatz genau beachtet werden muss. Die Benutzung der in `<cstring>` deklarierten Standardfunktionen wie `strcpy(ziel, quelle)` aus der C-Bibliothek setzt voraus, dass für `ziel` genügend Speicher vorhanden ist. Auch die Standardeingabe ist gefährlich (Seite 194). Deshalb bietet die C++-Standardbibliothek eine String-Klasse an, die die Verarbeitung von Strings stark vereinfacht und sicherer macht, indem die Speicherverwaltung und die Prüfungen auf das terminierende Nullbyte `'\0'` in der Klasse gekapselt werden. Die Benutzung der Standardklasse `string` ist von Seite 86 bekannt.

In diesem Abschnitt geht es darum zu zeigen, wie eine String-Klasse aufgebaut sein kann. Es wird die Klasse `MeinString` vorgestellt, die nur einen kleinen Teil der im Standard

vorgesehenen Funktionen liefert und vergleichsweise einfach aufgebaut ist. Die Klasse `MeinString` zeigt insbesondere die Notwendigkeit von Kopierkonstruktor, Destruktor und Zuweisungsoperator. Die Namen der Methoden entsprechen den Namen der Klasse `string` des C++-Standards, die Sie später ohnehin benutzen werden. Um eine Verwechslung zu vermeiden, wird hier für die Klasse selbst ein anderer Name gewählt. Um kompatibel zum bekannten Stringverhalten zu bleiben, wird in der Klasse vorausgesetzt, dass jede Zeichenkette mit einem Nullbyte abgeschlossen wird. Das Überladen von Operatoren wird erst in Kapitel 9 behandelt. Aus diesem Grund wird der Zuweisungsoperator »verboten«, indem er als `private` deklariert wird.

Listing 6.1: Klasse `MeinString`, erste Version

```
// /cppbuch/k6/meinstring/meinstring.h
// einfache String-Klasse. Erste Version
#ifndef MEINSTRING_H
#define MEINSTRING_H
#include <cstddef> // size_t
#include <iostream>

class MeinString {
public:
    MeinString(const char* str = ""); // allg. Konstruktor
    MeinString(const MeinString&);    // Kopierkonstruktor
    ~MeinString();                   // Destruktor
    MeinString& assign(MeinString);   // Zuweisung
    MeinString& assign(const char*);  // Zuweisung eines char*
    // Zur Begründung des Rückgabetyps MeinString& statt void siehe Punkt 6
    // der Faustregeln zur Methodenkonstruktion auf Seite 559.
    const char& at(size_t position) const; // Zeichen holen
    char& at(size_t position);             // Zeichen holen
    // Wegen des Rückgabetyps Referenz kann das Zeichen geändert werden.
    size_t length() const { return len; } // Anzahl der Zeichen
    const char* c_str() const { return start; } // C-String zurückgeben
private:
    size_t len; // Länge
    char* start; // Zeiger auf den Anfang
    void operator=(const MeinString&); // noch nicht behandelten Operator verbieten
};

void anzeigen(std::ostream&, const MeinString&); // siehe Text
#endif
```

Die Klasse `MeinString` hat nur zwei private Daten: `start` ist ein Zeiger auf den Beginn der Zeichenkette. Die Zeichenkette kann verschiedene Längen haben, daher wird dem Zeiger mit `new` der jeweils benötigte Platz zugewiesen. Daraus folgt, dass der Destruktor die Aufgabe hat, diesen Speicherplatz wieder freizugeben. `start` ist mit der Methode `c_str()` öffentlich lesbar. Das ist notwendig, um ein `MeinString`-Objekt als C-String an eine Funktion übergeben zu können. Ein Beispiel ist die `fstream`-Funktion `open()`, die einen Dateinamen des Typs `const char*` verlangt. `const` im Rückgabetypp verhindert, dass der Aufrufer der Funktion verändernden Zugriff erhält.

`len` enthält die aktuelle Länge des Strings. Die Abfrage mit `length()` ist schneller als der bekannte Aufruf von `strlen()` aus *string.h*, nicht durch die Realisierung als inline-Funktion, sondern weil der String zur Längenermittlung nicht jedesmal erneut durchlaufen wird. Abbildung 6.1 zeigt ein `MeinString`-Objekt namens `einMstring` als UML-Diagramm. Das Minuszeichen zeigt an, dass die Attribute privat sind. Weil es zunächst nur um die internen Daten geht, wurden die Methoden in der Abbildung weggelassen.

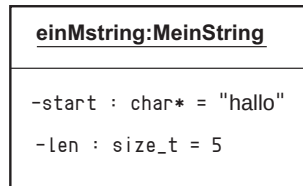


Abbildung 6.1: Ein Objekt der Klasse `MeinString` namens `einMstring`

Implementierung der Klasse `MeinString`

Eine mögliche Implementation zeigt die Datei *cppbuch/k6/meinstring/meinstring.cpp*. Innerhalb der Klasse `MeinString` werden die vorgefertigten Funktionen `strcpy()` und `strlen()` des Headers `<cstring>` genutzt.

Der allgemeine Konstruktor erzeugt aus einem klassischen C-String ein Objekt der Klasse `MeinString`, damit Anweisungen wie

```
MeinString meinStringObjekt("beliebige Zeichenkette");
```

möglich sind. Dazu wird zunächst die Länge des C-Strings ermittelt und ausreichend Platz bereitgestellt, auch für das Nullbyte. Anschließend werden alle Zeichen einschließlich `'\0'` kopiert:

```
MeinString::MeinString(const char *s) // allg. Konstruktor
: len(strlen(s)), start(new char[len+1]) {
    strcpy(start, s);
}
```

Im zweiten Teil der Initialisierungsliste wird auf das Attribut `len` Bezug genommen, das im ersten Teil initialisiert wird. Entsprechend Punkt 1 auf Seite 156 spielt aber nur die Reihenfolge der Attribute im `private`-Teil der Klasse eine Rolle, nicht die Reihenfolge in der Initialisierungsliste! Wenn die Reihenfolge umgekehrt wäre, nämlich

```
// falsch, falls sich die Initialisierung von start auf len bezieht
char *start; // Zeiger auf den Anfang
size_t len; // Länge
```

würde der Konstruktor fehlerhaft arbeiten: Zuerst würde `start` initialisiert, wobei eine undefinierte Menge an Speicher zugewiesen würde (`len` ist noch unbekannt!). Danach erst würde `len` initialisiert, falls nicht das Programm mit einer Fehlermeldung schon abgebrochen wurde.

Durch den Vorgabewert "" in der Deklaration auf Seite 234 wird kein Standardkonstruktor benötigt. Wenn ein leerer String etwa mit `MeinString leeresString;` erzeugt wird, ist die Wirkung genau dieselbe, die der folgende Konstruktors erzielen würde:

```
MeinString::MeinString() // (hier fiktiver) Standardkonstruktor
: len(0), start(new char[1]) { // Platz für '\0'
    *start = '\0';           // leerer String
}
```

Der Kopierkonstruktor kann die Länge des Objekts, mit dem initialisiert wird, direkt übernehmen:

```
MeinString::MeinString(const MeinString& m) // Kopierkonstruktor
: len(m.len), start(new char[len+1]) {
    strcpy(start, m.start);
}
```

Hier ist deutlich zu sehen, dass ein eigener Kopierkonstruktor für die Klasse notwendig ist. Der bei Abwesenheit dieses Konstruktors durch das System erzeugte Kopierkonstruktor würde nur die Länge und den Zeiger kopieren, nicht aber ein echtes Duplikat erzeugen! In diesen und ähnlichen Fällen muss stets ein besonderer Kopierkonstruktor gebildet werden, zum Beispiel, wenn ein Objekt wie hier Zeiger enthält, weil vom Kopierkonstruktor des Systems zwar die Zeiger kopiert werden (»flache« Kopie (englisch *shallow copy*)), nicht aber die Datenbereiche (wie Arrays, Strings oder andere Objekte), auf die die Zeiger verweisen (»tiefe« Kopie (englisch *deep copy*)). Der Unterschied wird in Abbildung 6.2 sichtbar. Der obige Kopierkonstruktor erzeugt also eine »tiefe Kopie«. Vergleichen Sie dazu den Abschnitt »Kopieren von Strings« auf Seite 197. Dieselbe Problematik tritt natürlich auch bei Zuweisungen auf (siehe `assign()`-Methode unten).

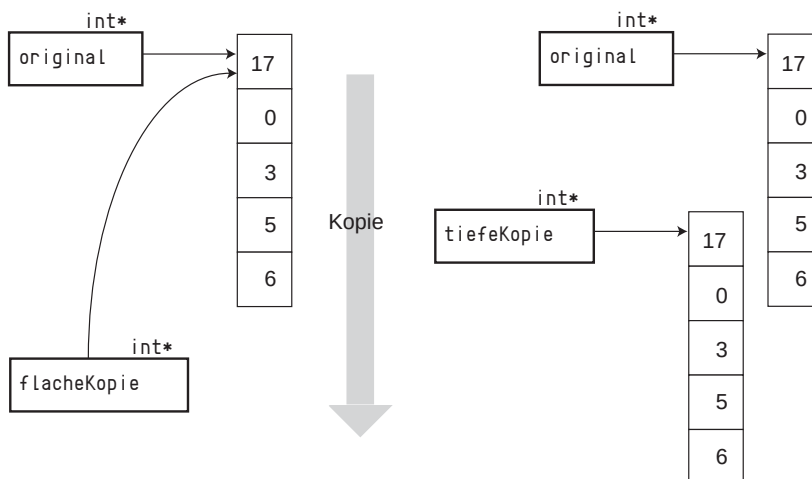


Abbildung 6.2: »Flache« und »tiefe« Kopie eines Objekts

Es gibt verschiedene Möglichkeiten für die Bedeutung einer Kopie: Es ist ein Unterschied, ob nach einer Zuweisung `a = b;` das Objekt `a` einen Zeiger enthält, der auf *den-*

selben Speicherbereich wie der entsprechende Zeiger des Objekts `b` zeigt (Referenzsemantik, Abbildung 6.2 links), oder ob der Zeiger von `a` auf ein neu erzeugtes *Duplikat* des Speicherbereichs verweist (Wertsemantik, siehe Abbildung 6.2 rechts). Der systemerzeugte Destruktor würde nur `start` und `len` vom Stack entfernen, was keinesfalls ausreichend ist. Der folgende Destruktor gibt den durch `new[]` beschafften Platz frei:

```
MeinString::~MeinString() {           // Destruktor
    delete [] start;
}
```

Die überladenen `assign()`-Methoden erlauben Zuweisungen von `MeinString`-Objekten oder C-Strings, zum Beispiel

```
MeinString nochEinString("hallo"); // allg. Konstruktor
einString.assign(nochEinString);   // Zuweisung
einString.assign("neuer Text");    // Zuweisung
einString.assign(einString);       // Zuweisung auf sich selbst?
einString = nochEinString; // (noch) nicht erlaubt, siehe Text:
```

Die Zuweisung mit dem Zuweisungsoperator `=` ist hier nicht gestattet, weil der systemgenerierte Zuweisungsoperator kein echtes Duplikat erzeugt. Wie Sie eigene Zuweisungsoperatoren schreiben können, wird unten in Abschnitt 9.2.2 erläutert. Die Methoden zur Zuweisung müssen in ausreichender Menge neuen Speicherplatz beschaffen, anschließend die Daten kopieren und dann den vorher benutzten Speicherplatz freigeben.

Bei der Zuweisung könnte man daran denken, dass eine Zuweisung auf sich selbst zwar nicht sinnvoll, aber syntaktisch möglich ist. Eine Kopie durchzuführen wäre vertane Zeit. Dieser Fall ist aber so selten, dass auf eine Optimierung hier verzichtet wird. Die Funktion `swap()` vertauscht zwei String-Objekte. Sie wird für die Zuweisung benötigt und benutzt die Bibliotheksfunktion `std::swap()`.

```
void MeinString::swap(MeinString& mString) {
    std::swap(len, mString.len);           // Bibliotheksfunktion
    std::swap(start, mString.start);
}
```

Um zu verhindern, dass ein `MeinString`-Objekt in einen nichtkonsistenten Zustand gerät, falls etwas mit der Speicherbeschaffung schief gehen sollte, wird der alte Speicherplatz erst nach vollendeter Kopie freigegeben. Das geht am einfachsten, indem eine Kopie erzeugt wird, denn nur dabei wird Speicher beschafft. Dann wird das aktuelle Objekt mit der Kopie vertauscht. Schließlich wird der alte Speicherplatz durch den Destruktor der Kopie freigegeben, entsprechend dem Verfahren des Abschnitts 5.7.3.

```
MeinString& MeinString::assign(MeinString m) { // Zuweisung eines MeinString
    swap(m);                                   // Platz mit Kopie m tauschen
    return *this;
} // Der Destruktor der Kopie m wird hier aktiv.
```

Die Zuweisung von C-Strings kann von der `assign(MeinString)`-Funktion profitieren. Die Methode wird daher noch kürzer:

```
MeinString& MeinString::assign(const char *s) { // Zuweisung eines char*
    return assign(MeinString(s));
}
```

Es wird ein temporäres `MeinString`-Objekt erzeugt, das der ersten `assign()`-Funktion übergeben wird. Um auf einzelne Zeichen des Strings lesend zuzugreifen, gibt es die Methode `at()`, der die Position eines Zeichens innerhalb der Zeichenkette übergeben wird. `at()` ist für konstante `MeinString`-Objekte überladen. Der String wird durch das Lesen nicht verändert (`const`):

```
char& MeinString::at(size_t position) { // Zeichen per Referenz holen
    assert(position < len); // Nullbyte lesen ist nicht erlaubt
    return start[position];
}

const char& MeinString::at(size_t position) const { // Zeichen holen
    assert(position < len); // Nullbyte lesen ist nicht erlaubt
    return start[position];
}
```

Die Rückgabe per nicht-konstanter Referenz erlaubt die Änderung eines Zeichens im String. Erinnern wir uns: Erstens ist eine Referenz nichts anderes als ein anderer Name für etwas, in diesem Fall ein Zeichen innerhalb des Strings, und zweitens kann man sich das Ergebnis eines Funktionsaufrufs anstelle des Aufrufs eingesetzt denken.

```
einString.at(0) = 'X';
```

heißt also, dass die gesamte linke Seite der Zuweisung nichts anderes darstellt als das Zeichen mit der Nummer 0 im String! Die Wirkung dieser zunächst ungewohnten Schreibweise ist dieselbe, als wenn wir `einString.start[0] = 'X';` geschrieben hätten, wobei wir hier ignorieren, dass `start` privat ist. Entsprechend der C-Konvention beginnt die Zählung bei 0. Mit `at()` kann bei der Abfrage in Bedingungen *nicht* mehr auf das (interne) abschließende Nullbyte eines Strings zugegriffen werden. Ersatzweise bietet sich die Abfrage auf die Länge des Strings an:

```
// at()-Anwendung zur Anzeige eines MeinString-Objekts
int i = -1;
while(++i < einString.length()) {
    cout << einString.at(i);
}
```

Eine direkte Ausgabe eines `MeinString`-Objekts, etwa der Art `cout << einString,` ist nicht möglich, weil der Ausgabeoperator `<<` für `MeinString`-Objekte noch nicht definiert ist. Das kann geändert werden, wie die Übung 9.6 auf Seite 331 zeigt. Hier kann die Ausgabe auf zwei Arten realisiert werden:

```
cout << einString.c_str();
anzeigen(cout, einString);
```

Letzteres ist der Aufruf der oben deklarierte Funktion `anzeigen()`, die wie folgt definiert ist:

```
void anzeigen(std::ostream& os, const MeinString& m) {
    os << m.c_str();
}
```

6.1.1 Optimierung der Klasse MeinString

Bei verschiedenen Operationen oben wird neuer Speicher beschafft und der alte freigegeben. Dies ist nicht notwendig, wenn der vorhandene Speicher genug Platz hat. Dann kann direkt in den alten Speicher kopiert werden. Weil in diesem Fall die Speicherbeschaffung mit `new` entfällt, ist keine Exception zu erwarten, sodass dieser Weg auch exception-sicher ist. Um dies zu implementieren, wird am besten ein Attribut `cap` (für Capacity) eingeführt, das den gesamten Speicherplatz (ohne das Nullbyte) für das `MeinString`-Objekt angibt. Die Methode `capacity()` gibt den insgesamt den für Zeichen zur Verfügung stehenden Platz zurück. Das Attribut `len`, das die Anzahl der tatsächlich gespeicherten Zeichen angibt, ist immer kleiner oder gleich `cap`.

Eine Methode `reserve(size_t groesse)` kann den entsprechenden Speicherplatz reservieren (bzw. nichts tun, wenn der aktuelle Platz schon gleich oder größer als `groesse` ist). Da `reserve()` dafür sorgen muss, dass der Inhalt des Objekts erhalten bleibt, ist ggf. ein Umkopieren notwendig. Falls Speicher ohne Umkopieren reserviert werden soll, sinnvoll zum Beispiel bei der Methode `assign()`, kann dies von einer privaten Hilfsfunktion `reserve_only()` erledigt werden. Falls keine Operationen mehr mit einem `MeinString`-Objekt zu erwarten sind, reduziert die Methode `void shrink_to_fit()` den Speicherplatz auf das Notwendige. In der Header-Datei sind die folgenden Änderungen vorzunehmen:

```
// Auszug aus cppbuch/k6/meinstringOpt/meinstring.h
public:
    size_t capacity() const { return cap; } // Kapazität zurückgeben
    void reserve(size_t bytes);           // Platz reservieren mit Erhalt des Inhalts
    void shrink_to_fit();                 // Platz minimieren
private:
    size_t cap;                          // Kapazität (direkt nach len eintragen)
    void reserve_only(size_t bytes);     // nur Platz reservieren
```

In der Implementierungsdatei sind nicht nur Methoden betroffen, sondern auch Konstruktoren. Nur die Änderungen sind im Folgenden angegeben:

```
MeinString::MeinString(const char *s) // allg. Konstruktor
    : len(strlen(s)), cap(len), start(new char[cap+1]) {
    strcpy(start, s);
}

MeinString::MeinString(const MeinString& m) // Kopierkonstruktor
    : len(m.len), cap(len), start(new char[cap+1]) {
    strcpy(start, m.start);
}

void MeinString::reserve(size_t groesse) {
    if(groesse > cap) { // nichts tun, wenn der Platz reicht
        char *temp = new char[groesse+1]; // neuen Platz beschaffen, Inhalt erhalten
        strcpy(temp, start);              // umkopieren
        delete [] start;                  // alten Platz freigeben
        start = temp;                     // Verwaltungsinformation aktualisieren
        cap = groesse;                    // Verwaltungsinformation aktualisieren
    }
}
```

```

void MeinString::reserve_only(size_t groesse) {
    if(groesse > cap) {                // nichts tun, wenn der Platz reicht
        char *temp = new char[groesse+1]; // nur neuen Platz beschaffen
        delete [] start;                // alten Platz freigeben
        start = temp;                   // Verwaltungsinformation aktualisieren
        cap = groesse;                  // Verwaltungsinformation aktualisieren
    }
}

void MeinString::shrink_to_fit() {
    if(cap > len) {                    // nichts tun, wenn kein Speicher eingespart wird
        char *temp = new char[len+1]; // neuen Platz beschaffen
        strcpy(temp, start);           // umkopieren
        delete [] start;               // alten Platz freigeben
        start = temp;                  // Verwaltungsinformation aktualisieren
        cap = len;                     // Verwaltungsinformation aktualisieren
    }
}

MeinString& MeinString::assign(const MeinString& m) { // Zuweisung eines MeinString
    reserve_only(m.len);
    strcpy(start, m.start);
    len = m.len;
    return *this;
}

MeinString& MeinString::assign(const char *s) { // Zuweisung eines char*
    size_t temp = strlen(s);
    reserve_only(temp);
    strcpy(start, s);
    len = temp;
    return *this;
}

```

Die `swap()`-Methode aus dem vorherigen Abschnitt funktioniert hier nicht, weil dort bei der Kopie stets Speicherplatz angefordert wird. Hier wird Speicherplatz nur dann angefordert, wenn der alte Platz nicht reicht.

Die `assign()`-Methoden benutzen `reserve_only()`. Bitte beachten Sie, dass das Attribut `len` für die Anzahl der Zeichen erst *nach* dem Aufruf von `reserve_only()` aktualisiert wird. Der Grund: Falls etwas bei der Speicherbeschaffung schief gehen sollte, bliebe das `MeinString`-Objekt in einem unveränderten Zustand. Der Aufruf der Methoden ist daher *exception safe*.



Mehr zum Thema Exception-Safety lesen Sie in Abschnitt 20.3.



Übung

6.1 Ergänzen Sie die Klasse `MeinString` um eine Methode `insert(size_t pos, const MeinString& m)`, die den Inhalt von `m` vor `pos` einfügt, also am Anfang, wenn `pos = 0` ist.

6.1.2 friend-Funktionen

Die oben gezeigte Schleife zur Anzeige eines `MeinString`-Objekts könnte in eine Funktion `anzeigen()` gepackt werden, für die hier drei verschiedene Möglichkeiten beschrieben werden sollen:

1. Elementfunktion `void anzeigen(std::ostream &os) const;`

Die Methode müsste in der Klasse deklariert werden. Der Parameter `os` vom Typ `ostream&` erlaubt die Ausgabe nicht nur auf der Standardausgabe, sondern auch auf dem Fehlerkanal oder in eine Datei:

```
einString.anzeigen(std::cout);
einString.anzeigen(std::cerr);
```

Innerhalb der Methode kann direkt auf `start` zugegriffen werden, weil `start` vom Grunddatentyp `char*` ist und der Ausgabeoperator `<<` in der Klasse `ostream` für alle Grunddatentypen definiert ist:

```
void MeinString::anzeigen(std::ostream& os) const { // Version 1
    os << start;
}
```

2. Globale Funktion `void anzeigen(std::ostream&, const MeinString&);`

Diese Funktion braucht dank `c_str()` nicht direkt auf die privaten Daten zuzugreifen. Das String-Objekt wird jetzt als Parameter übergeben.

```
// Aufruf:
anzeigen(std::cout, einString);
// Implementation (Version 2):
void anzeigen(std::ostream& os, const MeinString& m) {
    os << m.c_str();
}
```

3. `friend`-Funktion

Falls keine der beiden Möglichkeiten benutzt werden soll (wofür es hier keine guten Gründe gibt), kann ein Mittelding aus den beiden vorangegangenen Möglichkeiten gebildet werden, nämlich eine `friend`-Funktion. Eine als `friend` deklarierte Funktion ist *keine* Methode der Klasse, hat aber das Recht, auf deren private Daten zuzugreifen. Dies muss der Klasse natürlich bekannt sein. Das Schlüsselwort `friend` in der Deklaration sorgt dafür. Im Programmcode wird auf private Daten zugegriffen:

```
// Version 3
// Deklaration innerhalb der Klasse (meinstring.h)
friend void anzeigen(std::ostream& os, const MeinString& m);
// Definition (meinstring.cpp)
void anzeigen(std::ostream& os, const MeinString& m) {
    os << m.start;
}
```

Der Aufruf der Funktion wird im dritten Fall genauso wie im zweiten Fall geschrieben. Weil die Datenkapselung durch `friend`-Deklarationen durchlöchert wird, sollte man sparsam damit umgehen. Nicht nur fremden Funktionen, sondern auch anderen Klassen kann der Zugriff auf private Daten gestattet werden, wenn dies wegen des engen Zusammenwirkens der Klassen notwendig ist.

6.2 Klassenspezifische Daten und Funktionen

Klassenspezifische Daten sind Daten, die *nur einmal* für *alle* Objekte einer Klasse existieren. Sie sind also nicht an einzelne Objekte, sondern an alle Objekte einer Klasse gleichzeitig gebunden. Dies können Verwaltungsdaten wie die Anzahl der Objekte sein oder auch Bezugsdaten, die für alle Objekte gelten, wie ein gemeinsamer Koordinatenursprung für grafische Objekte. Diese von mehreren Objekten einer Klasse *gemeinsam* benutzbaren Daten müssen nicht global sein. Sie können genauso gut gekapselt werden wie andere Daten eines Objekts. Sie sind dann *innerhalb einer Klasse* und *nur für Objekte dieser Klasse* gleichermaßen zugreifbar. Diese Daten sind `static` – eine weitere Bedeutung dieses Schlüsselworts (die bisher bekannten sind »Variable für Funktionen mit Gedächtnis« und »nur in dieser Datei gültige Deklaration«). Bei der Definition und Initialisierung wird für eine beliebige Anzahl von Objekten einer Klasse nur *ein* Speicherplatz pro `static`-Element angelegt, auf den von *allen* Objekten dieser Klasse zugegriffen werden kann. Klassenspezifische Funktionen führen Aufgaben aus, die *an eine Klasse, nicht aber an ein Objekt gebunden sind*. Sie können zum Beispiel mit den `static`-Daten arbeiten. Auch Konstruktor sind klassenspezifische Funktionen, weil das zu konstruierende Objekt beim Aufruf noch nicht existiert. Das Beispiel demonstriert sowohl klassenspezifische Daten als auch klassenspezifische Funktionen.

Listing 6.2: Klasse `NummeriertesObjekt`, erste Version

```
// cppbuch/k6/numobj/numobj.h
#ifndef NUMOBJ_H
#define NUMOBJ_H

class NummeriertesObjekt { // noch nicht vollständig! (siehe Text)
public:
    NummeriertesObjekt();
    NummeriertesObjekt(const NummeriertesObjekt&);
    ~NummeriertesObjekt();
    unsigned long seriennummer() const {
        return serienNr;
    }
    static int anzahl() {
        return anz;
    }
    static bool testmodus;
private:
    static int anz;    // int statt unsigned (s. Text)
    static unsigned long maxNummer;
    const unsigned long serienNr;
};
#endif // Ende von numobj.h
```

Die Klasse `NummeriertesObjekt` hat die Aufgabe, jedem Objekt eine unverwechselbare Seriennummer mitzugeben und über die aktuelle Anzahl aktiver Objekte Buch zu führen.

Die static-Funktion `anzahl()` soll die momentane Anzahl der Objekte dieser Klasse zurückgeben und ist daher objektunabhängig. Die Funktion `seriennummer()` bezieht sich im Gegensatz dazu nur auf ein einzelnes Objekt. Die öffentliche Variable `testmodus` dient dazu, während der Laufzeit eines Programms den Testmodus für bestimmte Programmabschnitte aus- oder einzuschalten, um auf der Standardausgabe Entstehen und Vergehen aller Objekte zu dokumentieren.

Implementation

In der hier diskutierten Implementierung (siehe auch `cppbuch/k6/numobj/numobj.cpp`) werden die Namespace-Bezeichner bei der Standardausgabe und `endl` weggelassen, weil sie vorab bekannt gemacht werden:

```
using std::cout; // erfordert #include<iostream>
using std::endl;
```

Zur Initialisierung der static-Attribute genügt die Angabe von Typ, Klasse und Variablenname. Man sieht auch daran, dass die Initialisierung nicht an ein einzelnes Objekt gebunden ist. Die Initialisierung ist gleichzeitig die Definition der Variablen, die nur genau einmal im Programm vorhanden sein darf (one definition rule, siehe Seite 127).

```
// Initialisierung und Definition der klassenspezifischen Variablen:
int      NummeriertesObjekt::anz      = 0;
unsigned long NummeriertesObjekt::maxNumber = 0L;
bool     NummeriertesObjekt::testmodus = false;
```

Sie kann nicht in einen Konstruktor verlegt werden, weil sie sonst bei jeder Erzeugung eines Objekts durchgeführt würde. Der Standardkonstruktor initialisiert die objektspezifische Konstante `serienNr` in der Initialisierungsliste und aktualisiert die Anzahl aller Objekte:

```
// Standardkonstruktor
NummeriertesObjekt::NummeriertesObjekt()
: serienNr(++maxNumber) {
    ++anz;
    if(testmodus) {
        if(serienNr == 1) {
            cout << "Start der Objekterzeugung!\n";
        }
        cout << " Objekt Nr. " << serienNr << " erzeugt" << endl;
    }
}
```

Der Kopierkonstruktor hat hier eine besondere Bedeutung. Bisher diente er dazu, ein Duplikat eines Objekts bei der Initialisierung zu erzeugen. Das darf hier nicht sein! Das neu erzeugte Objekt soll nicht die Seriennummer eines anderen erhalten, sondern eine neue bekommen, weil sie sonst nicht eindeutig wäre. Der Kopierkonstruktor muss also dasselbe Verhalten wie der Standardkonstruktor aufweisen, mit Ausnahme des Testmodus, damit der unterschiedliche Aufruf protokolliert wird.

```
// Kopierkonstruktor
NummeriertesObjekt::NummeriertesObjekt( const NummeriertesObjekt &X)
```



```

: serienNr(++maxNumber) {
    ++anz;
    if(testmodus) {
        cout << " Objekt Nr. " << serienNr
            << " mit Nr. " << X.seriennummer() // bzw. X.serienNr
            << " initialisiert " << endl;
    }
}

```

Die Klassendeklaration auf Seite 242 ist noch nicht vollständig: Was geschieht bei der Zuweisung eines Objekts? Der systemerzeugte Zuweisungsoperator würde bei einer Zuweisung wie zum Beispiel

```

NummeriertesObjekt NumObjekt1;
NummeriertesObjekt NumObjekt2;
// ... irgendwelcher Programmcode
NumObjekt2 = NumObjekt1;           // ?

```

eine Zuweisung der Elemente von NumObjekt1 an NumObjekt2 bewirken. Das einzige Element, das dafür in Frage kommt, ist die private Konstante `serienNr`. Einer Konstanten kann aber nichts zugewiesen werden, sodass ein funktionierender systemerzeugter Zuweisungsoperator nicht erzeugt werden kann. Wenn die Zuweisung überhaupt erlaubt sein soll, darf sie einfach nichts bewirken! Dieses Problem können Sie leicht nach Studium des Kapitels 9 lösen, wo in einer Übungsaufgabe (Seite 330) auf dieses Problem eingegangen wird.

Der Destruktor vermerkt, dass es nun ein Objekt weniger gibt. Er wird am Ende eines Blocks und bei der Löschung dynamischer Objekte durch `delete` aufgerufen. Ein versehentliches zusätzliches `delete` kann vom Destruktor festgestellt werden, wenn nämlich `anz` negativ wird. Aus diesem Grund ist `anz` vom Typ `int` und nicht `unsigned int`. Die Zusicherung am Ende des Destruktors garantiert, dass das Löschen eines »hängenden Zeigers« nur im Testmodus gemeldet und toleriert wird.

```

// Destruktor
NummeriertesObjekt::~~NummeriertesObjekt() {
    --anz;
    if (testmodus) {
        cout << " Objekt Nr. "
            << serienNr << " gelöscht" << endl;
        if (anz == 0) {
            cout << "letztes Objekt gelöscht!" << endl;
        }
        if (anz < 0) {
            cout << "FEHLER! zu oft delete aufgerufen!" << endl;
        }
    }
    else {
        assert(anz >= 0);
    }
}

```

Konstruktor und Destruktor dokumentieren Werden und Vergehen der Objekte, sofern der Testmodus eingeschaltet ist. Klassenspezifische Funktionen können auch objektgebunden

aufgerufen werden, wie `main()` unten zeigt. Der klassenbezogene Aufruf einer Funktion oder die klassenbezogene Benennung eines Attributs ist dem objektgebundenen Aufruf vorzuziehen, weil der objektgebundene Aufruf die `static`-Eigenschaft verschleiert:

```
// schlechter Stil:
cout << dasNumObjekt.anzahl(); // objektgebundener Aufruf
cout << zeigerAufNumObjekt->anzahl(); // objektgebundener Aufruf
// richtig:
cout << NummeriertesObjekt::anzahl(); // klassenbezogener Aufruf
```

Wenn die letzte Zeile des Destruktors aufgerufen wird, gibt es mindestens eine `delete`-Anweisung zu viel. Zuwenige `delete`-Anweisungen können dagegen nicht zuverlässig ermittelt werden: Die Überprüfung würde durch einen Überschuss von mit `new` erzeugten und nicht gelöschten Objekten ausgetrickst werden.

6.2.1 Klassenspezifische Konstante

Klassenspezifische Variable müssen außerhalb der Klassendefinition definiert und initialisiert werden. Dies gilt nicht für klassenspezifische Konstanten, für die der Compiler keinen Platz anlegen muss, weil er direkt ihren Wert einsetzen kann. In C++ ist diese Ausnahme jedoch auf integrale und Aufzählungstypen beschränkt:

```
class KlasseMitKonstanten {
    enum RGB {rot = 0x0001, gelb = 0x0002, blau = 0x0004};
    static const unsigned int MAX_ZAHL = 1000;
    // Verwendung zum Beispiel:
    static int cArray[MAX_ZAHL];
    // ..
};
```

Listing 6.3: Klassenmethoden und Daten

```
// cppbuch/k6/numobj/nummain.cpp
// Demonstration von nummerierten Objekten
#include "numobj.h"
#include <iostream>
using namespace std;

int main() {
    // Testmodus für alle Objekte der Klasse einschalten
    NummeriertesObjekt::testmodus = true;
    NummeriertesObjekt dasNumObjekt_X; // ... wird erzeugt
    cout << "Die Seriennummer von dasNumObjekt_X ist: "
         << dasNumObjekt_X.seriennummer() << endl;

    // Anfang eines neuen Blocks
    {
        NummeriertesObjekt dasNumObjekt_Y; // ... wird erzeugt
        cout << NummeriertesObjekt::anzahl()
             << " Objekte aktiv" << endl;
        NummeriertesObjekt *p = new NummeriertesObjekt; // dynamisch erzeugt
        cout << NummeriertesObjekt::anzahl() << " Objekte aktiv" << endl;
        delete p; // *p wird gelöscht
        cout << NummeriertesObjekt::anzahl() << " Objekte aktiv" << endl;
```

```

        delete p; // Fehler: ein delete zu viel!
    }           // Blockende: dasNumObjekt_Y wird gelöscht

    cout << " Kopierkonstruktor: " << endl;
    NummeriertesObjekt dasNumObjekt_X1 = dasNumObjekt_X;
    cout << "Die Seriennummer von dasNumObjekt_X ist: "
        << dasNumObjekt_X.seriennummer() << endl;
    cout << "Die Seriennummer von dasNumObjekt_X1 ist: "
        << dasNumObjekt_X1.seriennummer() << endl;
    // Zuweisung wird wegen Konstanz der serienNr vom Compiler verboten
    dasNumObjekt_X1 = dasNumObjekt_X; // Fehler
} // dasNumObjekt_X und dasNumObjekt_X1 werden gelöscht

```

Diese Konstanten werden *innerhalb* der Klassendefinition initialisiert. Dabei wird `enum` stets ohne das Schlüsselwort `static` deklariert. In allen anderen Fällen muss es bei klassenspezifischen Objekten und Funktionen angegeben werden.

6.3 Klassen-Templates

Genau wie für Funktionen Templates definiert werden können (siehe Seite 134), sind Templates für Klassen möglich. Sie werden auch »parametrisierte Datentypen« genannt. Das Prinzip soll hier kurz dargestellt werden, indem wir einen Datentyp für einen einfachen Stapel (englisch *stack*), genannt `SimpleStack`, entwerfen.

6.3.1 Ein Stack-Template

Ein Stack hat die Eigenschaft, dass auf ihm Elemente abgelegt und wieder entnommen werden können, wobei die Reihenfolge der Entnahme entgegengesetzt der Ablage ist – wie bei einem Stapel von Tellern, auf den nur von oben zugegriffen wird. Die ab Seite 776 beschriebene Stack-Klasse der C++-Standardbibliothek basiert auf Templates. Um zu zeigen, wie es geht, soll hier ein Stack als *Template* für verschiedene Datentypen konstruiert werden. Die englischen Namen der Methoden sind auch in der deutschen Informatikwelt weit verbreitet und werden deshalb beibehalten. Zunächst setzen wir voraus, dass der Stack maximal 20 Elemente aufnehmen kann. Der dafür benötigte Behälter ist ein C-Array. Die auf Funktions-Templates bezogenen Erläuterungen bezüglich Dateien mit Templates auf Seite 138 gelten ebenso für Klassen-Templates. Prototyp und Definitionen sind in einer Datei *simstack1.t* zusammengefasst.

Der Datentyp τ in der folgenden Template-Klasse steht für einen beliebigen Datentyp als *Platzhalter*. Bei der Definition der Methoden außerhalb der Klasse muss der Datentyp in spitzen Klammern zusätzlich zum Namen der Klasse angegeben werden ($\langle \tau \rangle$). Innerhalb der Klassendefinition wird der Typ τ bei den Prototypen der Methoden vorausgesetzt, wenn er nicht angegeben ist. Bei der Benutzung in einem Programm wird ein konkreter Datentyp angegeben, der nicht nur eine Klasse, sondern auch ein Grunddatentyp sein kann. Der Compiler erzeugt damit Objekte dieses Datentyps nach dem Vorbild des Tem-

plates. Die Anwendung des Klassen-Templates zeigt das Beispiel auf Seite 247, in dem zwei Stacks unterschiedlichen Datentyps mit dem Template `SimpleStack` erzeugt werden.

Listing 6.4: Template-Klasse für einen einfachen Stack

```
// cppbuch/k6/stack/simstack1.t    ein einfaches Stack-Template
#ifndef SIMSTACK1_T
#define SIMSTACK1_T
#include<cassert>

template<typename T>
class SimpleStack {
public:
    static const unsigned int MAX_SIZE = 20; // siehe Text
    SimpleStack() : anzahl(0) {}
    bool empty() const { return anzahl == 0; }
    bool full() const { return anzahl == MAX_SIZE; }
    unsigned int size() const { return anzahl; }
    void clear() { anzahl = 0; } // Stack leeren
    const T& top() const; // letztes Element sehen
    void pop(); // Element entfernen
    // Vorbedingung für top und pop: Stack ist nicht leer
    void push(const T& x); // x auf den Stack legen
    // Vorbedingung für push: Stack ist nicht voll
private:
    unsigned int anzahl;
    T array[MAX_SIZE]; // Behälter für Elemente
};

// noch fehlende Methoden-Implementierungen
template<typename T>
const T& SimpleStack<T>::top() const {
    assert(!empty());
    return array[anzahl-1];
}

template<typename T>
void SimpleStack<T>::pop() {
    assert(!empty());
    --anzahl;
}

template<typename T>
void SimpleStack<T>::push(const T& x) {
    assert(!full());
    array[anzahl++] = x;
}
#endif // SIMSTACK1_T
```

Listing 6.5: Anwendung des Stack-Templates

```
// cppbuch/k6/stack/simmain1.cpp
// Anwendungsbeispiele für Stack-Template
#include<iostream>
```

```

#include "simstack1.t"
using namespace std;

int main() {
    SimpleStack<int> einIntStack; // ein Stack für int-Zahlen
    int i = 100;
    while(!einIntStack.full()) {
        einIntStack.push(i++); // Stack füllen
    }
    cout << "Anzahl : " << einIntStack.size() << endl;
    // Stack-Methoden aufrufen
    cout << "oberstes Element: " << einIntStack.top() << endl;
    cout << "alle Elemente entnehmen und anzeigen: " << endl;
    while(!einIntStack.empty()) {
        i = einIntStack.top();
        einIntStack.pop();
        cout << i << '\t';
    }
    cout << endl;
    SimpleStack<double> einDoubleStack; // ein Stack für double-Zahlen
    // Stack mit (beliebigen) Werten füllen
    double d = 1.00234;
    while(!einDoubleStack.full()) {
        d = 1.1 * d;
        einDoubleStack.push(d);
        cout << einDoubleStack.top() << '\t';
    }
    // einDoubleStack.push(1099.986); // Fehler, da Stack voll
    cout << "\n4 Elemente des Double-Stacks entnehmen:" << endl;
    for(i = 0; i < 4; ++i) {
        cout << einDoubleStack.top() << '\t';
        einDoubleStack.pop();
    }
    cout << endl;
    cout << "Restliche Anzahl : " << einDoubleStack.size() << endl;
    cout << "clear Stack" << endl;
    einDoubleStack.clear();
    cout << "Anzahl : " << einDoubleStack.size() << endl;
    // einDoubleStack.pop(); // Fehler, da Stack leer
}

```

Die Erzeugung der SimpleStack-Objekte für verschiedene Datentypen geschieht erst beim Lesen der Definition durch den Compiler, der dann in Kenntnis der Template-Beschreibung in *simstack1.t* einen Stack für `int`- und einen für `double`-Zahlen erzeugt. Die Erzeugung eines Objekts für einen konkreten Datentyp anstelle des Platzhalters `T` im Template wird *Instanziierung eines Templates* genannt. Ein Stack kann mit Hilfe des Templates für ganz verschiedene Datentypen deklariert werden, zum Beispiel können wir Stacks für rationale Zahlen oder andere beliebige Objekte bauen, zum Beispiel Datumobjekte, falls wir den Typ `Datum` vorher definiert haben:

```

SimpleStack<Rational> einStackFuerRationaleZahlen;
SimpleStack<Datum> einStackFuerDaten;

```

6.3.2 Stack mit statisch festgelegter Größe

Einen kleinen Schönheitsfehler hat der `SimpleStack`: Seine Größe ist auf `MAX_SIZE` fest eingestellt. Es gäbe natürlich die Lösung, die Größe dem Konstruktor zu übergeben, der den benötigten Platz dynamisch mit `new` beschafft, und den Stack dynamisch je nach Bedarf zu erweitern.

Der Template-Mechanismus bietet jedoch auch eine statische Lösung: Innerhalb der spitzen Klammern `< >` können *mehrere* Datentypen (Klassen und Grunddatentypen) und Werte integraler Typen angegeben werden, die in ihrer Gesamtheit einen neuen Datentyp definieren. Die Größe eines Stacks gehört dann zum Datentyp, wird also zur Compilierzeit festgelegt. Die geringfügigen Änderungen in *simstack.t* sind

- Streichen der Konstante `MAX_SIZE`,
- Ersatz von `<class T>` durch `<class T, unsigned int MAX_SIZE>` und Anpassung der darauf aufbauenden Definitionen (siehe unten).

Die Deklarationen in einem Anwendungsprogramm sind ebenfalls zu modifizieren, die Benutzung bleibt sonst gleich:

Listing 6.6: Template-Klasse für einen Stack, 2. Version

```
// cppbuch/k6/stack/simstack2.t einfaches Stack-Template, 2. Version
#ifndef SIMSTACK2_T
#define SIMSTACK2_T
#include<cassert>

// Parameter MAX_SIZE zur Festlegung der Stackgröße
template<typename T, unsigned int MAX_SIZE>
class SimpleStack {
    // ... genau wie oben, aber ohne MAX_SIZE
};

// noch fehlende Implementierungen
template<typename T, unsigned int m> // Parameter m wird nicht benutzt
const T& SimpleStack<T, m>::top() const {
    assert(!empty());
    return array[anzahl-1];
}

template<typename T, unsigned int m> // Parameter m wird nicht benutzt
void SimpleStack<T, m>::pop() {
    assert(!empty());
    --anzahl;
}

template<typename T, unsigned int m> // Parameter m wird nicht benutzt
void SimpleStack<T, m>::push(const T& x) {
    assert(!full());
    array[anzahl++] = x;
}
#endif // SIMSTACK2_T
```

Eine mögliche Anwendung finden Sie auf der nächsten Seite.

```
// ein int-Stack mit max. 100 Elementen:
SimpleStack<int, 100> einIntStack;
// Stack füllen
int i;
while(!einIntStack.full()) {
    cin >> i;
    einIntStack.push(i);
}
```

```
// ein char-Stack mit max. 9 Elementen
SimpleStack<char, 9> einCharStack;
// ...
```

Weil die Größe `MAX_SIZE` nicht dem Objekt, sondern dem Template übergeben wurde, ist die Größe eines `SimpleStack` schon zur Übersetzungszeit bekannt, sodass `simpleStack`-Objekten statisch Speicherplatz zugeteilt wird, also ohne Rückgriff auf den dynamischen Speicher.



Übungen

6.2 Schreiben Sie eine Klasse `Format` zum Formatieren von Zahlen. Benutzungsbeispiel:

```
// Konstruktion des Format-Objekts
Format f(12, 3); // Ausgabe 12 Zeichen breit, 3 Nachkommastellen
cout << f.toString(789.906625) << endl; // Benutzung
cout << f.toString(-123456789.906625) << endl;
```

Das Ergebnis soll `789,907` im ersten Fall und `-123456789,907` im zweiten Fall sein, wobei hier für ein Leerzeichen steht. Im zweiten Fall reichen 12 Plätze zur Darstellung aller Ziffern nicht aus. Die Weite wird daher automatisch erweitert, um Informationsverlust zu vermeiden.

6.3 Das folgende Programmfragment soll einen Ausschnitt einer Party simulieren. Das Array `alle` fasst alle Teilnehmer zusammen; mit seiner Hilfe werden ihre Namen und die ihrer Bekannten ausgegeben.

```
int main() {
    Teilnehmer otto("Otto");
    Teilnehmer andrea("Andrea");
    Teilnehmer jens("Jens");
    Teilnehmer silvana("Silvana");
    Teilnehmer miriam("Miriam");
    Teilnehmer paul("Paul");
    Teilnehmer* const alle[] = {&otto, &andrea, &jens,
                               &silvana, &miriam, &paul, 0}; // 0 = Endeckennung
    andrea.lerntKennen(jens);
    silvana.lerntKennen(otto);
    paul.lerntKennen(otto);
    paul.lerntKennen(silvana);
    miriam.lerntKennen(andrea);
    jens.lerntKennen(miriam);
    jens.lerntKennen(silvana);
    if(jens.kennt(andrea)) {
        cout << "Jens kennt Andrea" << endl;
    }
}
```

```

    }
    int i = 0;
    // Ausgabe aller Teilnehmer mit Angabe, wer wen kennt:
    while(alle[i]) {
        cout << alle[i]->gibNamen() << " kennt: ";
        alle[i]->druckeBekannte();
        ++i;
    }
}

```

Schreiben Sie eine zu diesem Programm passende Klasse `Teilnehmer`. Dabei soll berücksichtigt werden, dass »kennenlernen« als »sich gegenseitig kennenlernen« gemeint ist. Wenn Paul also Silvana kennen lernt, lernt sie ihn umgekehrt auch kennen. Ein Methodenaufruf, der meint, jemand lernt sich selbst kennen (etwa `jens.lerntKennen(jens)`), soll ignoriert werden. Was muss für den Gültigkeitsbereich der Objekte gelten, wenn Sie die Bekannten eines Teilnehmers in einem Attribut des Typs `vector<Teilnehmer*>` speichern? Gibt es eine bessere Möglichkeit?

6.4 Template-Metaprogrammierung¹

Zur Vertiefung des Verständnisses von Templates wird gezeigt, wie der *Compiler* zum Rechnen gebracht werden kann. Diese Art der Programmierung von Templates heißt *Template-Metaprogrammierung*. Das folgende Programm berechnet eine Zweierpotenz, im Beispiel $2^{11} = 2048$. Der Compiler versucht bei der Übersetzung, den Wert des Attributs `Zweihoch<11>::wert` zu ermitteln. Die klassenspezifische Aufzählungskonstante `wert` hat für jeden Typ der Klasse `Zweihoch`, der von `n` abhängt, einen anderen Wert. Bei der Ermittlung stellt der Compiler fest, dass `Zweihoch<11>::wert` dasselbe wie `2·Zweihoch<10>::wert` ist. `Zweihoch<10>::wert` wiederum ist dasselbe wie `2·Zweihoch<9>::wert` usw.

Listing 6.7: Rekursive Templates

```

// cppbuch/k6/rekursiveTemplates/zweihoch.cpp
#include<iostream>

template<int n>
struct Zweihoch {
    enum { wert = 2*Zweihoch<n-1>::wert };
};
template<> struct Zweihoch<0> {
    enum { wert = 1 };
};

int main() {
    std::cout << Zweihoch<11>::wert << std::endl;
}

```

¹ Der Rest des Kapitels kann beim ersten Lesen übersprungen werden.

Die Rekursion bricht bei der Berechnung von `Zweihoch<0>::wert` ab, weil das Template für diesen Fall spezialisiert und der Wert mit 1 besetzt ist. Der Compiler erzeugt insgesamt 12 Datentypen (0 bis 11), die er zur Auswertung heranzieht. Weil er konstante Ausdrücke zur Compilationszeit kennt und berechnen kann, wird an die Stelle von `Zweihoch<11>::wert` direkt das Ergebnis 2048 eingetragen, sodass zur Laufzeit des Programms keinerlei Rechnungen mehr nötig sind!

Diese Methode zur Berechnung von Zweierpotenzen schlägt damit jede andere, was die Rechenzeit des Programms angeht. Dieses Verfahren wird mit gutem Erfolg erweitert auf andere Probleme wie zum Beispiel die Berechnung der schnellen Fouriertransformation und die Optimierung von Vektoroperationen ([Ve95]), stellt aber hohe Anforderungen an die verwendeten Compiler. Insbesondere ist die Tiefe der möglichen Template-Instanziierungen begrenzt.

Als Ergänzung werden im folgenden Beispiel Primzahlen vom *Compiler* berechnet, aber erst zur Laufzeit ausgegeben. Dabei gibt es keinerlei Schleifen oder Funktionsaufrufe, nur die statische, allerdings rekursive Konstruktion von Objekten. Die Spezialisierungen sorgen für den Abbruch der Rekursion. Anstelle der 17 kann eine andere Zahl stehen. Die mögliche Höchstzahl ist abhängig vom verwendeten Compiler. Dieses Programm wurde nach einer Idee von Erwin Unruh geschrieben, der 1994 ein Programm konstruierte, das bei Übersetzung Primzahlen in den Fehlermeldungen des Compilers erzeugte [Unr]. Für sich genommen, scheinen beide Beispiele eher Kuriositäten zu sein. Als Übung zum Verständnis der angegebenen weiterführenden Literatur [CE] und [Ve95] bzw. der darauf aufbauenden numerischen Bibliotheken sowie des folgenden Abschnitts sind sie aber gut geeignet.

Listing 6.8: Primzahlen mit Templates berechnen

```
// cppbuch/k6/rekursiveTemplates/primzahl.cpp
#include<iostream>
using namespace std;

template<int p, int i>
struct istPrimzahl {
    // p ist nur dann prim, wenn p nicht durch i teilbar ist und auch nicht durch
    // alle anderen Teiler zwischen 2 und i. Wenn i==2 ist, wird
    // istPrimzahl<0, 1>::prim gefragt, d.h. Abbruch der Rekursion (s.u.).
    enum {prim = (p%i) && istPrimzahl<(i)>? p:0}, i-1>:prim};
};

template<int i>
struct druckePrimzahlenBis {
    // Der folgende Konstruktoraufruf sorgt dafür, dass die
    // kleineren Primzahlen rekursiv ausgegeben werden.
    druckePrimzahlenBis<i-1> a;
    enum { prime = istPrimzahl<i, i-1>:prim};
    druckePrimzahlenBis() {
        if(prime) {
            cout << i << endl;
        }
    }
};
```

```
// Rekursionsabbruch durch Spezialisierungen
template<> struct istPrimzahl<0,1> { enum {prim = 1};};
template<> struct druckePrimzahlenBis<2> { //
    druckePrimzahlenBis() { cout << 2 << endl; }
};

int main() {
    druckePrimzahlenBis<17> a;
}
```

Die in diesem Abschnitt diskutierte Metaprogrammierung bekommt steigendes Gewicht. Es geht darum, schon zur Compilationszeit abhängig vom Typ oder bestimmten Eigenschaften Entscheidungen zu treffen, etwa welcher Algorithmus gewählt werden soll. Diese Typinformationen sind in sogenannten Traits-Klassen festgelegt. Traits heißt Eigenschaft oder Merkmal. Der C++-Standard unterstützt Metaprogrammierung durch Bereitstellung von Templates für Typinformationen. Die vordefinierten Typ-Klassen der Standardbibliothek finden Sie im Header `<type_traits>`, weitere Informationen in [ISOC++, Abschnitt 20.7]. Der Abschnitt 29.1.1 dieses Buchs zeigt die Anwendung von Traits.

Das obige Beispiel zeigt, dass schon zur Compilationszeit gerechnet werden kann. Schleifen werden dabei stets mit einer Rekursion realisiert – wie auch im nächsten Abschnitt zu sehen. Menschen mit Lisp- oder Prolog-Erfahrung wird das bekannt vorkommen.

6.5 Variadic Templates: Templates mit variabler Parameterzahl

Die Anzahl der Parameter von Funktionen und Operatoren wird Stelligkeit oder Arität genannt. Zum Beispiel ist die Addition zweistellig, weil sie zwei Argumente benötigt. Bis zu diesem Abschnitt werden Templates mit einer festen Anzahl von Parametern (Typen) definiert. Bei der Benutzung müssen die Typen entsprechend der Anzahl angegeben werden. Die Stelligkeit ist festgelegt. So hat das Stack-Template von Seite 249 die Stelligkeit zwei:

```
template<typename T, unsigned int MAX_SIZE> // zwei Parameter
class SimpleStack {
    // ... Rest weggelassen
};
```

Die C++-Standardbibliothek kann in vielen Teilen einfacher geschrieben werden, seitdem es Templates mit variabler Stelligkeit (englisch *variadic templates*) gibt. Ein Beispiel dafür ist die Bibliotheksklasse `tuple` (für Tupel) von Seite 752. Ein anderes Beispiel ist eine Funktion zum Ausdrucken aller Parameter, bei der man vorher noch nicht weiß, mit wie vielen Argumenten sie aufgerufen werden wird. Für jede beliebige Anzahl von Parametern jeweils eine Funktion zu schreiben, ist nicht praktikabel. Aus diesem Grund gibt es in der Programmiersprache C die Funktion `int printf(const char* format, ...)`,

der eine beliebig lange Parameterliste übergeben werden kann. `format` ist der C-String, der die Formatierung steuert. Die drei Punkte heißen Ellipse, was Auslassung bedeutet. Sie stehen für eine Folge von Parametern. So gibt der Aufruf `printf("Wert = %.4f\n", 1.2345678)`, die Zeile »Wert = 1.2346« aus. Dabei meint `%.4f\n`, dass eine Float-Zahl mit 4 Dezimalstellen nach dem Komma ausgegeben und dann eine neue Zeile begonnen wird. `printf()` gibt die Anzahl der ausgegebenen Zeichen zurück bzw. -1 bei einem Fehler. `printf()` ist jedoch nicht typsicher, das heißt, falsche Typen in der Parameterliste können nicht schon vom Compiler entdeckt werden.

Douglas Gregor hatte unter der Überschrift »Variadic Templates« einen Vorschlag, wie Ellipsen typsicher gestaltet werden können, entwickelt. Der Vorschlag wurde in den C++-Standard aufgenommen. Einen Übersichtsartikel, aus dem die folgenden, leicht abgewandelten Beispiele stammen, finden Sie unter [GrJ]. Wie Templates mit variabler Stelligkeit funktionieren, sei am Beispiel einer Funktion `anzeigen()` demonstriert, der beliebig viele Parameter zur Ausgabe mit `cout <<` übergeben werden können. Dabei prüft der Compiler, ob der Typ eines Parameters überhaupt zum Ausgabeoperator `<<` passt. Zur Vereinfachung wird auf die Steuerung des Ausgabeformats verzichtet.

Listing 6.9: Funktion mit variabler Parameteranzahl

```
// cppbuch/k6/variadicTemplate/anzeigen.cpp, nach [GrJ]
#include<iostream>

void anzeigen() {
    std::cout << std::endl;
}

template<typename T, typename... Rest>
void anzeigen(const T& obj, const Rest&... rest) {
    std::cout << obj << " ";
    anzeigen(rest...);
}

int main() {
    anzeigen(1);
    anzeigen(2, "Hallo");
    anzeigen("Text", 7.978353, 3);
}
```

Wie Sie sehen, wird `anzeigen()` mit einem bis drei Parametern aufgerufen. Das entscheidende Element ist die Template-Definition:

- Die Ellipse `...` nach `typename` bedeutet, dass an dieser Stelle null oder mehr Template-Parameter in `Rest` zusammengefasst werden. Der Parameter `Rest` wird »Template Parameter Pack« genannt. Wenn `anzeigen()` mit mehreren Parametern aufgerufen wird, wird der erste `T` zugeordnet, alle anderen dem Parameter `Rest` – daher der Name.
- In der Parameterliste der Funktion wird `Rest` fast wie ein normaler Typ verwendet. Der syntaktische Unterschied besteht nur darin, dass eine Ellipse `...` folgt, um die Eigenschaft »Template Parameter Pack« zu markieren.

- Dasselbe gilt für den Funktionsaufruf `anzeigen(rest...)`: In diesem Aufruf ist `rest...`, gekennzeichnet durch eine Ellipse, ein »Template Parameter Pack«. Dieser Aufruf hat einen Parameter weniger als die aufrufende Funktion!

Wie verarbeitet der Compiler die Anweisung `anzeigen("Text", 7.978353, 3);`? Die Abfolge in einzelnen Schritten ist:

- Zuerst wird eine Ausgabe für das erste Objekt, den C-String `"Text"`, erzeugt. Der anschließende Aufruf `anzeigen(rest...)`, ist nichts anderes als `anzeigen(7.978353, 3)`; – es wird nur der Rest übergeben.
- Der Aufruf `anzeigen(7.978353, 3)` wird genauso behandelt; es wird 7.978353 ausgegeben und dann `anzeigen(3)` aufgerufen.
- `anzeigen(3)` resultiert in der Ausgabe von 3 und dem Aufruf `anzeigen()`.
- `anzeigen()` ruft die Funktion ohne Parameter oben in der Datei. Diese Funktion beendet nur die laufende Zeile.

Der Compiler erzeugt aus dem Aufruf `anzeigen("Text", 7.978353, 3)` also die folgenden Funktionen aus dem Template:

```
void anzeigen(const char*, double, int);
void anzeigen(double, int);
void anzeigen(int);
```

Die Funktion ohne Argumente (`anzeigen()`) ist vorhanden und wird daher nicht erzeugt. Sie beendet die Rekursion.

Anzahl der Parameter ermitteln

Templates mit variabler Stelligkeit sind auch für Klassen möglich, wie hier am Beispiel einer Struktur zum Zählen der Parameteranzahl gezeigt wird. Die Auswertung geschieht zur Compilationszeit:

Listing 6.10: Anzahl der Parameter ermitteln

```
// cppbuch/k6/variadicTemplate/paramzaehlen.cpp, nach [GrJ]
#include<iostream>

// Template-Deklaration
template<typename... Args> struct Anzahl;

// partielle Spezialisierung (Rekursion)
template<typename Head, typename... Tail>
struct Anzahl<Head, Tail...> {
    static const int wert = 1 + Anzahl<Tail...>::wert;
};

// partielle Spezialisierung (Rekursionsabbruch)
template<>
struct Anzahl<> {
    static const int wert = 0;
};

int main() {
    std::cout << "Parameteranzahl von Anzahl<char*, int, double>: "
```

```
    << Anzahl<char*, int, double>::wert << std::endl;  
}
```

Ganz oben steht die Template-Deklaration. Eine Implementierung fehlt, weil sie nicht gebraucht wird. Anschließend sehen Sie eine partielle Spezialisierung, in der die Liste der Typen in das erste Element (`Head`) und den Rest (`Tail`) zerlegt wird. Bei der Auswertung dieser Spezialisierung stellt der Compiler fest, dass er zur Berechnung `Anzahl<Tail...>::wert` benötigt. Dazu muss er den Typ `Anzahl<Tail...>` instanziiieren, dessen Typliste nunmehr um ein Element verkürzt ist. Auf `Anzahl<Tail...>` wird dasselbe Verfahren angewendet, sodass auch `Tail` in einen Kopf und einen Rest verwandelt wird usw. Dieser Prozess wird abgearbeitet, bis die Typliste leer ist. Dann kommt die zweite Spezialisierung zur Geltung, und die Rekursion bricht ab. Der Compiler erzeugt also bei der Auswertung von `Anzahl<char*, int, double>::wert` sukzessive die folgenden Typen aus dem Template:

```
Anzahl<char*, int, double>  
Anzahl<int, double>  
Anzahl<double>
```

`Anzahl<>` wird, weil vorhanden, nicht erzeugt. Für jeden der neu erzeugten Typen wird die Konstante `wert` bereits zur Compilationszeit berechnet, sodass sich 3 als Ausgabe des Programms ergibt.

7

Vererbung

Dieses Kapitel behandelt die folgenden Themen:

- Vererbung und Objektorientierung
- Beziehung zwischen Ober- und Unterklasse
- Polymorphismus und seine Vorteile
- Mehrfachvererbung
- Typ eines Objekts zur Laufzeit eines Programms ermitteln

Der Vererbungsmechanismus zeichnet sich durch folgende Punkte aus:

- Eigenschaften, die einer Menge von Dingen gemeinsam sind, können als verallgemeinertes Konzept betrachtet werden, das besonders behandelt wird.
- Es gibt geringe Unterschiede zwischen diesen Dingen.
- Die Vererbung ist hierarchisch organisiert.

Ein Beispiel ist die Klassifizierung von Transportmitteln. Abbildung 7.1 auf der nächsten Seite zeigt sie in der UML-Notation (mehr zur UML siehe Seite 579). Die vererbende Klasse heißt *Oberklasse* oder *Basisklasse*, die erbende Klasse heißt Unterklasse oder *abgeleitete Klasse*. In der Literatur werden die Begriffe nicht einheitlich gebraucht. Im Falle von nur einer abgeleiteten Klasse ist die Oberklasse gleichzeitig die Basisklasse. Die Vererbung beschreibt eine *ist-ein*-Beziehung. Ein Fahrrad *ist ein* Landtransportmittel, ein Motorboot *ist ein* Wassertransportmittel. Die Vererbung ist eine gerichtete Beziehung, weil die Umkehrung im Allgemeinen nicht gilt: Ein Landtransportmittel ist nicht unbedingt ein Fahrrad.

Wie eine Klasse die Abstraktion von ähnlichen Eigenschaften und Verhaltensweisen ähnlicher Objekte ist, ist eine Oberklasse die *Abstraktion* oder *Generalisierung* von ähnlichen Eigenschaften und Verhaltensweisen der Unterklassen. Die Unterklasse fügt zu den allgemeinen Eigenschaften und Verhaltensweisen der Oberklasse nur die für diese Unterklasse spezifischen Dinge hinzu oder definiert das von der Oberklasse geerbte Verhalten neu. Die Unterklasse ist eine *Spezialisierung* der Oberklasse. Bei der Klassifikation von Objekten muss also nach Ähnlichkeiten und Unterschieden gefragt werden. Die Unterklasse *erbt* von der Oberklasse

- die Eigenschaften (Attribute, Daten) und
- das Verhalten (die Methoden).

Wenn eine Oberklasse bekannt ist, brauchen in einer zugehörigen Unterklasse nur die *Abweichungen* beschrieben zu werden. Alles andere kann *wieder verwendet* werden, weil es in der Oberklasse bereits vorliegt.

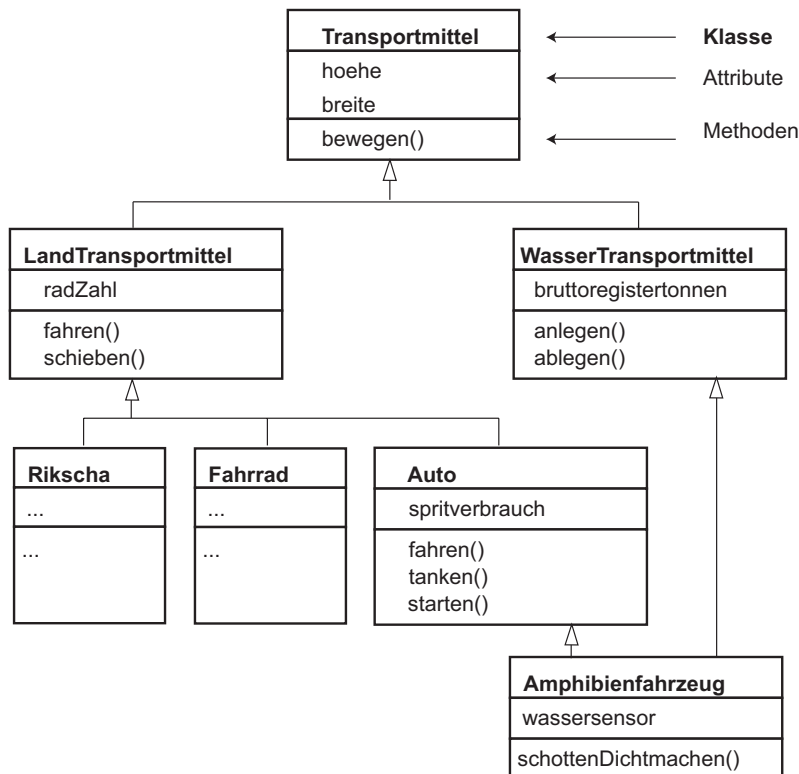


Abbildung 7.1: Vererbung von Daten und Methoden (unvollständige Klassen)

Die Menge der für ein Objekt zur Verfügung stehenden Methoden enthält die Methoden der Oberklasse(n) als Teilmenge. Umgekehrt sind alle Objekte einer Unterklasse (zum Beispiel Fahrräder) Teilmenge der möglichen Objekte einer Oberklasse (Landtransportmittel). Die Abstraktion wird durch »: public« ausgedrückt (siehe Syntaxdiagramm 7.2), es kann als »ist ein« oder »ist eine Art« gelesen werden.

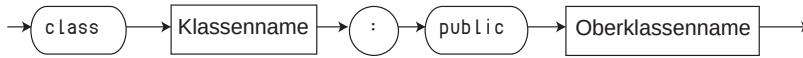


Abbildung 7.2: »: public« kennzeichnet Vererbung.

Ergänzend zur Abbildung 7.1 soll das Prinzip an der folgenden Formulierung in C++ gezeigt werden. Falls die Vererbung von Eigenschaften und Verhaltensweisen auf mehrere Oberklassen zurückgeführt werden kann, spricht man von Mehrfachvererbung (englisch *multiple inheritance*), hier gezeigt am Amphibienfahrzeug.

```

class Transportmittel {
public:
    void bewegen();
private:
    double hoehe, breite;
};

class LandTransportmittel : public Transportmittel { // erben
public:
    void fahren();
    void schieben();
private:
    int radZahl;
};

class WasserTransportmittel : public Transportmittel { // erben
public:
    void anlegen();
    void ablegen();
private:
    double bruttoregistertonnen;
};

class Auto : public LandTransportmittel {           // erben
public:
    void fahren(); // überschreibt LandTransportmittel::fahren()!
    void tanken();
    void starten();
private:
    double spritverbrauch;
};

class Amphibienfahrzeug:           // Mehrfachvererbung:
    public Auto, public WasserTransportmittel {
public:
    void schottenDichtmachen();
private:
    const char* wassersensor;
};
  
```

Wenn in C++ eine abgeleitete Klasse Abgeleitet von einer Oberklasse Oberklasse *erbt*, ist damit gemeint:

- Jedes Objekt `objAbgeleitet` vom Typ `Abgeleitet` enthält ein (anonymes) Objekt vom Typ `Oberklasse`, hier `Subobjekt` genannt, das entsprechend Speicher belegt. Dieses Subobjekt wird noch vor der Erzeugung von `objAbgeleitet` durch impliziten Aufruf des Oberklassenkonstruktors gebildet. Abbildung 7.3 zeigt, wie ein Oberklassenobjekt als Subobjekt in ein Objekt einer abgeleiteten Klasse eingebettet ist. Durch diesen Mechanismus wird erreicht, dass zu einem `Auto`-Objekt nicht nur der `spritverbrauch`, sondern auch `radZahl`, `hoehe` und `breite` als Attribute gehören.

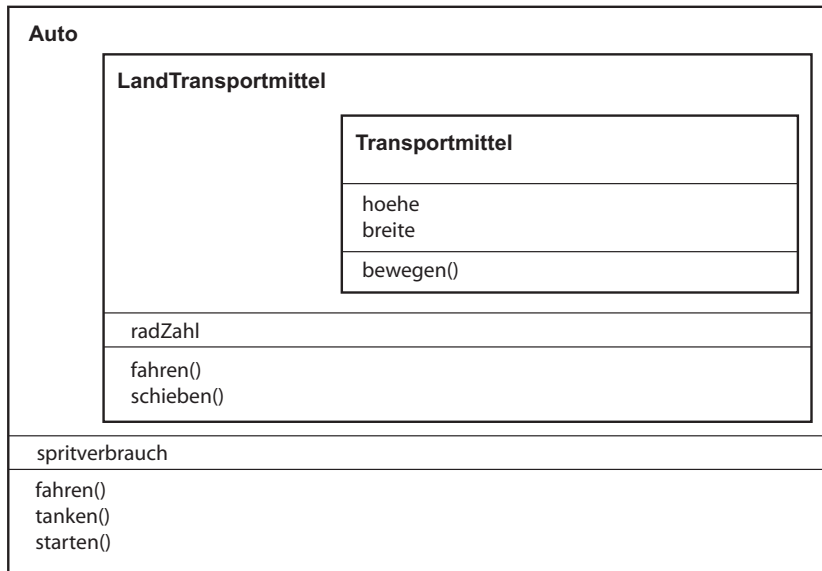


Abbildung 7.3: Einschluss von Subobjekten

- Jede Elementfunktion von `Oberklasse` kann auf ein Objekt des Typs `Abgeleitet` angewendet werden, sofern die Elementfunktion öffentlich zugänglich (`public`) ist. Die Funktion `bewegen()` ist benutzbar für ein Objekt `LandTransportmittel` ebenso wie für ein `Auto`, obwohl sie nicht speziell dort angegeben ist. Der Aufruf einer Operation für ein Objekt lässt nicht erkennen, ob sie der Klasse des Objekts oder einer Oberklasse zugeordnet ist.
- Die Klasse `Abgeleitet` kann Erweiterungen der Daten und zusätzliche Methoden enthalten, die keinen Bezug zur `Oberklasse` haben. Der `spritverbrauch` ist kennzeichnend für ein `Auto`. Es muss auch `tanken`, aber beides gilt nicht allgemein für ein (Land-) `Transportmittel`.
- Zusätzliche Methoden können in `Abgeleitet` deklariert werden, die in ihrer Signatur mit Elementfunktionen der Oberklasse übereinstimmen. Diese Methoden *überschreiben* die Elementfunktionen der Oberklasse bezüglich aller `Abgeleitet`-Objekte. In der Abbildung ist dies die Methode `fahren()`, die speziell für ein `Auto` entworfen wurde, weil die von `LandTransportmittel` geerbte Methode nicht geeignet ist.
- Eine Klasse ist ein *Datentyp* in C++. Eine abgeleitete Klasse kann als *Subtyp* der Oberklasse aufgefasst werden. Ein Objekt `objAbgeleitet` der abgeleiteten Klasse ist zuweisungskompatibel zu einem Objekt `objOberklasse` der Oberklasse. Die Zuweisung

```
objOberklasse = objAbgeleitet;
```

kopiert den Inhalt des in `objAbgeleitet` enthaltenen Subobjekts vom Typ `Oberklasse` nach `objOberklasse`. Die nur zu `objAbgeleitet` gehörenden spezifischen Daten werden nicht kopiert, weil in `objOberklasse` dafür kein Platz vorgesehen ist. Die Umkehrung `objAbgeleitet = objOberklasse` ist *nicht* möglich, weil der Abgeleitet-spezifische Teil undefiniert bleiben würde:

```
LandTransportmittel einLandTransportmittel;
Auto einAuto;
// ...
einLandTransportmittel = einAuto; // ok, aber Datenverlust
einAuto = einLandTransportmittel; // Fehler!
```

Auf die Subobjekte werde ich noch zurückkommen. In (älteren) Programmierhandbüchern findet man gelegentlich Beispiele, die Vererbung als *hat*-Beziehung einsetzen. Ein Kreis *hat* einen (Mittel-)Punkt, also wird von der Oberklasse Punkt geerbt. Dieses Vorgehen ist fehlerhaft, weil ein Kreis tatsächlich keine Spezialisierung eines Punktes darstellt: ein Kreis *ist kein* Punkt. Obwohl programmtechnisch möglich, sollte eine Vererbungshierarchie nicht *hat*-Beziehungen, auch Aggregation genannt, sondern Ebenen von Verallgemeinerungen darstellen. Im Folgenden wird die Basisklasse `GraphObj` (grafisches Objekt) als einfaches Beispiel verwendet. Sie wird in abgeleiteten Klassen (wie Linie, Rechteck, Dreieck) benutzt. Das Beispiel wird nach und nach entwickelt, ist also einigen Änderungen unterworfen. Alle möglichen auf dem Bildschirm sichtbaren Dinge sind grafische Objekte. Gemeinsam soll allen Objekten sein, dass jedes Objekt einen Bezugspunkt *referenzkoordinaten* in Pixelkoordinaten hat. Der Bezugspunkt soll nur über die Methode `bezugspunkt()` veränderbar sein, andererseits sollen die Koordinaten des Bezugspunktes von anderen gelesen werden können.

Klassen für grafische Objekte

Die Klasse `GraphObj` ist recht einfach. Das aggregierte Objekt *referenzkoordinaten* ist vom Typ `Ort`, der auf Seite 157 beschrieben ist. Nach `GraphObj` folgt eine Klasse `Strecke`, die von `GraphObj` erbt.

Listing 7.1: Klasse `GraphObj`, 1. Version

```
// cppbuch/k7/erben/graphobj.h
#ifndef GRAPHOBJ_H
#define GRAPHOBJ_H
#include "ort.h"

class GraphObj {                                // Version 1
public:
    GraphObj(const Ort& einOrt)    // allg. Konstruktor
    : referenzkoordinaten(einOrt) {
    }

    // Bezugspunkt ermitteln
    const Ort& bezugspunkt() const {
        return referenzkoordinaten;
    }
}
```

```

// alten Bezugspunkt ermitteln und gleichzeitig neuen wählen
Ort bezugspunkt(const Ort& n0) {
    Ort temp = referenzkoordinaten;
    referenzkoordinaten = n0;
    return temp;
}
// Koordinatenabfrage
int getX() const {
    return referenzkoordinaten.getX();
}
int getY() const {
    return referenzkoordinaten.getY();
}
// Standardimplementation:
double flaeche() const {return 0.0;}
private:
    Ort referenzkoordinaten;
};

// Die Entfernung zwischen 2 GraphObj-Objekten ist hier als Entfernung ihrer
// Bezugspunkte (überladene Funktion) definiert.
inline double entfernung(const GraphObj& g1,
                        const GraphObj& g2) {
    return entfernung(g1.bezugspunkt(), g2.bezugspunkt());
}
#endif // GRAPHOBJ_H

```

Alle Methoden sind wegen ihrer Kürze `inline`. Innerhalb der am Ende der Header-Datei definierten globalen Funktion `Entfernung( const GraphObj &g1, const GraphObj &g2)` wird die schon vorher in `ort.h` für die Bezugspunkte des Typs `Ort` definierte gleichnamige Funktion aufgerufen. Der Compiler erkennt die richtige Funktion an Anzahl und Typ der Parameter. Eine kleine Besonderheit besteht darin, dass die Methode `bezugspunkt()` überladen ist. Wenn sie ohne Parameter aufgerufen wird, gibt sie den Bezugspunkt zurück. Wenn sie mit einem `Ort` als Parameter aufgerufen wird, setzt sie diesen `Ort` als neuen Bezugspunkt, gibt aber den vorherigen Bezugspunkt zurück. Diese Technik wird beim Setzen von Attributen häufig verwendet, weil sie einem aufrufenden Programm die Möglichkeit gibt, ein Attribut zu ändern und sich dabei den alten Wert zu merken, um ihn später wieder einzusetzen. Der Rückgabewert kann natürlich auch verworfen werden.

Die Fläche eines allgemeinen grafischen Objekts ist eigentlich nicht 0, sondern undefiniert. Auf diese Besonderheit wird in Abschnitt 7.6.2 eingegangen. Bis dahin bietet die Funktion `flaeche()` eine Standardimplementation für abgeleitete Klassen. Damit ist klar, dass diese Funktion in einer abgeleiteten Klasse möglicherweise neu definiert werden muss, nicht in der nachfolgend besprochenen Klasse `Strecke`, wohl aber in einer Klasse `Rechteck`. Eine `Strecke` ist ein `GraphObj`. In der Klassendeklaration wird diese Beziehung syntaktisch durch `: public` und den Namen der Oberklasse ausgedrückt. Die Bedeutung von `public` an dieser Stelle wird im folgenden Abschnitt 7.2 erläutert.

Listing 7.2: Klasse `Strecke`

```

// cppbuch/k7/erben/strecke.h
#ifndef STRECKE_H

```

```

#define STRECKE_H
#include "graphobj.h"

class Strecke : public GraphObj { // erben von GraphObj
public:
    Strecke(const Ort& ort1, const Ort& ort2)
        : GraphObj(ort1), // Initialisierung des Subobjekts, siehe Kap. 7.1
          endeckpunkt(ort2) // Initialisierung des Attributs
    { } // leerer Code-Block
    double laenge() const {
        return entfecknung(bezugspunkt(), endeckpunkt);
    }
private:
    Ort endeckpunkt; // zusätzlich: 2. Punkt der Strecke
};
#endif // STRECKE_H

```

7.1 Vererbung und Initialisierung

Auf Seite 260 wurde darauf hingewiesen, dass jedes Objekt einer abgeleiteten Klasse ein anonymes Subobjekt der Oberklasse enthält. Der Oberklassenkonstruktor sollte bei der Initialisierung eines Objekts durch eine Liste (vergleiche Seite 156) explizit aufgerufen werden.

Nur ein Endpunkt der Strecke wird als Attribut angegeben, der andere wird geerbt (Attribut `GraphObj::referenzkoordinaten`). Der Konstruktor benötigt zwei Punkte zur Konstruktion der Strecke. Weil ein Objekt der Klasse `Strecke` ein anonymes Subobjekt der Klasse `GraphObj` enthält, ist der Anfangspunkt bereits durch die Referenzkoordinaten gegeben, und es ist nur noch ein Endpunkt als Attribut notwendig. Falls es einen Standardkonstruktor für die Klasse `GraphObj` gäbe, bräuhete man das Subobjekt nicht zu initialisieren und könnte den Konstruktor der Klasse `Strecke` wie folgt schreiben:

```

// nur bei Standardkonstruktor GraphObj() möglich, aber nicht empfehlenswert
Strecke(const Ort& ort1, const Ort& ort2) {
    bezugspunkt(ort1); // geerbter Code der Oberklasse
    endeckpunkt = ort2;
}

```

Es gibt aber keinen Standardkonstruktor `GraphObj()`; außerdem ist die Initialisierung mit einer Initialisierungsliste generell vorzuziehen, weil das Objekt in *einem* Schritt mit den richtigen Werten initialisiert wird, also

```

Strecke(const Ort& ort1, const Ort& ort2)
: GraphObj(ort1), // Initialisierung des Subobjekts
  endeckpunkt(ort2) { // Initialisierung des Attributs
} // leerer Code-Block

```

Die Initialisierung innerhalb des Blocks {...} ist aufwendiger, weil die Konstruktoren für alle Objektelemente stets *vor* Betreten des Blocks aufgerufen werden und beliebige Daten eintragen, die dann innerhalb des Blocks neu zugewiesen werden müssen. Dasselbe gilt für die Initialisierung von Subobjekten, wie hier für das in einem *Strecke*-Objekt enthaltene Subobjekt des Typs *GraphObj*. Die Initialisierungsliste darf enthalten:

- Elemente der Klasse selbst, aber keine geerbten Elemente;
- Konstruktoraufrufe der Oberklassen.

Nach dem folgenden Abschnitt über Zugriffsschutz wird das Beispiel wieder aufgegriffen.

7.2 Zugriffsschutz

Unter *Zugriffsschutz* ist die Abstufung von Zugriffsrechten auf Daten und Elementfunktionen zu verstehen. Bisher sind zwei Fälle bekannt:

- `public`
Elemente und Methoden unterliegen keiner Zugriffsbeschränkung.
- `private`
Elemente und Methoden sind ausschließlich innerhalb der Klasse zugreifbar sowie für `friend`-Klassen und -Funktionen.

Die Zugriffsspezifizierer `private` und `public` gelten genauso in einer Vererbungshierarchie. Um abgeleiteten Klassen gegenüber der »Öffentlichkeit« weitgehende Rechte einräumen zu können, ohne den privaten Status mancher Elemente aufzugeben, gibt es einen weiteren Zugriffsspezifizierer:

- `protected`
Elemente und Methoden sind in der eigenen und in allen `public` abgeleiteten Klassen zugreifbar, nicht aber in anderen Klassen oder außerhalb der Klasse.

Vererbung von Zugriffsrechten

Gegeben sei eine Oberklasse, von der eine weitere Klasse abgeleitet wird. Für die Vererbung der Zugriffsrechte gelten folgende *Regeln*, die weiter unten anhand einiger Beispiele verdeutlicht werden:

- `private`-Elemente sind in einer abgeleiteten Klasse nicht zugreifbar.
- In allen anderen Fällen gilt das jeweils restriktivere Zugriffsrecht, bezogen auf die Zugriffsrechte für ein Element und die Zugriffskennung der Vererbung einer Klasse. Beispiel: Ein `protected`-Element einer `private`-vererbten Klasse ist `private` in der abgeleiteten Klasse. Typischerweise werden jedoch Oberklassen `public` vererbt, so dass die Zugriffsrechte von Oberklassenelementen in abgeleiteten Klassen erhalten bleiben.

Tabelle 7.1 zeigt die Vererbung von Zugriffsrechten für den häufigen Fall der `public`-Vererbung. Die `private`- und `protected`-Vererbung werden Sie in Abschnitt 7.12 kennenlernen.

Tabelle 7.1: Zugriffsrechte bei public-Vererbung

Zugriffsrecht in der Basisklasse	Zugriffsrecht in einer abgeleiteten Klasse
private	kein Zugriff
protected	protected
public	public

Wenn anstatt `class` das Schlüsselwort `struct` geschrieben wird, ist die Voreinstellung `public`. Im Grunde ist »`struct s` {« nur eine Abkürzung für »`class s {public:«`. Man geht Zugriffskonflikten aus dem Weg, indem man überall `class` durch `struct` ersetzt – dann aber verletzt man das Prinzip der Datenkapselung! Besser ist es, sich genau zu überlegen, auf welche Daten und Funktionen der Oberklasse eine Klasse zugreifen darf, und im Zweifelsfall restriktiver vorzugehen. Das folgende Beispiel zeigt typische Möglichkeiten, die sich aus den Regeln ergeben. Alle Programmzeilen, die einen Zugriffsfehler ergeben, sind markiert.

```
class Oberklasse {
private:           // Voreinstellung
    int oberklassePriv;
    void privateFunktionOberklasse();
protected:
    int oberklasseProt;
public:
    int oberklassePubl;
    void publicFunktionOberklasse();
};

// Oberklasse wird mit der Zugriffskennung public vererbt
class AbgeleiteteKlasse : public Oberklasse {
    int abgeleiteteKlassePriv;
public:
    int abgeleiteteKlassePubl;

    void publicFunktionAbgeleiteteKlasse() {
        oberklassePriv = 1;           // Fehler: nicht zugreifbar
        // in einer abgeleiteten Klasse zugreifbar:
        oberklasseProt = 2;
        // generell zugreifbar
        oberklassePubl = 3;
    }
};

int main() {
    int m;
    AbgeleiteteKlasse objekt;
    m = objekt.oberklassePubl;
    m = objekt.oberklasseProt;           // Fehler: nicht zugreifbar
    m = objekt.oberklassePriv;           // Fehler: nicht zugreifbar
    m = Objekt.abgeleiteteKlassePubl;
    m = Objekt.abgeleiteteKlassePriv;    // Fehler: nicht zugreifbar
    Objekt.publicFunktionAbgeleiteteKlasse(); // ok
}
```

```
// Aufruf geerbter Funktionen
Objekt.publicFunktionOberklasse();
Objekt.privateFunktionOberklasse(); // Fehler: nicht zugreifbar
}
```

7.3 Typbeziehung zwischen Ober- und Unterklasse

Eine abgeleitete Klasse kann als *Subtyp* der Oberklasse aufgefasst werden (siehe Seite 260). Daher ist ein Objekt der abgeleiteten Klasse zuweisungskompatibel zu einem Objekt der Oberklasse:

```
GraphObj g(01); // 01, 02, 03 = Objekte vom Typ Ort
Strecke s(02, 03);
g = s;
```

Eine Strecke wird einem GraphObj zugewiesen, explizites Typumwandeln ist zwar möglich, aber nicht notwendig. Die Wirkung ist wie

```
g.referenzkoordinaten = s.referenzkoordinaten;
```

Eine direkte Zuweisung wäre natürlich wegen `private` nicht möglich. Der Endpunkt der Strecke wird *nicht* kopiert, da er in einem GraphObj nicht vorhanden ist. Es gibt also einen Informationsverlust. Die umgekehrte Zuweisung ist *nicht* möglich, weil dann Informationen undefiniert blieben: Die Zuweisung eines GraphObj-Objekts an ein Strecke-Objekt würde den zweiten Endpunkt undefiniert lassen. Die Typbeziehung zwischen Basisklasse und abgeleiteter Klasse kann umgangssprachlich am Beispiel verdeutlicht werden: »Alle Tannen sind Bäume, aber das Umgekehrte (alle Bäume sind Tannen) gilt nicht.« Dabei sind die Tannen Exemplare der Unterklasse und Bäume Exemplare der Oberklasse. Für Zeiger und Referenzen gilt entsprechend:

```
GraphObj& rg = g;      GraphObj *pg;
Strecke & rs = s;      Strecke *ps = &s;
rg = rs;               pg = ps; // erlaubte Zuweisungen
```

Zeiger und Referenzen vom Oberklassentyp (`pg`, `rg`) beziehen sich auf das im Objekt `s` enthaltene anonyme Subobjekt vom Typ `GraphObj`. Aus diesem Grund kann *ohne zusätzlichen Code* die Entfernung der Bezugspunkte zweier Strecken ohne Umweg über die Bezugspunkte berechnet werden.¹ Aufgerufen wird hier die in `graphobj.h` deklarierte globale Funktion

```
double entfernung(const GraphObj& g1, const GraphObj& g2);
```

¹ In der Parameterliste von `entfernung()` wurde der Datentyp `GraphObj` per Referenz anstatt per Wert deklariert, weil es im Allgemeinen schneller ist, und weil manche Compiler die Typumwandlung innerhalb der Parameterliste nur eingeschränkt erlauben.



Hinweis

Die Typbeziehung kann nicht auf Arrays übertragen werden! Auch wenn `GraphObj` Oberklasse von `Strecke` ist, folgt daraus nicht, dass ein `GraphObj`-Array Oberklasse eines `Strecke`-Arrays ist – ein C-Array ist gar keine Klasse. Obwohl syntaktisch korrekt (d.h. compilierbar), ist die folgende Anweisung daher falsch.

```
Oberklasse* array = new Unterklasse[4]; // sinnlos.
```

Der Compiler sieht bei `array` nur den statischen Typ. Die nicht geerbten Attribute eines Unterklassenobjekts sind nicht zugreifbar, weil nur die Oberklassenanteile abgespeichert werden (englisch *object slicing*). Die Adressen `&array[i]` ($0 \leq i < 4$) liegen nur `sizeof(Oberklasse)` Bytes auseinander, nicht `sizeof(Unterklasse)`.

7.4 Code-Wiederverwendung

In den abgeleiteten Klassen können Methoden der Oberklasse wiederverwendet werden, zum Beispiel die Methoden `bezugspunkt()` und `flaeche()`. Eine Folgerung der durch die `public`-Vererbung repräsentierten *ist-ein*-Beziehung besteht darin, dass für eine Klasse alles möglich sein soll, was für die Oberklasse möglich ist, wenn auch vielleicht mit Anpassungen. Der Code der Oberklasse wird dabei wiederverwendet. In Abschnitt 7.12 werden wir andere Fälle kennenlernen, in denen die Wiederverwendung von Programmcode sinnvoll und möglich ist, obwohl keine *ist-ein*-Beziehung zwischen Klassen besteht. Ein kleines Programm zeigt, was mit den bisherigen Deklarationen und Definitionen für die Klassen `GraphObj` und `Strecke` möglich ist.

Listing 7.3: Beispiel mit Klasse `Ort`

```
// cppbuch/k7/erben/main.cpp
#include "strecke.h"
using namespace std;

int main() {
    // Definition zweier grafischer Objekte
    Ort nullpunkt;
    GraphObj g0(nullpunkt);
    Ort einOrt(10, 20);
    GraphObj g1(einOrt);
    // Ausgabe beider Bezugspunkte auf verschiedene Art
    cout << "g0.getX() = " << g0.getX() << endl;
    cout << "g0.getY() = " << g0.getY() << endl;

    Ort ort = g1.bezugspunkt();
    cout << "ort.getX() = " << ort.getX() << endl;
    cout << "ort.getY() = " << ort.getY() << endl;
    // Ausgabe der Entfernung
    cout << "Entfernung = " << entfernung(g0, g1) << endl;
```



```

cout << "neuer Bezugspunkt für g0:" << endl;
g0.bezugspunkt(ort); // Rückgabewert wird hier ignoriert
cout << "g0.bezugspunkt() = ";
anzeigen(g0.bezugspunkt()); // ort.h, siehe Seite 157
cout << "\n Entfernung = " << entfernungen(g0, g1) << endl;
Ort anf;
Strecke s1(anf, ort);
cout << "Strecke von ";
anzeigen(anf);
cout << " bis ";
anzeigen(ort);
cout << "\n Fläche der Strecke s1 = "
    << s1.fläche() // geerbte Methode
    << endl;
cout << "Länge der Strecke s1 = "
    << s1.länge() // zusätzliche Methode
    << endl;

einOrt = Ort(20, 30); // Neuzuweisung
Ort o2(100, 50);
Strecke s2(einOrt, o2);
cout << "= Entfernung der Bezugspunkte: "
    << entfernungen(s1.bezugspunkt(), s2.bezugspunkt()) << endl;
cout << "Entfernung der Strecken s1, s2 = " << entfernungen(s1, s2) << endl;
// ...
}

```

Am Ende des Programmfragments wird zur Berechnung der Entfernung einmal

```
double entfernungen(const Ort&, const Ort&);
```

aus *ort.h* aufgerufen. Anschließend wird die Entfernung noch einmal ausgegeben, wobei jetzt die Strecken direkt als Aufrufparameter dienen. Wie kann das angehen, wo doch bisher keine Funktion zur Entfernungsberechnung mit *Strecke*-Objekten in der Parameterliste beschrieben wurde? Der Grund liegt in der Typbeziehung zwischen Basisklasse und abgeleiteter Klasse.

7.5 Überschreiben von Funktionen in abgeleiteten Klassen

In diesem Abschnitt geht es um das Überschreiben von Funktionen innerhalb einer Vererbungshierarchie. Einen ähnlichen Mechanismus hatten wir bereits in Abschnitt 3.2.5 auf Seite 114 kennengelernt, der das Überladen von gleichnamigen Funktionen *mit unterschiedlicher Schnittstelle* behandelte. Dieser Abschnitt zeigt die Wirkungsweise für *Elementfunktionen mit derselben Schnittstelle* in abgeleiteten Klassen, die *Überschreiben* genannt wird. Die Methoden *bezugspunkt()* und *fläche()* der Oberklasse *GraphObj*

können auch für die abgeleitete Klasse *Strecke* verwendet werden. Falls die gleiche Bedeutung gemeint ist, aber ein anderer Mechanismus zugrunde liegt, können Funktionen überschrieben werden. Um das zu zeigen, führen wir eine Klasse *Rechteck* ein:

Listing 7.4: Klasse *Rechteck*

```
// cppbuch/k7/erben/rechteck.h
#ifndef RECHTECK_H
#define RECHTECK_H
#include "graphobj.h"

class Rechteck : public GraphObj { // von GraphObj erben
public:
    Rechteck(const Ort& ort, int h, int b)
        : GraphObj(ort), dieHoehe(h), dieBreite(b) {

        double flaeche() const {
            // int-Überlauf vermeiden
            return static_cast<double>(dieHoehe) * dieBreite;
        }
    private:
        int dieHoehe, dieBreite;
};
#endif // RECHTECK_H
```

Am Beispiel der Flächenberechnung mit der Funktion *flaeche()* sehen wir das Prinzip des Überschreibens:

```
Rechteck rechteck(Ort(0,0), 20, 50);
cout << "rechteck.flaeche = "
    << rechteck.flaeche() << endl; // 1000
```

Dieses Mal wird nicht wie bei der Klasse *Strecke* als Ergebnis 0 ausgegeben, sondern der Zahlenwert 1000. Die Funktion überschreibt jetzt *GraphObj::flaeche()*. Wenn aus irgendwelchen Gründen (in diesem Beispiel nicht sinnvoll) dennoch die Oberklassenfunktion aufgerufen werden soll, müssen der Klassenname und der Bereichsoperator *::* angegeben werden:

```
cout << rechteck.GraphObj::flaeche(); // null!
```

Im Gegensatz zu den überladenen Funktionen von Abschnitt 3.2.5 (Seite 114) können überschreibende Funktionen in abgeleiteten Klassen die gleiche Signatur haben, weil der Compiler sich die Klasse zusätzlich zur Signatur merkt. Das normale Überladen ist weiterhin möglich. Es gibt einen weiteren Unterschied: Das Überladen von Nicht-Elementfunktionen funktioniert nur innerhalb desselben Gültigkeitsbereichs (siehe Seite 115), während die überschreibenden Elementfunktionen in verschiedenen Klassen und damit unterschiedlichen Gültigkeitsbereichen sind.



Hinweis

Überschriebene Funktionen sollen grundsätzlich virtuell sein. Was das bedeutet und warum es so sein soll, wird im nächsten Abschnitt erläutert.

7.6 Polymorphismus

Polymorphismus heißt auf Deutsch Vielgestaltigkeit. Damit ist in der objektorientierten Programmierung die Fähigkeit einer Variable gemeint, zur *Laufzeit* eines Programms auf verschiedene Objekte zu verweisen. Anders formuliert: Erst zur Laufzeit eines Programms wird die zu dem jeweiligen Objekt passende Realisierung einer Operation ermittelt. In C++ wird die Einschränkung getroffen, dass die Objekte abgeleiteten Klassen zugeordnet sind. Ein Funktionsaufruf muss irgendwann an eine Folge von auszuführenden Anweisungen gebunden werden. Wenn es erst während der Ausführung des Programms geschieht, wird der Vorgang *dynamisches* oder spätes *Binden* genannt, andernfalls statisches oder frühes Binden. Eine zur Laufzeit ausgewählte Methode heißt *virtuelle Funktion*. Trotz der äußerlichen Ähnlichkeit und der ähnlichen Absicht dahinter sind Überladen und Polymorphismus verschiedene Konzepte. Virtuelle Funktionen haben *dieselbe* Schnittstelle in allen abgeleiteten Klassen, andernfalls würde man sie nicht brauchen.

7.6.1 Virtuelle Funktionen

Möglicherweise tritt der Fall ein, dass erst *zur Laufzeit* entschieden werden soll, welches Objekt angesprochen wird. Damit wird auch erst zur Laufzeit bestimmt, welche (Element-) Funktion verwendet werden soll, wie wir schon in Abschnitt 5.9 gesehen haben. In abgeleiteten Klassen können für solche Fälle *virtuelle* Funktionen der Basisklassen überladen werden.

Die überladenen Funktionen *müssen* in diesem Fall die *gleiche Signatur* haben, also den gleichen Namen und eine übereinstimmende Parameterliste, ansonsten werden sie wie normale überladene Funktionen aufgefasst. Der *Rückgabety*p einer virtuellen Funktion in einer abgeleiteten Klasse muss mit dem Rückgabety

in der Basisklasse *übereinstimmen* (Spezialisierungen sind dabei möglich, siehe unten).

Die Deklaration einer Funktion als `virtual` bewirkt, dass Objekten indirekt die Information über den Objekttyp mitgegeben wird. Dies wird ohne Zutun des Programmiers realisiert, indem im Speicherbereich eines Objekts zusätzlich zu den Objektattributen ein Zeiger *vp*tr auf eine besondere Tabelle *vtbl* (virtual table = Tabelle von Zeigern auf virtuelle Funktionen) eingebaut wird. Die Tabelle gehört zu der Klasse des Objekts und enthält ihrerseits Zeiger auf die virtuellen Funktionen dieser Klasse.

Wenn nun eine virtuelle Funktion über einen Zeiger oder eine Referenz auf dieses Objekt angesprochen wird, weiß das Laufzeitsystem, dass die Funktion über den Zeiger *vp*tr in der Tabelle gesucht und angesprochen werden muss. Es wird damit die *zu diesem Objekt* gehörende Funktion aufgerufen. Wenn die Klasse dieses Objekts aber *keine* Funktion mit gleicher Signatur hat, wird die entsprechende Funktion der *Oberklasse* gesucht und aufgerufen.

Um den internen Mechanismus muss man sich nicht kümmern. Es genügt zu wissen, dass Objekte durch den versteckten Zeiger *vp*tr etwas größer werden und dass der Zugriff auf virtuelle Funktionen durch den Umweg über die Zeiger geringfügig länger dauert. Oft kann der Zugriff auf eine virtuelle Funktion bereits statisch aufgelöst werden, sodass der Compiler in der Lage ist, den Zugriff zu optimieren. Um den Unterschied zwischen virtu-

ellen und nicht-virtuellen Funktionen herauszuarbeiten, vergleiche ich beide Varianten anhand des bekannten Beispiels.

Verhalten einer nicht-virtuellen Funktion

Rufen wir uns die überschriebenen Funktionen `flaeche()` des obigen Beispiels in Erinnerung:

```
class GraphObj {
    // ....
    double flaeche() const { return 0.0; } // nicht virtuell
};
```

Die Fläche eines allgemeinen grafischen Objekts ist eigentlich nicht 0, sondern undefiniert. In Abschnitt 7.6.2 wird darauf eingegangen.

```
class Rechteck : public GraphObj {
    // ....
    double flaeche() const {           // nicht virtuell
        return static_cast<double>(dieHoehe) * dieBreite;
    }
};
```

Wir definieren ein grafisches Objekt `graphObj`, ein Rechteck `R` und einen Zeiger `graphObjPtr`, den wir auf `graphObj` zeigen lassen:

```
GraphObj graphObj(Ort(20, 20));
Rechteck rechteck(Ort(100, 100), 20, 50); // (x, y), Höhe, Breite
GraphObj *graphObjPtr;                    // Zeiger auf graphObj
```

Nun wird die Fläche beider Objekte ausgegeben. Dazu wird der Zeiger zuerst auf das grafische Objekt `graphObj` gerichtet. Über `graphObjPtr` wird die Funktion `flaeche()` aufgerufen. Dann (zur Programmlaufzeit!) wird `graphObjPtr` auf das Rechteck `rechteck` gerichtet und der Aufruf wiederholt. Zum Vergleich wird `rechteck.flaeche()` angezeigt:

```
graphObjPtr = &graphObj; // Zeiger auf graphObj richten
cout << "graphObjPtr->flaeche() =" << graphObjPtr->flaeche() << endl;
graphObjPtr = &rechteck; // Zeiger auf Rechteck richten
cout << "graphObjPtr->flaeche() =" << graphObjPtr->flaeche() << endl;
cout << "rechteck.flaeche()      =" << rechteck.flaeche() << endl;
```

Was geschieht? Zweimal gibt es den Wert 0 und nur im dritten Aufruf den korrekten Wert 1000, obwohl `graphObjPtr` auf das Rechteck zeigt. Weil der Zeiger `graphObjPtr` vom Typ »Zeiger auf `GraphObj`« ist und keine Information über das Objekt hat, auf das er verweist, wird im ersten *und im zweiten* Fall `GraphObj::flaeche()` aufgerufen. Im zweiten Fall wird das anonyme Subobjekt vom Typ `GraphObj` angesprochen, das innerhalb des `rechteck`-Objekts liegt.

Verhalten einer virtuellen Funktion

Der Einsatz virtueller Funktionen bewirkt, dass Objekten die Typinformation über sich mitgegeben wird. Um das zu zeigen, erweitern wir das Beispiel um eine *virtuelle* Funktion `v_flaeche()`:

```

class GraphObj {
    // ...
    virtual double v_flaeche() const { return 0.0; }
};

class Rechteck : public GraphObj {
    // ...
    virtual double v_flaeche() const {
        return double(hoehe) * breite;
    }
};

```

Virtuelle Funktionen sind auch in allen nachfolgend abgeleiteten Klassen virtuell. Das Schlüsselwort `virtual` muss nur in der Basisklasse angegeben werden. Zu Dokumentationszwecken sollte es aber besser jeweils hingeschrieben werden. Das obige Beispiel wird jetzt mit der Funktion `v_flaeche()` in genau der gleichen Art und Weise wiederholt:

```

graphObjPtr = &graphObj;           // Zeiger auf graphObj richten
cout << "graphObjPtr->v_flaeche() =" << graphObjPtr->v_flaeche() << endl;
graphObjPtr = &rechteck;           // Zeiger auf Rechteck richten
cout << "graphObjPtr->v_flaeche() =" << graphObjPtr->v_flaeche() << endl;

```

Jetzt erhalten wir als Ergebnis 0 im ersten Fall (wie vorher), aber 1000 *im zweiten Fall*. Zur Laufzeit des Programms wird der Zeiger auf verschiedene Objekte gerichtet, und es wird die zum jeweiligen Objekt passende Funktion aufgerufen, nämlich im zweiten Fall `Rechteck::v_flaeche()`.

Ein Unterschied im Verhalten eines Objekts durch Aufruf einer virtuellen Funktion im Vergleich zu nichtvirtuellen Funktionen zeigt sich nur, wenn der Aufruf durch *Oberklassenzeiger oder -referenzen* geschieht statt über den Objektnamen. Im letzteren Fall gibt es ja ohnehin keine Zweifel über den Typ.

Welche Folgen hätte es, wenn die Forderung nach der gleichen Signatur nicht eingehalten wird? Dazu nehmen wir an, dass die Deklaration in der Klasse `Rechteck` einen Parameter `int` enthält (die Definition entsprechend; die Bedeutung des Parameters ist beliebig und spielt hier keine Rolle):

```
double v_flaeche(int z); // Fehler
```

- Es gibt nun keine Funktion `Rechteck::v_flaeche()` mehr mit einer passenden Signatur, sodass der Aufruf `graphObjPtr->v_flaeche()` im Gegensatz zum obigen Beispiel als `graphObjPtr->GraphObj::v_flaeche()` interpretiert wird und daher 0 ergibt – zu wenig für ein 20*50 Rechteck! Weil für das Objekt keine passende Funktion vorhanden ist, wird zur Oberklasse »durchgegriffen«. Im Falle mehrerer Vererbungsebenen wird die Hierarchie in Richtung der Basisklasse so lange durchsucht, bis eine passende Funktion gefunden wird. Dieser Vorgang ist durch den Einsatz interner Tabellen sehr schnell.
- `rechteck.v_flaeche()` (ohne `int`-Parameter) wäre nicht mehr möglich, weil die Basis-klassenfunktion `GraphObj::v_flaeche()` von `rechteck` nicht mehr zugreifbar ist. `rechteck.v_flaeche(99)` wäre zulässig.

- `graphObjPtr->v_flaeche(100)` wäre nicht möglich, weil `v_flaeche(int)` *keine* virtuelle Funktion ist und damit ausschließlich zur Klasse Rechteck gehört und nicht zu einem `graphObjPtr` als »Zeiger auf `GraphObj`« passt.

Eigenschaften virtueller Funktionen

Als wesentliche Merkmale virtueller Funktionen lassen sich zusammenfassen:

- Virtuelle Funktionen dienen zum Überladen bei gleicher Signatur und bei gleichem Rückgabotyp. Erlaubte Erweiterung: Wenn der Rückgabotyp einer virtuellen Funktion eine Referenz auf eine Klasse ist, dann darf der Rückgabotyp der entsprechenden Funktion in der abgeleiteten Klasse eine Referenz auf die abgeleitete Klasse sein. Das Gleiche gilt für Zeiger anstelle von Referenzen.
- Der Aufruf einer nicht-virtuellen Elementfunktion hängt vom Typ des Zeigers ab, über den die Funktion aufgerufen wird, während der Aufruf einer virtuellen Elementfunktion *vom Typ des Objekts* abhängt, auf das der Zeiger verweist. Der Aufruf von virtuellen Funktionen über Basisklassenzeiger oder -referenzen, die auf ein Objekt einer abgeleiteten Klasse zeigen, bezieht sich auf die *genau zu diesem Objekt* passende Funktion.
- Eine in einer Basisklasse als `virtual` deklarierte Funktion definiert eine Schnittstelle für alle abgeleiteten Klassen, auch wenn diese zum Zeitpunkt der Festlegung der Basisklasse noch unbekannt sind. Ein Programm, das Zeiger oder Referenzen auf die Basisklasse benutzt, kann damit sehr leicht um abgeleitete Klassen erweitert werden, weil der Aufruf einer virtuellen Funktion über Zeiger oder Referenzen sicherstellt, dass die zum referenzierten Objekt gehörende Realisierung der Funktion aufgerufen wird (siehe weiteres Beispiel in Abschnitt 7.6.2).
- Der vorstehende Punkt gilt auch für Destruktoren. Wenn es überhaupt virtuelle Funktionen in einer Klasse gibt, sollte der Destruktor als `virtual` deklariert werden. Die Definition der Klasse `GraphObj` muss um die Zeile

```
virtual ~GraphObj() {}
```

erweitert werden. Einzelheiten folgen ab Seite 280.

Aus diesen Punkten lässt sich eine wichtige Regel ableiten: Nicht-virtuelle Funktionen einer Basisklasse sollen *nicht* in abgeleiteten Klassen überschrieben werden! Oder anders ausgedrückt: Wenn ein Überschreiben notwendig erscheint, sollte die Funktion in der Basisklasse als `virtual` deklariert werden. Der Grund liegt darin, dass die Bedeutung (= das Verhalten) eines Programms sich nicht ändern sollte, wenn auf eine Methode über den Objektnamen oder über Basisklassenzeiger bzw. -referenzen zugegriffen wird.

Das folgende Programm ist ein erweitertes und erläutertes Beispiel aus [ES]. Es zeigt in konzentrierter Form die Eigenschaften virtueller Funktionen. Zum besseren Verständnis sollten wir uns daran erinnern, dass ein Name (englisch *identifier*), der in einem Gültigkeitsbereich (*scope*) definiert wird, alle außerhalb dieses Bereichs getroffenen Definitionen desselben Namens überdeckt. Diese Regel gilt unabhängig von der `virtual`-Eigenschaft von Funktionen. Namen in einer Basisklasse sind in einem äußeren Gültigkeitsbereich relativ zu Namen in einer abgeleiteten Klasse.

```

class Basisklasse {
public:
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    virtual Basisklasse* vf4();
    virtual Basisklasse& vf5();
    void f();
};

class AbgeleiteteKlasse : public Basisklasse {
public:
    virtual void vf1();
    virtual void vf2(int);
    virtual char vf3();           // Fehler! falscher Rückgabotyp
    virtual AbgeleiteteKlasse* vf4(); // geänderter Rückgabotyp
    virtual AbgeleiteteKlasse& vf5(); // geänderter Rückgabotyp
    void f();
};

int main () {
    AbgeleiteteKlasse d;
    Basisklasse *bp = &d;
    // In der folgenden Anweisung wird richtig AbgeleiteteKlasse::vf1()
    // aufgerufen, weil vf1() virtuell ist.
    bp->vf1();           // AbgeleiteteKlasse::vf1()
    // Eine Funktion AbgeleiteteKlasse::vf2(), das heißt ohne Parameter, gibt
    // es nicht. Deshalb wird durch bp->vf2(); die Funktion Basisklasse::vf2()
    // aufgerufen.
    bp->vf2();           // Basisklasse::vf2()

    // Die Funktion Basisklasse::vf2() ist von d aus nicht mehr zugreifbar. Mit
    // dem int-Parameter gibt es kein Problem, weil vf2(int) in der abgeleiteten
    // Klasse deklariert ist.
    d.vf2();             // Fehler!
    d.vf2(7);            // ok

    // Obwohl bp auf ein Objekt der abgeleiteten Klasse zeigt, ist bp->vf2(7) nicht
    // möglich, weil eine Funktion mit int-Parameter in der Basisklasse nicht existiert.
    bp->vf2(7);           // Fehler!

    // bp->f() ruft Basisklasse::f() für das in d enthaltene Subobjekt auf, weil
    // f() nicht virtuell ist.
    bp->f();

    AbgeleiteteKlasse* dp;
    dp = d.vf4();         // AbgeleiteteKlasse::vf4()
    d.vf5();              // AbgeleiteteKlasse::vf5()
    d.vf5().vf1();
    // Eine Referenz kann als Alias-Name für ein Objekt aufgefasst werden.
    // Weil d.vf5() eine Referenz auf AbgeleiteteKlasse
    // zurückgibt, wird der Aufruf der Funktion d.vf5().vf1()
    // interpretiert als (d.vf5()).vf1(). Typischerweise wird das

```

```
// (möglicherweise veränderte) Objekt selbst als Referenz zurückgegeben.  
// Die Zeile kann dann in zwei Teile zerlegt werden:  
// d.vf5();  
// d.vf1();  
} // Ende von main
```

7.6.2 Abstrakte Klassen

In vielen Fällen sollte die Basisklasse einer Hierarchie sehr allgemein sein und Code enthalten, der aller Voraussicht nach nicht geändert werden muss. Es ist dann oft nicht notwendig oder gewünscht, dass Objekte dieser Klassen angelegt werden. Diese *abstrakten Klassen* dienen ausschließlich als *Ober- oder Basisklassen*. Objekte werden nur von den abgeleiteten Klassen erzeugt, die dann jeweils ein Subobjekt vom Typ der abstrakten Basisklasse enthalten. Das syntaktische Mittel, um eine Klasse abstrakt zu machen, sind *rein virtuelle Funktionen* (englisch *pure virtual*). Abstrakte Klassen haben mindestens eine rein virtuelle Funktion, die typischerweise *keinen* Definitionsteil hat, aber einen haben kann. Durch die rein virtuelle Funktion wird gewährleistet, dass stets die zum Objekttyp passende Methode aufgerufen wird. Definieren einer abstrakten Klasse heißt also nichts anderes, als ein gemeinsames Protokoll für alle abgeleiteten Klassen zu definieren. Eine rein virtuelle Funktion wird durch Ergänzung von `= 0` deklariert:

```
virtual int rein_virtuelle_func(int) = 0;
```

Unser Beispiel mit den grafischen Objekten ist wie geschaffen zur Anwendung abstrakter Klassen, denn ein grafisches Objekt ist entweder ein Rechteck, ein Polygon, ein Kreis oder was man sich sonst noch ausdenken kann, aber niemals ein grafisches Objekt »an sich«. Ein *allgemeines* grafisches Objekt kann *nicht* gezeichnet werden und hat keine definierte Fläche. Also benötigen wir in einem Programm *keine* Objekte der Klasse `GraphObj`, außer natürlich als (versteckte) Subobjekte von Rechtecken, Kreisen und so weiter. Wir können die Klasse `GraphObj` daher als abstrakte Klasse formulieren, indem wir `flaeche()` in eine rein virtuelle Funktion umwandeln:

```
virtual double flaeche() const = 0;
```

Klassen, von denen Objekte erzeugt werden können, nennt man konkrete Klassen, wenn der Unterschied zu abstrakten Klassen betont werden soll. Wenn eine konkrete Klasse von einer abstrakten Klasse erbt, muss sie zu den rein virtuellen vorgegebenen Funktionsprototypen konkrete Implementierungen bereitstellen, zum Beispiel um die Fläche als Produkt von Höhe mal Breite zu berechnen.

Wenn in einer vermeintlich konkreten Klasse eine Implementierung fehlt, zum Beispiel, weil sie vergessen wurde, ist sie tatsächlich nicht konkret, sondern selbst abstrakt. Die Eigenschaft »abstrakt« wird auf Klassen ohne oder mit unvollständiger Implementation vererbt. Falls versucht wird, von einer Klasse dieser Art ein Objekt zu erzeugen, gibt es eine Fehlermeldung des Compilers.

Das unten stehende Beispiel zeigt eine typische Art, abstrakte Klassen und virtuelle Funktionen einzusetzen. Wir erweitern dazu die Klasse `GraphObj` um eine Funktion `zeichnen()`, die das Objekt auf dem Bildschirm darstellen soll. Die Funktion sieht natürlich für Kreise und Rechtecke unterschiedlich aus, der Aufruf jedoch beziehungsweise die Schnittstelle

ist stets die gleiche. Um das Beispiel nicht mit graphikspezifischen Details zu überfrachten, besteht die einzige Aufgabe der Funktion `zeichnen()` darin, eine Meldung auf dem Bildschirm auszugeben.

Weitere Besonderheiten des Beispiels sind wie folgt:

- Die Methode `flaeche()` ist in der Klasse `GraphObj` als rein virtuelle Funktion ohne Definition deklariert.
- Im Unterschied dazu stellt die ebenfalls rein virtuelle Methode `zeichnen()` eine Standarddefinition bereit, die von den abgeleiteten Klassen benutzt wird.
- Die Klasse `Quadrat`² braucht die Funktion `flaeche()` nicht neu zu implementieren, weil die Implementierung von der Klasse `Rechteck` geerbt wird. Dies gilt auch für `zeichnen()`, wenn auf eine Unterscheidung bei der Ausgabe verzichtet werden soll.
- Die `while`-Schleife im Main-Programm zeigt die Stärke des Polymorphismus. Ohne dass man sich um den Typ der einzelnen Objekte kümmern muss, wird stets die richtige Funktion aufgerufen.

Der Übersichtlichkeit halber und weil später Bezug darauf genommen wird, sind die Dateien mit den Änderungen vollständig wiedergegeben.

Listing 7.5: Klasse `GraphObj`, 2. Version

```
// cppbuch/k7/abstrakt/graphobj.h
#ifndef GRAPHOBJ_H
#define GRAPHOBJ_H
#include "ort.h" // enthält #include<iostream>

class GraphObj {                                // Version 2
public:
    GraphObj(const Ort& einOrt)                  // allg. Konstruktor
        : referenzkoordinaten(einOrt) {}
    virtual ~GraphObj() {}                      // virtueller Destruktor

    const Ort& bezugspunkt() const { // Bezugspunkt ermitteln
        return referenzkoordinaten;
    }
    // alten Bezugspunkt ermitteln und gleichzeitig neuen wählen
    Ort bezugspunkt(const Ort& n0) {
        Ort temp = referenzkoordinaten;
        referenzkoordinaten = n0;
        return temp;
    }
    // Koordinatenabfrage
    int getX() const { return referenzkoordinaten.getX(); }
    int getY() const { return referenzkoordinaten.getY(); }
    // rein virtuelle Methoden
    virtual double flaeche() const = 0;
    virtual void zeichnen() const = 0;
private:
    Ort referenzkoordinaten;
};
```

² Zur Diskussion, ob ein Quadrat ein Rechteck im Sinn der objektorientierten Programmierung ist, siehe unten Seite 282.

```
// Die Standardimplementierung einer rein virtuellen Methode
// muss außerhalb der Klassendefinition stehen:
inline void GraphObj::zeichnen() const {
    std::cout << "Zeichnen: ";
}
// Die Entfernung zwischen zwei GraphObj-Objekten ist hier als Entfernung ihrer
// Bezugspunkte (überladene Funktion) definiert.
inline double entfernung(const GraphObj& g1,
                        const GraphObj& g2) {
    return entfernung(g1.bezugspunkt(), g2.bezugspunkt());
}
#endif // GRAPHOBJ_H
```

Die Klassen *Strecke* und *Rechteck* müssen die rein virtuellen Methoden implementieren. Andernfalls wären die Klassen ebenfalls abstrakt, und es könnte keine Instanzen von ihnen geben. Ein Endpunkt der *Strecke* wird von *GraphObj* geerbt, der andere ist Attribut der Klasse.

Listing 7.6: Klasse *Strecke*

```
// cppbuch/k7/abstrakt/strecke.h
#ifndef STRECKE_H
#define STRECKE_H
#include "graphobj.h"

class Strecke : public GraphObj { // erben von GraphObj
public:
    // Initialisierung von Subobjekt und Attribut mit Initialisierungsliste
    Strecke(const Ort& ort1, const Ort& ort2)
        : GraphObj(ort1), endpunkt(ort2) {
    }

    double laenge() const {
        return entfernung(bezugspunkt(), endpunkt);
    }

    // Definition der rein virtuellen Methoden
    virtual double flaeche() const {
        return 0.0;
    }

    virtual void zeichnen() const {
        GraphObj::zeichnen();
        std::cout << "Strecke von ";
        anzeigen(bezugspunkt());
        std::cout << " bis ";
        anzeigen(endpunkt());
        std::cout << std::endl;
    }
private:
    Ort endpunkt; // zusätzlich: 2. Punkt der Strecke
};
#endif // STRECKE_H
```

Listing 7.7: Klasse Rechteck

```
// cppbuch/k7/abstrakt/rechteck.h
#ifndef RECHTECK_H
#define RECHTECK_H
#include "graphobj.h"

class Rechteck : public GraphObj { // von GraphObj erben
public:
    Rechteck(const Ort& ort, int h, int b)
        : GraphObj(ort), dieHoehe(h), dieBreite(b) {}

    // wird von Quadrat benötigt
    int hoehe() const {
        return dieHoehe;
    }
    int breite() const {
        return dieBreite;
    }

    // Definition der rein virtuellen Methoden
    virtual double flaeche() const {
        return static_cast<double>(dieHoehe) * dieBreite;
    }

    virtual void zeichnen() const {
        GraphObj::zeichnen();
        std::cout << "Rechteck (h x b = " << dieHoehe << " x "
                    << dieBreite << ") an der Stelle ";
        anzeigen(bezugspunkt());
        std::cout << std::endl;
    }
private:
    int dieHoehe, dieBreite;
};
#endif
```

Listing 7.8: Klasse Quadrat

```
// cppbuch/k7/abstrakt/quadrat.h
#ifndef QUADRAT_H
#define QUADRAT_H
#include "rechteck.h"

class Quadrat : public Rechteck { // siehe Text
public:
    Quadrat(const Ort& ort, int seite)
        : Rechteck(ort, seite, seite) {}

    // Definition der rein virtuellen Methoden
    virtual void zeichnen() const {
        GraphObj::zeichnen();
        std::cout << "Quadrat (Seitenlaenge = " << hoehe()
```

```

        << ") an der Stelle ";
        anzeigen(bezugspunkt());
        std::cout << std::endl;
    }
    // Die Methoden bezugspunkt(), flaeche(), hoehe(), breite() werden geerbt.
};
#endif // QUADRAT_H

```

Das folgende Beispielprogramm ruft die Methoden der grafischen Objekte polymorph auf. Entscheidend ist nicht der (statische) Typ des Zeigers, den der Compiler sieht, sondern der polymorphe oder dynamische Typ, das heißt, der Typ des Objektes, auf das der Zeiger zur Laufzeit verweist. Die Elemente des Feldes `GraphObjZeiger` sind alle vom statischen Typ `GraphObj*`, sie verweisen aber zur Laufzeit auf Objekte von Klassen, die von `GraphObj` abgeleitet wurden.

Dasselbe gilt für Referenzen. So sind die Referenzen `R_Ref`, `S_Ref` und `Q_Ref` im Programm alle vom Typ der Basisklasse `GraphObj`. Die Referenzen verweisen aber zur Laufzeit auf Objekte verschiedener Typen, nämlich der Klassen `Rechteck`, `Strecke` und `Quadrat`.

Das Beispiel ist sehr leicht um beliebige grafische Klassen erweiterbar (zum Beispiel `Kreis`, `Ellipse`, `Polygon` ...), ohne dass die Anweisung »Zeichnen aller Objekte« überhaupt geändert werden muss.

Listing 7.9: Anwendung von Polymorphismus

```

// cppbuch/k7/abstrakt/main.cpp
#include "strecke.h"
#include "quadrat.h" // schließt rechteck.h ein

int main() {
    // GraphObj g; Fehler! Instanzen abstrakter Klassen gibt es nicht
    Rechteck r(Ort(0,0), 20, 50);
    Strecke s(Ort(1,20), Ort(200,0));
    Quadrat q(Ort(122, 99), 88);
    // C-Array mit Basisklassenzeigern, initialisiert mit
    // den Adressen der Objekte und 0 als Endekennung
    GraphObj* graphObjZeiger[] = {&r, &s, &q, 0};

    // Ausgabe der Fläche aller Objekte
    int i = 0;
    while(graphObjZeiger[i]) {
        std::cout << "Fläche = " << graphObjZeiger[i++]->flaeche() << std::endl;
    }
    // Zeichnen aller Objekte
    i = 0;
    while(graphObjZeiger[i]) {
        graphObjZeiger[i++]->zeichnen();
    }
    // Referenzen statt Zeiger
    std::cout << "Auch Referenzen sind polymorph:\n";
    GraphObj &r_ref = r, // Der statische Typ ist derselbe,
            &s_ref = s,
            &q_ref = q;
    r_ref.zeichnen(); // der dynamische nicht.

```

```

s_ref.zeichnen();
q_ref.zeichnen();
}

```

7.6.3 Virtueller Destruktor

Ein virtueller Destruktor sorgt ähnlich wie virtuelle Funktionen dafür, dass Zeigern die Typinformation über ein Objekt zur Verfügung steht und deshalb die Speicherfreigabe exakt erfolgt. Über einen Zeiger `px` vom Typ »Zeiger auf Basisklasse«, der auf ein Objekt `x` einer abgeleiteten Klasse zeigt, kann zur Compilerzeit, also statisch, nur die Größe des Subobjekts (vom Typ Basisklasse) von `x` ermittelt werden. Die Operation `delete` auf `px` angewendet, gäbe ohne virtuellen Destruktor Platz entsprechend `sizeof(*px)` frei, also zu wenig, sodass langlaufende Programme Speicherprobleme bekommen können. Interessant ist hier aber der *dynamische* Typ, also der Typ des Objekts `x`, denn für diesen Typ muss der Speicherplatz freigegeben werden. Das Beispielprogramm demonstriert die Notwendigkeit für virtuelle Destruktoren.

Listing 7.10: Beispielprogramm mit virtuellem Destruktor

```

// cppbuch/k7/virtdest.cpp
#include<iostream>
using namespace std;
#define PRINT(X) cout << (#X) << " = " << (X) << endl

class Basis {
    int bWert;
public:
    Basis(int b = 0)
        : bWert(b) {}
    virtual ~Basis() { // virtueller Destruktor!
        cout << "Objekt " << bWert << " Basis-Destruktor aufgerufen!\n";
    }
};

class Abgeleitet : public Basis {
    double aWert;
public:
    Abgeleitet(int b = 0, double a = 0.0)
        : Basis(b), aWert(a) {}
    ~Abgeleitet() {
        cout << "Objekt " << aWert << " Abgeleitet-Destruktor aufgerufen!\n";
    }
};

int main () {
    Basis *pb = new Basis(1);
    PRINT(sizeof(*pb));
    Abgeleitet *pa = new Abgeleitet(2, 2.2);
    PRINT(sizeof(*pa));
    Basis *pba = new Abgeleitet(3, 3.3);
    PRINT(sizeof(*pba));
    cout << "pb löschen:\n";
    delete pb; // ok
}

```

```

cout << "pa löschen:\n";
delete pa;                      // ok
cout << "pba löschen:\n";
delete pba;                     // ok nur mit virtuellem Destruktor!
}

```

Das Makro PRINT ist auf Seite 132 erklärt. Die Basisklassenobjekte werden durch eine ganze Zahl, die Objekte der abgeleiteten Klasse durch eine Zahl des Typs `double` identifiziert. Es werden ein Basisklassenobjekt und zwei Objekte der abgeleiteten Klasse erzeugt. Im Beispielpogramm werden 4 Bytes für `int`, 8 Bytes für `double` und 4 Bytes für den versteckten Zeiger *vptr* (Seite 270) benötigt. Es liefert die Ausgabe (die Zahlen können auf Ihrem System andere sein):

```

sizeof(*pb) = 8
sizeof(*pa) = 16
sizeof(*pba) = 8
pb löschen:
Objekt 1 Basis-Destruktor aufgerufen!
pa löschen:
Objekt 2.2 Abgeleitet-Destruktor aufgerufen!
Objekt 2 Basis-Destruktor aufgerufen!
pba löschen:
Objekt 3.3 Abgeleitet-Destruktor aufgerufen!
Objekt 3 Basis-Destruktor aufgerufen!

```

`sizeof` gibt die statisch aus dem Typ des Zeigers ermittelbare Objektgröße an. `delete` ruft den korrekten Destruktor auch im letzten Fall auf. *Ohne* das Schlüsselwort `virtual` würde nur jeweils der Destruktor aufgerufen, der zum Typ des Zeigers passt. Ausgabe bei *Fehlen* des Schlüsselworts `virtual`:

```

sizeof(*pb) = 4                      veränderte Werte!
sizeof(*pa) = 12
sizeof(*pba) = 4

```

... und so weiter wie oben, aber es *fehlt* die Ausgabe

```
Objekt 3.3 Abgeleitet-Destruktor aufgerufen!
```

Man sieht daran, dass nur der Basisklassenanteil des Objektes `*pba` freigegeben wurde, entsprechend dem statischen Datentyp von `pba`. Der Rest bleibt im Speicher hängen.



Merke:

Virtuelle Destruktoren sollten immer dann verwendet werden, wenn von der betreffenden Klasse abgeleitet wird oder nicht auszuschließen ist, dass von ihr zukünftig durch Ableitung neue Klassen gebildet werden.

An den nun ausgegebenen, veränderten `sizeof`-Werten ist ferner zu erkennen, dass die Objekte nunmehr *keine* besondere Typinformation enthalten, das heißt in diesem Fall, dass die Tabelle der Zeiger auf virtuelle Funktionen (siehe Seite 270) nicht existiert. Der Effekt ist hier mit `sizeof` natürlich nur deshalb erkennbar, weil es keine weitere virtuelle Funktion gibt (die die Objektgröße verändern würde).

Immer wenn Basisklassenzeiger oder -referenzen auf dynamisch erzeugte Objekte benutzt werden, was normalerweise im Zusammenhang mit der Benutzung virtueller Methoden steht, sollte ein virtueller Destruktor eingesetzt werden. Wenn eine Klasse von anderen per Vererbung genutzt werden kann, kann die Art der zukünftigen Benutzung nicht bekannt sein. Also: Destruktoren immer virtuell machen, falls vererbt werden könnte!



Übung

7.1 Auf Seite 268 wurde die Funktion `fLaeche()` für ein Objekt der Klasse `Strecke` aufgerufen. Ist der Aufruf auch möglich, wenn `GraphObj` als *abstrakte* Klasse definiert ist?

7.7 Probleme der Modellierung mit Vererbung

Dass Vererbung die geeignete programmiertechnische Umsetzung einer *ist-ein-* oder *ist-eine-Art-*Beziehung zwischen Objekten ist, kann durchaus fraglich sein. Das Für und Wider wird hier anhand einiger Grenzfälle diskutiert.

Eine abgeleitete Klasse kann als Subtyp der Oberklasse aufgefasst werden. *Ein Objekt einer abgeleiteten Klasse kann damit stets an die Stelle eines Objekts der Oberklasse treten* – es sind ja alle Methoden der Oberklasse vorhanden, wenn auch möglicherweise überschrieben (Liskovsches Substitutionsprinzip, siehe [Lis]). Dies erscheint auf den ersten Blick einleuchtend. Dennoch gibt es Fälle, in denen dieser Satz der Konvention oder der menschlichen Erfahrung widerspricht. Ein einfaches Beispiel soll dies erläutern.

Seit Euklid, also seit mehr als 2000 Jahren, ist bekannt, dass ein Quadrat ein Rechteck und ein Kreis eine Ellipse ist. Genauer formuliert, ist ein Quadrat ein Rechteck mit gleichen Seitenlängen, also ein Spezialfall eines Rechtecks. Die Spezialisierung wird in C++ durch *public*-Vererbung ausgedrückt:

```
class Quadrat : public Rechteck { ...};
```

Nun kann man sich aber eine Klasse `Rechteck` vorstellen, die es erlaubt, die Seiten ungleichmäßig zu ändern; denken wir nur an einen grafischen Editor, mit dem ein Rechteck in verschiedene Richtungen auseinandergezogen werden kann:

```
class Rechteck {
public:
    virtual void hoeheAendern(int neu) { hoehe = neu;}
    virtual void breiteAendern(int neu) { breite = neu;}
    // ...
private:
    int hoehe;
    int breite;
};
```

Vordergründig ist klar, dass diese Methoden in einer Klasse `Quadrat` nichts zu suchen haben, wenn die Forderung aufrechterhalten bleiben soll, dass ein `Quadrat`-Objekt stets an die Stelle eines `Rechteck`-Objekts treten kann. Die manchmal empfohlene »Lösung«, dass zwischen `Quadrat` und `Rechteck` gar keine Vererbungsbeziehung besteht und beide Klassen von einer abstrakten Klasse `Viereck` erben sollten, ist nicht sinnvoll, weil das Problem nur auf eine andere Ebene verschoben wird: Ein allgemeines `Viereck` kann man diagonal zu einer Raute verformen, ein `Rechteck` nicht, wenn es eines bleiben soll. Um solche Fälle vernünftig darstellen zu können, wird Vererbung gelegentlich benutzt, um *Einschränkungen* (englisch *constraints*) einer Oberklasse zu formulieren (*inheritance for restriction*, siehe nachfolgendes Beispiel). Dennoch sollte man sorgfältig überlegen, ob es nicht andere Wege gibt.

Ein großer Vorteil der Objektorientierung besteht darin, dass die Begriffe der Anwendung weit mehr als in nicht-objektorientierten Programmiersprachen durchgängig von der Analyse zum Code benutzbar sind. Davon sollte man nicht ohne schwerwiegenden Grund abweichen – den es durchaus geben kann. Die Beziehung »ein `Quadrat` ist ein `Rechteck`« ist die natürliche Beziehung in einer mathematisch-geometrischen Anwendung, die beibehalten werden sollte. Nur vom Standpunkt der Implementierung her sollte man sich überlegen, ob der zusätzliche Aufwand in Kauf genommen werden soll, Platz für zwei Seitenlängen zu spendieren, obwohl nur eine nötig ist.

In der objektorientierten Programmierung geht es unter anderem um einen *Vertrag* mit dem Benutzer einer Klasse. Der Benutzer muss sich darauf verlassen können, dass die Klasse den Vertrag einhält, das heißt, dass die Seitenlängen im `Quadrat` untereinander stets gleich bleiben.

Wenn die Klasse `Quadrat` von der Klasse `Rechteck` erben soll, lässt sich das Einhalten der Bedingung gleicher Seitenlängen leicht bewerkstelligen:

```
class Quadrat : public Rechteck { // empfehlenswert?
public:
    // ... (Konstruktor usw. weggelassen)
    virtual void hoeheAendern(int neu) {
        Rechteck::hoeheAendern(neu);
        Rechteck::breiteAendern(neu);
    }
    virtual void breiteAendern(int neu) {
        hoeheAendern(neu);
    }
};
```

Die vertragliche Einschränkung, dass Höhe und Breite eines Quadrats stets gleich sind, wird an alle von `Quadrat` abgeleiteten Klassen vererbt. Eine Möglichkeit, ohne Vererbung auszukommen und ohne auf die Funktionen eines `Rechtecks` zu verzichten, soweit sie angemessen sind, zeigt das folgende Beispiel, in dem ein `Quadrat` ein `Rechteck` *benutzt*:

```
class Quadrat { // empfehlenswert?
public:
    Quadrat(const Ort& ort, int seite)
        : r(ort, seite, seite) { // privates Rechteck initialisieren
    }
    virtual void seiteAendern(int neu) {
        r.hoeheAendern(neu);
    }
};
```



```

        r.breiteAendern(neu);
    }
    // ... viele weitere Funktionen, die Methoden der Klasse Rechteck benutzen
private:
    Rechteck r;
};

```

Die Methoden des Rechtecks sind für Quadratbenutzer nicht mehr zugreifbar, aber die Klasse `Quadrat` macht sich die Methoden zunutze, indem es die Aufgaben an das Rechteck `r` *delegiert*. Der Nachteil dieser Lösung besteht darin, dass `Quadrat` und `Rechteck` nicht weiterhin polymorph benutzbar sind. Wenn man `Quadrat` von der Klasse `GraphObj` erben ließe, hätte man das Problem, dass der Bezugspunkt doppelt angelegt wäre: im anonymen Subobjekt und im privaten Rechteck-Objekt. Falls `Quadrat` nur wenige Funktionen von `Rechteck` benutzt, der Aspekt der Wiederverwendung von Code also keine große Rolle spielt, ist es besser, `Quadrat` als eigenständige Klasse zu implementieren, die von `GraphObj` erbt. Problemstellungen dieser Art kommen gelegentlich vor. Ein weiteres Beispiel: Eine sortierte Liste ist doch sicherlich auch eine Liste – oder? Bei näherer Betrachtung stellt man fest, dass die sortierte Reihenfolge zerstört werden kann. Die Operation, ein beliebiges Element am Anfang einer Liste einzufügen, darf nicht für eine sortierte Liste gelten.

Die Ursache für das Dilemma liegt im Verständnis des Begriffs Spezialisierung bzw. der *ist-ein*-Relation. Mit der `public`-Vererbung ist stets eine Spezialisierung der Schnittstellen oder eine Erweiterung gemeint, in der Mathematik oder in der Umgangssprache kann es aber auch eine *Einschränkung* oder *Verminderung* der Schnittstellen bedeuten.

Nur wenn ein Objekt einer abgeleiteten Klasse jederzeit an die Stelle eines Basisklassenobjekts treten kann, ist die `public`-Vererbung sinnvoll, und nur dann kann der Typ der abgeleiteten Klasse als Subtyp der Basisklasse aufgefasst werden. Andernfalls ist die umgangssprachlich in der Modellierung benutzte *ist-ein*-Beziehung auf andere Art darzustellen. Damit kann `Quadrat` zwar von der oben beschriebenen Klasse `Rechteck` erben. Dies würde jedoch nicht mehr gelten, wenn die Klasse `Rechteck` eine weitere Methode `seitenverhaeltnisAendern()` hätte, weil sie vom `Quadrat` nicht ohne Verletzung des Vertrags realisiert werden kann. Die erwähnte sortierte Liste sollte nicht `public` von einer Listenklasse erben.



Übungen

7.2 Schreiben Sie eine Klasse `Person` mit den zwei Attributen `Nachname` und `Vorname`, sowie eine Klasse `StudentIn` und eine Klasse `ProfessorIn`, die beide von `Person` erben. Die Klasse `StudentIn` soll ein Attribut »Matrikelnummer«, die Klasse `ProfessorIn` ein Attribut »Lehrgebiet« haben. Der Einfachheit halber seien alle Attribute vom Typ `string`. Fügen Sie Methoden zum Lesen der Attribute hinzu, zum Beispiel `const string& getNachname()` bei der Klasse `Person`. Es soll auch eine Methode `toString()` geben, die die vollständigen Informationen liefert und deren Schnittstelle und eine Standardimplementierung in der Klasse `Person` definiert ist. Die Standardimplementierung soll einen aus Vor- und Nachnamen zusammengesetzten String zurückliefern, die in den Unterklassen zu redefinierenden Implementierungen auch den Status (`StudentIn/ProfessorIn`) und die Matrikelnummer bzw. das Lehrgebiet enthält. Von der Klasse `Person` soll kein Objekt erzeugt werden können, sie sei also abstrakt. Der folgende Programmauszug zeigt die Benutzung der Klassen:

```
vector<Person*> diePersonen;
diePersonen.push_back(
    new StudentIn("Risse", "Felicita", "635374"));
diePersonen.push_back(
    new ProfessorIn("Philippsen", "Nele", "Datenbanken"));
diePersonen.push_back(
    new StudentIn("Spillner", "Julian", "123429"));
for(size_t i = 0; i < diePersonen.size(); ++i) {
    cout << diePersonen[i]->getVorname() << endl;
}
for(size_t i = 0; i < diePersonen.size(); ++i) {
    cout << diePersonen[i]->toString() << endl;
}
```

Die Ausgabe des Programms sei z.B.:

Felicita

Nele

Julian

Student/in Felicita Risse, Mat.Nr.: 635374

Prof. Nele Philippsen, Lehrgebiet: Datenbanken

Student/in Julian Spillner, Mat.Nr.: 123429

7.3 Wie kann man im obigen Programmauszug auf eine Methode der Klasse `StudentIn` zugreifen, zum Beispiel auf die Methode `getMatrikelnummer()`?

7.8 Mehrfachvererbung³

Die Mehrfachvererbung gewährt eine große Flexibilität insbesondere bei der Systemmodellierung, wird jedoch nicht häufig benötigt – je nach Art der Problemstellung. Die Mehrfachvererbung bietet gegenüber der Einfachvererbung bessere Möglichkeiten, Objekte der realen Welt abzubilden.

Eine Klasse kann von *mehreren* Basisklassen erben, wie in Abbildung 7.1 auf Seite 258 zu sehen ist. Da hier *nur das Prinzip* der Mehrfachvererbung gezeigt werden soll, betrachten wir im Folgenden ein möglichst einfaches Beispiel, das als C++-Programm ausformuliert wird. Auf einem Graphikbildschirm sollen verschiedene Objekte dargestellt werden, hier ein Rechteck (Rechteck) und ein beschriftetes Rechteck (beschriftetesRechteck).

Ein beschriftetes Rechteck *ist ein* beschriftetes grafisches Objekt, und ein beschriftetes grafisches Objekt wiederum *ist ein* grafisches Objekt. Dieser Zusammenhang wird durch die in Abbildung 7.4 dargestellte Vererbungsstruktur gezeigt. Die Klasse `beschriftetesObjekt` ist wie `GraphObj` abstrakt, weil die in der letzteren Klasse deklarierte rein virtuelle Funktion `flaeche()` nicht in `beschriftetesObjekt` definiert ist und daher die Eigenschaft »abstrakt« geerbt wird.

³ Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

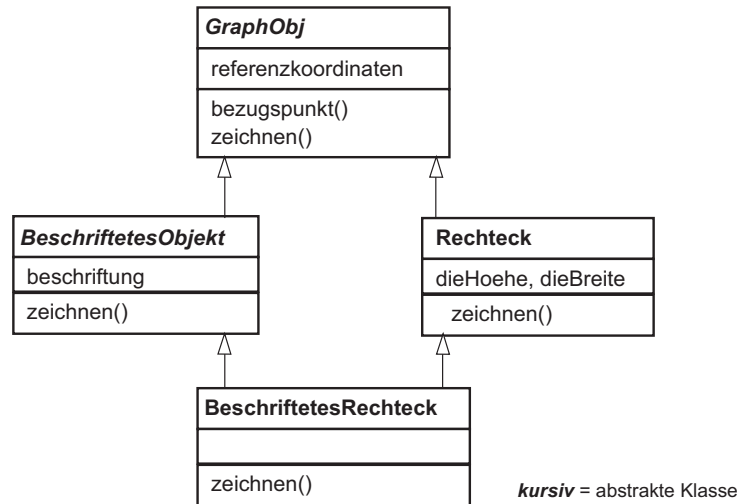


Abbildung 7.4: Vererbungsstruktur grafischer Objekte

Es ist im Allgemeinen nicht notwendig, dass von einer gemeinsamen Basisklasse geerbt wird. Hier wurde das Beispiel absichtlich so gewählt, weil mit einer gemeinsamen Basisklasse eine spezielle Problematik auftritt, die in Abschnitt 7.8.1 besprochen wird.

Alle grafischen Objekte haben bestimmte gemeinsame Eigenschaften. Zum Beispiel hat jedes Objekt einen bestimmten Ort auf dem Bildschirm, nämlich den Bezugspunkt *referenzkoordinaten*. Es folgen die Header-Dateien *.h mit den Deklarationen für *BeschriftetesObjekt* und *BeschriftetesRechteck*. Die anderen Deklarationen sind in Abschnitt 7.6.2 ab Seite 276 zu finden.

Listing 7.11: Klasse *BeschriftetesObjekt*

```
// cppbuch/k7/mehrfach/konflikt/beschriftetesobjekt.h
#ifndef BESCHRIF_H
#define BESCHRIF_H
#include "graphobj.h"
#include <string>

class BeschriftetesObjekt : public GraphObj { // erben
public:
    BeschriftetesObjekt(const Ort& ort, const std::string& b)
        : GraphObj(ort), beschriftung(b) {
    }

    virtual void zeichnen() const {
        GraphObj::zeichnen();
        std::cout << "Beschriftung bei ";
        anzeigen(bezugspunkt());
        std::cout << beschriftung << std::endl;
    }
private:
    std::string beschriftung;
};
```

```
};
#endif // BESCHRIF_H
```

Die Klasse `BeschriftetesObjekt` enthält ein Objekt `beschriftung` des Typs `string`. Der Einfachheit halber sind alle Methoden `inline`. Die Klasse `BeschriftetesObjekt` benötigt keinen Destruktor, weil der systemerzeugte Destruktor die Destruktoren für alle Elemente einer Klasse aufruft, sodass zum Beispiel der Destruktor von `string` den dynamisch bereitgestellten Platz für die Beschriftung freigibt.

Die zur tatsächlichen Ausgabe auf dem Bildschirm notwendigen Graphikfunktionen sind systemspezifisch, sodass hier nur eine schlichte Textausgabe auf dem Bildschirm erscheinen soll. Der Konstruktor ruft jeweils den Basisklassenkonstruktor zur Initialisierung auf. Auch ein `BeschriftetesRechteck` wird mit den Oberklassenkonstruktoren initialisiert, die ihrerseits den Basisklassenkonstruktor aufrufen. Die Funktion `zeichnen()` ruft die entsprechenden Methoden der Subobjekte auf.

Listing 7.12: Klasse `BeschriftetesRechteck`

```
// cppbuch/k7/mehrfach/konflikt/beschriftetesrechteck.h
#ifndef BES_R_H
#define BES_R_H
#include "beschriftetesobjekt.h"
#include "rechteck.h"

// Mehrfachvererbung
class BeschriftetesRechteck
: public BeschriftetesObjekt, public Rechteck {
public:
    BeschriftetesRechteck(const Ort& o, int h, int b,
                          const std::string& beschr)
: BeschriftetesObjekt(o, beschr),
  Rechteck(o, h, b) {
    }
    // Definition der rein virtuellen Methoden
    virtual double flaeche() const {
        // Definition ist notwendig, damit die Klasse nicht abstrakt ist (durch Vererbung über
        // BeschriftetesObjekt und GraphObj)
        return Rechteck::flaeche();
    }
    virtual void zeichnen() const {
        Rechteck::zeichnen();
        BeschriftetesObjekt::zeichnen();
    }
};
#endif // BES_R_H
```

In einem Hauptprogramm könnten nach diesen Definitionen Anweisungen folgender Art stehen:

```
// cppbuch/k7/mehrfach/konflikt/main.cpp
// Auszug:
Rechteck r(Ort(0,0), 20, 50);
BeschriftetesRechteck bR(Ort(1,20), 60, 60,
```

```

                                std::string("Mehrfachvererbung"));
r.zeichnen();
bR.zeichnen();
BeschriftetesRechteck *zBR = new BeschriftetesRechteck(
                                Ort(100,0), 20, 80,
                                std::string("dynamisches Rechteck"));
zBR->zeichnen();

```

Das Objekt `*zBR` muss mit `delete` gelöscht werden, weil es mit `new` erzeugt wurde. Die Anwendung von `delete` auf einen Zeiger ruft automatisch den Destruktor des referenzierten Objekts auf.

7.8.1 Namenskonflikte

Bei Mehrfachvererbung können Namenskonflikte und Mehrdeutigkeiten auftreten. Zum Beispiel könnte man versuchen, sich die Koordinaten der Objekte ausgeben zu lassen:

```

std::cout << "Rechteck-Position: ";
anzeigen(r.bezugspunkt());
std::cout << "beschriftetes-Rechteck-Position: ";
anzeigen(bR.bezugspunkt()); // Compiler-Fehlermeldung!

```

Vom Rechteck `r` würde der Bezugspunkt ausgegeben werden, die Ausgabe der Koordinaten des beschrifteten Rechtecks `bR` führt hingegen zu einer Fehlermeldung des Compilers. Warum? Der Aufruf ist zweideutig. Die Ursache liegt darin, dass `GraphObj` *zweimal* geerbt wurde. Der Compiler weiß nicht, ob er den Bezug zu `GraphObj::bezugspunkt()` über das in `BeschriftetesObjekt` oder das in `Rechteck` enthaltene Subobjekt vom Basisklassentyp `GraphObj` konstruieren soll. Durch die Angabe der Basisklasse wird die Zweideutigkeit beseitigt:

```

anzeigen(bR.Rechteck::bezugspunkt()); // eindeutig

```

Ferner wird durch verschiedene Bezugspunkte im Konstruktor nachgewiesen, dass `BeschriftetesRechteck` *zwei* `GraphObj`-Objekte besitzt:

```

// absichtlich veränderter Konstruktor
BeschriftetesRechteck(const Ort& ort, int h, int b,
                    const std::string& b)
: BeschriftetesObjekt(ort, b),
  Rechteck(Ort(100, 100), h, b) { // verschiedene Koordinaten!
}
// jetzt verschiedene Werte:
anzeigen(bR.Rechteck::bezugspunkt());
anzeigen(bR.BeschriftetesObjekt::bezugspunkt());

```

Weil zwei Subobjekte vom Typ `GraphObj` vorliegen, ist wegen der Nicht-Eindeutigkeit die Zuweisung eines Zeigers nicht möglich:

```

int main() {
    Rechteck r1(Ort(0,0), 20, 50);
    Rechteck r2(Ort(0,100), 10, 40);
    BeschriftetesRechteck bR2(Ort(1,20), 60, 60,
                             std::string("Mehrfachvererbung"));
    // Feld mit Basisklassenzeigern, initialisiert mit

```

```

// den Adressen der Objekte, 0 als Endekennung
GraphObj* graphObjZeiger[] = {&r1, &r2, 0}; // ok
// Fehler
// GraphObj* graphObjZeiger[] = {&r1, &r2, &bR, 0};

// Zeichnen aller Objekte im Feld
int i = 0;
while (graphObjZeiger[i]) {
    graphObjZeiger[i++] -> zeichnen();
}
}

```

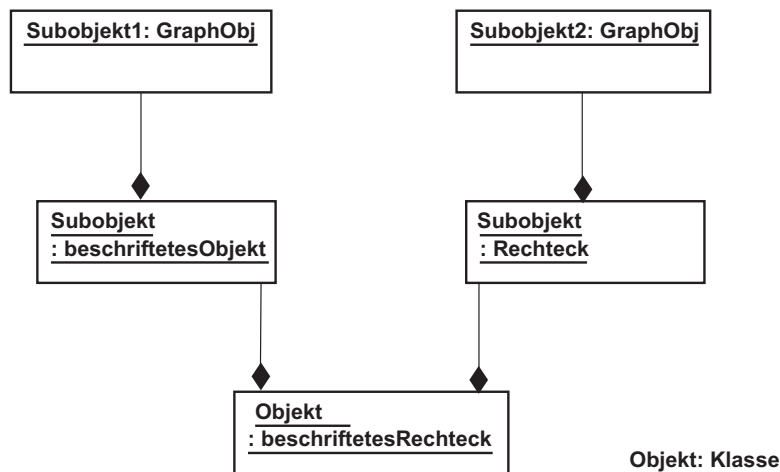


Abbildung 7.5: Zweideutig: *enthält*-Beziehungen bei nicht-virtueller Vererbung

Auf welches Subobjekt soll der Zeiger `graphObjZeiger[2]` verweisen, wenn die `//`-Markierung in der Zeile unter dem »Fehler«-Hinweis entfernt würde? Die tatsächliche Objekthierarchie für ein `BeschriftetesRechteck`-Objekt `bR` ergibt sich aus der Abbildung 7.5, wobei die Pfeile hier eine *enthält*-Beziehung symbolisieren, das heißt, das `beschriftetesRechteck bR` *enthält* ein `Rechteck`- und ein `beschriftetesObjekt`-Subobjekt, die beide je ein `GraphObj`-Subobjekt *enthalten*, deren Koordinaten nicht notwendigerweise gleich sein müssen. Im nächsten Abschnitt wird gezeigt, wie die Zweideutigkeiten aufgelöst werden.

7.8.2 Virtuelle Basisklassen

Wenn bei Mehrfachvererbung nicht erwünscht ist, dass mehrere Basisklassensubobjekte erzeugt werden, können *virtuelle Basisklassen* verwendet werden. Von diesen Basisklassen wird nur ein Subobjekt erzeugt, auf das über verschiedene Vererbungswege zugegriffen werden kann. Die Mehrdeutigkeit im obigen Beispiel wäre dadurch aufgehoben. Im Folgenden werden *nur* die Deklarationen und die Methoden aus dem vorherigen Abschnitt ganz oder teilweise aufgelistet, die notwendige Änderungen enthalten.

```
// Datei rechteck.h:
class Rechteck : virtual public GraphObj {
    // ... Rest wie vorher
};

// Datei beschriftetesobjekt.h:
class BeschriftetesObjekt : virtual public GraphObj {
    // ... Rest wie vorher
};

// Datei beschriftetesrechteck.h:
class BeschriftetesRechteck
    : public BeschriftetesObjekt, public Rechteck {
    // geänderter Konstruktor
    BeschriftetesRechteck(const Ort& ort, int h, int b,
                          const std::string& b)
    : GraphObj(ort),
      BeschriftetesObjekt(ort, b), // Diese Initialisierungen mit ort
      Rechteck(ort, h, b) { // werden ignoriert, siehe Abschnitt 7.8.2
    }
    // ... Rest wie vorher
};
```

Mit diesen Änderungen sind Aufrufe wie

```
cout << "BeschriftetesRechteck-Position: ";
anzeigen(bR.bezugspunkt());
```

möglich und unproblematisch, weil nun genau *ein* Basisklassensubobjekt für bR existiert. Der Konstruktor der Klasse `BeschriftetesRechteck` initialisiert jetzt das Basisklassensubobjekt; die Erklärung dafür finden Sie im folgenden Unterabschnitt »Virtuelle Basisklassen und Initialisierung«.

Weil nun genau ein Basisklassensubobjekt pro vollständigem Objekt existiert, kann ein Basisklassenzeiger auf ein Objekt der abgeleiteten Klasse gerichtet und damit der Polymorphismus ausgenutzt werden. Unter einem »vollständigen Objekt« wird ein Objekt verstanden, das nicht als Subobjekt dient, also nicht in einem anderen Objekt durch Vererbung enthalten ist. Im folgenden Beispiel sind R1, R2 und bR vollständige Objekte, nicht aber die in ihnen enthaltenen Subobjekte.

```
int main() { // geändert
    Rechteck R1(Ort(0,0), 20, 50);
    Rechteck R2(Ort(0,100), 10, 40);
    BeschriftetesRechteck bR(Ort(1,20), 60, 60,
                           std::string("virtuelle Mehrfachvererbung"));
    // Feld mit Basisklassenzeigern, initialisiert mit
    // den Adressen der Objekte, 0 als Endekennung
    GraphObj* graphObjZeiger[] = {&R1, &R2, &bR, 0}; // jetzt ok!
    // Zeichnen aller Objekte
    int i = 0;
    while(graphObjZeiger[i])
        graphObjZeiger[i++]->zeichnen();
}
```

Virtuelle Basisklassen und Initialisierung

Im Abschnitt 7.1 (Seite 263) wird die Initialisierung von Subobjekten behandelt. Dabei werden Initialisierer in einer Liste angegeben, die noch vor dem Codeblock des Konstruktors abgearbeitet wird. In einer Klassenhierarchie kann es mehrere Initialisierer für eine Basisklasse geben. Falls wir jedoch virtuelle Basisklassen haben, wird *nur ein* Subobjekt dieser Basisklasse in Objekten einer abgeleiteten Klasse angelegt. Dann darf natürlich *ein* Initialisierer wirksam werden, damit es keine widersprüchlichen Ergebnisse gibt, wenn einer »Links!« und der andere »Rechts!« sagt. Um dieses Problem zu lösen, wird in C++ der Basisklasseninitialisierer genommen, *der bei dem Konstruktor eines vollständigen Objekts angegeben ist*, also einem Objekt, das bei der Definition in der Vererbungshierarchie ganz unten steht und das daher nicht als Subobjekt innerhalb eines anderen Objekts dient. Die anderen Basisklasseninitialisierer werden *ignoriert*. Wenn im Konstruktor *kein* Basisklasseninitialisierer aufgeführt ist, wird der Standardkonstruktor der virtuellen Basisklasse genommen. Das Programm zeigt die Initialisierung von Subobjekten virtueller Basisklassen. Es gibt zweimal *Basis-Standardkonstruktor* aus. Der Basisklasseninitialisierer *Basis(a)* in der Klasse *Rechts* wird beim Konstruktor von *Unten* ignoriert.

Listing 7.13: Initialisierung bei virtueller Basisklasse

```
// cppbuch/k7/mehrfach/basinit.cpp
#include<iostream>
class Basis {
public:
    Basis() { std::cout << "Basis-Standardkonstruktor\n"; }
    Basis(const char* a) { std::cout << a << std::endl; }
    virtual ~Basis() {} // virtueller Destruktor
};

class Links : virtual public Basis {
public:
    Links(const char* a)
        // : Basis(a) // siehe Text unten
    { }
};

class Rechts : virtual public Basis {
public:
    Rechts(const char* a) : Basis(a) {}
};

class Unten: public Links, public Rechts {
public:
    Unten(const char* a) :
        // Basis(a), // siehe Text unten
        Links(a), Rechts(a) {}
};

int main() {
    Unten un("Unten");
    Links li("Links");
}
```


Stattdessen wird nur der beim Konstruktor von Unten direkt angegebene Basisklassenkonstruktor berücksichtigt. Da er hier auskommentiert ist, wird der Standardkonstruktor von Basis genommen. Wenn jedoch die Kommentarzeichen `//` aus den Initialisierungslisten entfernt werden, ist die Ausgabe

Unten

Links.

`Rechts::Basis(a)` wird weiterhin ignoriert. Die Regel ist: Der Konstruktor eines vollständigen Objekts ist für die Initialisierung des Basisklassensubobjekts bei virtueller Vererbung verantwortlich.

7.9 Standard-Typumwandlungsoperatoren

Meistens sind zunächst scheinbar notwendige Typumwandlungen nur ein Zeichen für schlechtes Design und sollten daher zum Nachdenken anregen. Die Typumwandlung (*cast*) im C-Stil umgeht die Typkontrolle durch den Compiler und ist deshalb gefährlich. Die syntaktische Notation nur durch Klammern kann leicht übersehen werden und ist auch mit Werkzeugen oder automatisierter Suche mit dem Editor schwierig, wenn man alle Casts verschiedener Datentypen finden will.

Andererseits sind Typumwandlungen für spezielle Zwecke notwendig, wie unter anderem in Abschnitt 7.10 gezeigt wird. Um die Nachteile der Casts im C-Stil zu umgehen, wurden neue Typumwandlungsoperatoren entworfen, die die vorherigen Casts überflüssig machen. Sie haben einige Vorteile:

- Sie sind durch ihre Namen optisch und syntaktisch leicht zu erkennen.
- Sie sind spezialisiert, sodass nur der erwünschte Effekt eintritt – also nicht mehr ein Cast für alles.

Die Syntax ist bis auf den Operatornamen für alle Typumwandlungs-Operatoren gleich:

Operatorname`<T>(Ausdruck)`

Das Ergebnis des Ausdrucks soll in den Typ *T* gewandelt werden.

Der `static_cast`-Operator

Der `static_cast`-Operator ist dazu gedacht, implizit erlaubte Standard-Typumwandlungen durchzuführen oder rückgängig zu machen (vergleiche Beispiel auf Seite 80):

```
enum Wochentag {sonntag, montag, dienstag, mittwoch,
               donnerstag, freitag, samstag
               } heute = dienstag;

int i = dienstag;           // implizite Umwandlung nach int
heute = i;                  // Fehler, Datentyp inkompatibel
heute = static_cast<Wochentag>(i); // erlaubt!
```

Falls die Variable `i` einen Wert hat, der nicht einem der Werte des Aufzählungstyps entspricht, ist der Wert der Variablen `heute` undefiniert.

Die implizite Typumwandlung in einer Klassenhierarchie, die auf Seite 266 beschrieben wird, lässt sich ebenfalls invertieren, sodass zum Beispiel Wandlungen wie `Basis*` zu `Abgeleitet*` vorgenommen werden können:

```
GraphObj g(Ort(3, 17));
Strecke s(Ort(3, 17), (Ort(0, 0))); // Strecke ist von GraphObj abgeleitet
GraphObj *pg;
Strecke *ps = &s;
pg = ps; // bekannte implizite Konversion
ps = pg; // verboten!
ps = (Strecke*) pg; // gefährlicher C-Stil!
ps = static_cast<Strecke*> (pg); // richtig (falls pg auf ein Strecke-Objekt zeigt)
```

Der `static_cast`-Operator ist nur dann geeignet, wenn zur Compilierzeit bereits feststeht, dass der Basisklassenzeiger (`pg`) auf ein Objekt einer abgeleiteten Klasse zeigt. Anstelle von Zeigern sind Referenzen möglich. Die Typumwandlung von einer Basisklasse zur abgeleiteten Klasse wird *downcast* genannt und ist nicht erlaubt, wenn die Basisklasse virtuell ist. Die `const`-Eigenschaft von Objekten kann nicht mit dem `static_cast` eliminiert werden.

Der `dynamic_cast`-Operator

Der Operator `dynamic_cast<T>(Ausdruck)` wirkt ähnlich wie der `static_cast`-Operator, jedoch mit folgenden Unterschieden:

- Die Typprüfung findet *zur Laufzeit* statt, falls das Ergebnis nicht schon zur Compilierzeit bestimmt werden kann. Dann verhält sich `dynamic_cast` wie ein `static_cast`. Weitere Möglichkeiten zur Typprüfung zur Laufzeit siehe Abschnitt 7.10.
- Typ `T` muss ein Zeiger oder eine Referenz auf eine Klasse sein.
- Falls das Argument *Ausdruck* ein Zeiger ist, der nicht auf ein Objekt vom Typ `T` (oder abgeleitet von `T`) zeigt, wird als Ergebnis der Typumwandlung ein Null-Zeiger auf den Ergebnistyp, d.h. `(T*)0` zurückgegeben.
- Falls das Argument *Ausdruck* eine Referenz ist, die nicht auf ein Objekt vom Typ `T` (oder abgeleitet von `T`) verweist, wird eine Ausnahme (Exception) vom Typ `bad_cast` ausgeworfen.

Die wesentlichen Varianten sind im Beispiel dargestellt:

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

class Abgeleitet : public Basis {
public:
    virtual void f() {}
};

Abgeleitet* g(Basis *pB) { // g() benutzt f()
    Abgeleitet *pA = dynamic_cast<Abgeleitet*>(pB);
```

```

    if(pA)        // NULL bei Scheitern des dynamic_cast
        pA->f();    // Abgeleitet::f()
    return static_cast<Abgeleitet*>(pB);
}

int main() {
    Basis einB;
    Abgeleitet einA;
    Basis *pBB = & einB;
    Basis *pBA = & einA;
    Abgeleitet *pErgebnis;
    // Durch den folgenden Aufruf von g() wird Abgeleitet::f() ausgeführt,
    // weil pBA auf ein Abgeleitet-Objekt zeigt. pErgebnis zeigt auf einA:
    pErgebnis = g(pBA);

    // Abgeleitet::f() wird unten nicht ausgeführt, weil pB in g() auf ein Basis
    // -Objekt zeigt. pErgebnis ist undefiniert(!), weil der static_cast ungeeignet
    // ist: Der dynamische Typ des per Zeiger übergebenen Objekts ist nicht vom Typ
    // Abgeleitet.
    pErgebnis = g(pBB);
} // Ende von main()

```



Übung

7.4 Lösen Sie die Aufgabe 7.3 auf Seite 285 mit dem `dynamic_cast<>()`-Operator. Geben Sie die Matrikelnummern aller Personen aus, sofern diese eine haben.

Der `const_cast`-Operator

Dieser Operator ist der einzige, der die `const`-Eigenschaft eines Objekts beseitigen kann. Dementsprechend sollte er möglichst nicht eingesetzt werden, zumal ein verändernder Zugriff auf ein konstantes Objekt über `const_cast` zu einem unerwarteten Verhalten eines Programms führen kann.

```

const int i = 100;
const int *ip = &i;
*ip = 0;                // geht nicht
int *iq = const_cast<int*>(&i); // explizite Typumwandlung
*iq = 0;                // Wert von i wird geändert!

```

Der Operator `const_cast<T>(obj)` ändert die `const`-Eigenschaft des Objektes `obj`. Der Datentyp von `obj` muss `const T` (oder `T`) sein, wobei `T` auch ein Zeiger oder eine Referenz sein kann. Der `const_cast`-Operator ersetzt die früher übliche Form `(T) obj`, die eine erzwungene Typumwandlung eines *beliebigen* Datentyps nach `T` bewirkt. Die Typumwandlung sollte nur in begründeten Ausnahmefällen vorgenommen werden; schließlich hat eine `const`-Deklaration ihren Sinn. Der Operator kann auch benutzt werden, um einen nicht-konstanten Typ als konstant erscheinen zu lassen. Beispiel für einen Typ `X`:

```

X einX;
X const& cr = const_cast<X const&>(einX);

```

Über `cr` können für das Objekt `einX` nur nicht verändernde (d.h. `const`-qualifizierte) Methoden aufgerufen werden.

Der reinterpret_cast-Operator

Dieser Operator kann die `const`-Eigenschaft eines Objekts nicht ändern, aber ansonsten ist jede Typumwandlung möglich. Die Typumwandlung findet zur Compilierzeit statt. Ein Objekt wird im Sinne des gewünschten Datentyps »re-interpretiert«. Weil ganz verschiedene Datentypen ineinander gewandelt werden können, ist das Ergebnis meistens implementationsabhängig. Dieser Operator sollte nur in den ganz seltenen Fällen benutzt werden, in denen die Anwendung der vorher beschriebenen Typumwandlungsoperatoren nicht möglich ist, zum Beispiel wenn es nur um die reinen Bits geht wie bei der binären Ein-/Ausgabe in Abschnitt 5.8.

7.10 Typinformationen zur Laufzeit

Der oben beschriebene `dynamic_cast`-Operator wandelt den Typ eines Objekts zur Laufzeit und führt dabei gleichzeitig eine Prüfung durch. In den meisten Fällen ist dies ausreichend, manchmal möchte man aber mehr wissen. Die Laufzeit-Typinformation kann für alle Methoden benutzt werden, die als Argument den Klassentyp selbst (d.h. auch Zeiger und Referenzen auf die Basisklasse) haben und polymorph benutzt werden sollen.

Typidentifizierung mit typeid()

Das Ergebnis eines `typeid()`-Ausdrucks ist vom vordefinierten Typ `type_info&`. Wenn das Argument von `typeid()` ein polymorpher Typ ist, bezieht sich das Ergebnis von `typeid()` auf das zugehörige vollständige Objekt. Mit »polymorpher Typ« ist gemeint, dass das Argument eine Referenz vom Basisklassentyp ist, die auf ein Objekt einer abgeleiteten Klasse verweist. Die Dereferenzierung eines Zeigers durch ein vorangestelltes `*` liefert ebenfalls eine Referenz:

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

class Basis { ... };
class Abgeleitet: public Basis { ... };

int main() {
    Basis einBasisObjekt;
    Abgeleitet Objekt1, Objekt2;
    Basis *p = &Objekt1;
    Basis *pNull = 0;
    if(typeid(Objekt2) == typeid(*p)) { // *p ist polymorph
        cout << "true";
    }
}
```

```

else {
    cout << "false";
}
if(typeid(Objekt1) == typeid(einBasisObjekt)) {
    cout << "true";
}
else {
    cout << "false";
}
if(typeid(Objekt1) == typeid(*pNull))
// ...

```

Im Programm wird nacheinander *true* und *false* ausgegeben, bevor in der letzten *if*-Anweisung eine *bad_typeid*-Ausnahme ausgeworfen wird, weil *pNull* ein Null-Zeiger ist. Der Vergleichsoperator vergleicht die von *typeid()* zurückgegebenen *type_info*-Objekte. Anstelle eines Objekts kann der Klassenname verwendet werden, die beiden Abfragen

```

if(typeid(Objekt2) == typeid(*p)) { ... }
if(typeid(Abgeleitet) == typeid(*p)) { ... }

```

haben dieselbe Wirkung. Der Typ eines Objekts (Klassenname) kann als compilerabhängiger Wert vom Typ `const char*` erhalten werden:

```
cout << typeid(Objekt1).name(); // Ausgabe: Abgeleitet
```

Auf Seite 371 finden Sie ein Beispiel für die Anwendung von *typeid*.



Übung

7.5 Lösen Sie die Aufgabe 7.3 auf Seite 285 mit dem *typeid()*-Operator. Geben Sie die Matrikelnummern aller Personen aus, sofern diese eine haben.

7.11 Using-Deklaration für Klassen

Ein Namespace ist ein Sichtbarkeitsbereich (englisch *scope*) ähnlich wie der einer Klasse, der ebenfalls einen Namen hat. Die für Namespaces verwendete Using-Deklaration wird in ähnlicher Form für Klassenmethoden verwendet, um in einer abgeleiteten Klasse den gezielten Zugriff auf eine Basisklassenmethode zu ermöglichen:

```

class Basis {
protected:
    void f(int);
};

class Abgeleitet : public Basis {
public:
    using Basis::f;
    // ...
};

```

Mit dieser Using-Deklaration ist `Abgeleitet::f()` ein *öffentliches* Synonym für `Basis::f()`:

```
Basis einBasisObjekt;
einBasisObjekt.f(0); // Fehler! f() ist nicht public.

Abgeleitet einAbgeleitetObjekt;
einAbgeleitetObjekt.f(0); // ok
```

Private Methoden der Klasse `Basis` können auf diese Art nicht öffentlich gemacht werden.

7.12 Private- und Protected-Vererbung⁴

Delegation ist *eine* Möglichkeit zur Wiederverwendung von Code, private Vererbung, auch Implementationsvererbung genannt, ist eine andere. Die Delegation ist vorzuziehen, um mit der Vererbung ausschließlich eine *ist-ein*-Beziehung zwischen Klassen abzubilden (Vererbung der *Schnittstellen*). Aber Sie sollen wenigstens wissen, was private Vererbung bedeutet, wenn auf der nächsten Party die Rede davon ist, um elegant zu einem für eine Party interessanteren Thema wechseln zu können. Die private Vererbung wird hier am Beispiel einer Warteschlange oder Queue gezeigt (ansonsten sollten Sie die Klasse `std::queue` der Standardbibliothek benutzen). Dabei machen wir uns die Eigenschaften der Klasse `std::list`, einer doppelt-verketteten Liste (siehe auch Seite 772), zunutze. Eine einfache Anwendung könnte wie folgt aussehen:

Listing 7.14: Art der Vererbung ist nicht sichtbar

```
// cppbuch/k7/privat/main.cpp
#include<string>
#include<iostream>
#include"warteschlange.t"
using namespace std;

int main() {
    Warteschlange<string> fifo;
    fifo.push( string("eins"));
    fifo.push( string("zwei"));
    fifo.push( string("drei"));

    while(!fifo.empty()) {
        cout << fifo.size() << " Element(e) vorhanden!\n";
        string buf = fifo.front(); // lesen
        fifo.pop();               // löschen
        cout << "Element " << buf << " entnommen\n";
    }
    cout << "Liste ist leer!" << endl;
}
```

⁴ Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

Bei privater Vererbung dürfen öffentliche Methoden der Oberklasse zwar innerhalb der Unterklasse benutzt werden, nicht aber von Objekten der Unterklasse (vgl. Tabelle 7.1 auf Seite 265). Es wird nicht mehr die Schnittstelle geerbt, sondern die Implementierung. Sollen einzelne Methoden für Objekte abgeleiteter Klassen nutzbar sein, also für Objekte der Klasse Warteschlange, sind sie durch eine Benutzungsdeklaration (englisch *using declaration*) zu kennzeichnen, wie in Abschnitt 7.11 beschrieben. Die Benutzungsdeklaration besteht nur aus dem Schlüsselwort `using` und dem Namen der Funktion einschließlich der Klassenbezeichnung, aber ohne Parameterliste und Rückgabety. Nun ist erreicht, dass wirklich nur die gewünschten Methoden aufgerufen werden können. Falls von Warteschlange selbst eine weitere Klasse `public` abgeleitet würde, könnte sie nur die in der öffentlichen Schnittstelle von Warteschlange deklarierten Methoden benutzen. Auf diese Art wird der von der `std::list`-Klasse vererbte Methodenumfang der Oberklasse ausgewählt.

Listing 7.15: Klasse mit privater Vererbung

```
// cppbuch/k7/privat/warteschlange.t Warteschlangen-Template
#ifndef WARTESCHLANGE_T
#define WARTESCHLANGE_T
#include<list>

template<typename T>
class Warteschlange
: private std::list<T> { // mit privater Vererbung (Implementationsvererbung)
public:
    using std::list<T>::empty;
    using std::list<T>::size;
    // am Ende einfügen:
    void push(const T& x) {
        std::list<T>::push_back(x);
    }
    // am Anfang entnehmen:
    void pop() {
        std::list<T>::pop_front();
    }
    // am Anfang bzw. Ende lesen
    using std::list<T>::front;
    using std::list<T>::back;
};
#endif
```

Ein privater Teil ist überflüssig, das verborgene Basisklassensubobjekt vom Typ `std::list` erledigt alles. Die Methoden `push()` und `pop()` existieren nicht unter diesem Namen in der Oberklasse und können deshalb nicht per `using`-Deklaration öffentlich gemacht werden. Konstruktor, Destruktor und Zuweisungsoperator sind nicht notwendig. Zum Vergleich sei hier ein Template gezeigt, in dem die Delegation anstelle der privaten Vererbung tritt:

```
template<typename T>
class Warteschlange { // mit Delegation an ein list-Objekt (Attribut liste)
public:
    bool empty() {
        return liste.empty();
    }
};
```

```

    }
    size_t size() {
        return liste.size();
    }
    // am Ende einfügen:
    void push(const T& x) {
        liste.push_back(x);
    }
    // am Anfang entnehmen:
    void pop() {
        liste.pop_front();
    }
    // am Anfang bzw. Ende lesen
    const T& front() {
        return liste.front();
    }
    const T& back() {
        return liste.back();
    }
private:
    std::list<T> liste;
};

```

Das Prinzip ist einfach: Ein Listenobjekt `liste` wird privat angelegt, und die Elementfunktionen der Klasse Warteschlange rufen die öffentlichen Elementfunktionen des Objekts `liste` auf. Die Klasse Warteschlange delegiert damit Aufgaben an die Klasse `std::list`, weswegen das Prinzip *Delegation* genannt wird. Bisher sind wir davon ausgegangen, dass man für dynamische Datenstrukturen einen besonderen Kopierkonstruktor benötigt, wie am Beispiel der »flachen« und »tiefen« Kopie auf Seite 236 gezeigt. Wenn ein besonderer Konstruktor notwendig ist, gilt dies meistens auch für einen Destruktor und einen Zuweisungsoperator. Das alles können wir hier vergessen! Durch die Delegation enthält jedes Warteschlange-Objekt ein Objekt vom Typ `std::list`, und nur dieses enthält eine dynamische Struktur. Weil bei der Kopie oder Zuweisung ein Objekt *elementweise* kopiert wird, wird also das einzige Element der Klasse Warteschlange kopiert (das private Objekt `liste`), indem der Kopierkonstruktor bzw. Zuweisungsoperator für dieses Objekt aufgerufen wird. Die Klasse `std::list` stellt alle Dienstleistungen bereit, sodass sie nicht besonders programmiert werden müssen, und die Klasse Warteschlange wird dadurch zu einem »Datentyp erster Klasse«, der genauso einfach wie die Grunddatentypen zu handhaben ist. Der Zuweisungsoperator jedes Elements (hier nur `liste`) wird bei der Zuweisung eines Warteschlange-Objekts aufgerufen.

protected-Vererbung

Die `protected`-Vererbung spielt nur selten eine Rolle. Ein bruchstückhaftes Beispiel zeigt die Syntax:

```

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

```



```
class Abgeleitet : protected Basis {  
    public:  
        void g();  
        // ....  
};
```

Die Wirkung ist, dass alle `public`-Elemente der Klasse `Basis` nunmehr `protected`-Elemente der Klasse `Abgeleitet` werden. Damit sind sie innerhalb der Klasse `Abgeleitet` und aller von ihr abgeleiteten Klassen benutzbar, aber nicht von außerhalb. Beispiel:

```
void Abgeleitet::g() { // Implementierung  
    f();               // ok, f() ist zugreifbar  
}
```

```
// Benutzung im main-Programm:  
Basis einBasisObjekt;  
einBasisObjekt.f();    // ok, public  
Abgeleitet einAbgeleitetObjekt;  
einAbgeleitetObjekt.g(); // ok, public  
einAbgeleitetObjekt.f(); // Fehler, nicht zugreifbar
```

8

Fehlerbehandlung

Dieses Kapitel behandelt die folgenden Themen:

- Strategien zur Fehlerbehandlung
- Fehler abfangen
- Was tun, wenn Abfangen nicht möglich ist?

Oft tritt ein Fehler in einer Funktion auf, der innerhalb der Funktion selbst nicht behoben werden kann. Der Aufrufer der Funktion muss Kenntnis von dem aufgetretenen Fehler bekommen, damit er den Fehler abfangen oder noch weiter »nach oben« melden kann. Ein Programmabbruch in *jedem* Fehlerfall ist benutzungsunfreundlich und nicht immer erforderlich. Eine generelle Fehlerbehebung ist leider nicht möglich; jeder Einzelfall ist gesondert zu überlegen. Mit den bisher behandelten Mitteln lassen sich verschiedene Strategien zur Fehlerbehandlung realisieren. In der Aufzählung wird angenommen, dass der Fehler in einer aufgerufenen Funktion stattfindet.

1. Das programmiertechnisch Einfachste ist der sofortige Programmabbruch innerhalb der Funktion, die einen Fehler feststellt. Wenn keine erläuternden Meldungen über den Abbruch ausgegeben werden, ist die Fehlerdiagnose erschwert.
2. Ein üblicher Mechanismus zur Fehlerbehandlung ist die Übergabe eines Parameters an den Aufrufer der Funktion, der Auskunft über Erfolg oder Misserfolg der Funktion

gibt. Der Aufrufer hat den Parameter auszuwerten und entsprechend zu reagieren. Der Parameter ist nach jedem Funktionsaufruf abzufragen.

```
int ergebnis1 = f(par1, par2, fehler); // fehler wird per Referenz übergeben.
if(fehler) {
    switch (fehler) {
        case 1: ergebnis1 = -1; break;
        case 2: ergebnis1 = 0; break;
        default: cout << "nicht behebbarer Fehler in f()!" << endl;
                exit(-2); // Programmabbruch
    }
}

// ...
int ergebnis2 = g(par3, fehler);
if(fehler)
    exit(-3);          // Programmabbruch
// ... usw.
```

Der Vorteil liegt in der selektiven Art und Weise, wie der Aufrufer einer Funktion Fehler an der Stelle des Auftretens behandeln kann, sodass vielleicht sogar ein Abbruch nicht nötig ist. Der Nachteil dieser Verfahrensweise besteht darin, dass der Programmcode durch viele eingestreute Prüfungen schwerfällig wirkt und dass die Lesbarkeit leidet. Ein zweiter Nachteil ist, dass sich der Programmierer die Abfragen aus Schreibfaulheit spart in der Hoffnung, dass alles gut gehen wird.

3. Eine Funktion kann im Fehlerfall die in der C-Welt übliche globale Variable `errno` setzen, die dann wie im vorhergehenden Fall abgefragt wird – oder auch nicht! Globale Variablen sind jedoch grundsätzlich nicht gut geeignet, weil sie die Portabilität von Funktionen beeinträchtigen und weil beim Zusammenwirken mehrerer Programmteile Werte der Variablen möglicherweise nicht mehr eindeutig sind.
4. Eine Funktion, die einen Fehler feststellt, kann eine andere Funktion zur Fehlermeldung und Bearbeitung aufrufen, die gegebenenfalls auch den Programmabbruch herbeiführt. Diese Methode erspart dem Aufrufer die Fehlerabfrage nach Rückkehr aus der Funktion. Es ist in Abhängigkeit von der Art des Fehlers zu überlegen, ob dem Aufrufer die Information des Fehlschlags mitgeteilt werden muss. Dies gilt sicher dann, wenn die Funktion die ihr zuge dachte Aufgabe nicht erledigen konnte.
5. Die leider noch anzutreffende »Kopf-in-den-Sand-Methode« ist die, einem Aufrufer trotz eines Fehlers einen gültigen Wert zurückzuliefern und weiter nichts zu tun. Beispiel: Den Index-Operator `[ ]` (siehe Kapitel 9) für eine String-Klasse so zu programmieren, dass bei einer Indexüberschreitung ohne Meldung immer das Zeichen an der Stelle 0 zurückgegeben wird. Diese Methode ist besonders tückisch, weil der Fehler sich möglicherweise durch die falschen Daten an einer ganz anderen Stelle und sehr viel später bemerkbar macht und daher schwer zu finden ist. Oder: Einzutragende Daten ohne Meldung verwerfen, wenn die Datei voll ist. Oder ähnlich Schlimmes mehr.

Die Art der Fehlerbehandlung hängt auch davon ab, wie sicherheitskritisch der Einsatz der Software ist. Eine Fehlermeldung *Index-Fehler im Array CONTROLPARAMETER, Index = 2198, max = 82* ist in einem Textverarbeitungsprogramm noch vertretbar und führt vielleicht zu einem Fehlerbericht an den Hersteller des Programms. Ganz anders kann es

sein, wenn diese Fehlermeldung in der Software eines in der Luft befindlichen Flugzeugs auftritt. Es reicht wahrscheinlich nicht aus, die Meldung mit Uhrzeit in einer Log-Datei zu speichern und dann im Programm fortzufahren. Es würde auch nichts helfen, dem Piloten *diese* Fehlermeldung anzuzeigen, weil er damit nichts anfangen kann.

Unter den oben genannten Möglichkeiten sind die zweite und noch mehr die vierte akzeptabel. C++ stellt zusätzlich die Ausnahmebehandlung bereit, mit der Fehler an spezielle Fehlerbehandlungsroutinen des aufrufenden Kontexts übergeben werden können.

8.1 Ausnahmebehandlung

Die vierte Fehlerbehandlungsstrategie von Seite 302 hat den Vorteil, dass der Programmcode, der die eigentliche Aufgabe erledigen soll, von vielen eingestreuten Fehlerprüfungen entlastet wird. C++ bietet die *Ausnahmebehandlung* (englisch *exception handling*) an, die ebenfalls den Fehlerbehandlungscode vom »normalen« Programm sauber trennt und die eine spezielle Abfrage von Fehlerparametern an vielen Stellen im Programm überflüssig macht. Zu unterscheiden ist zwischen der *Erkennung* von Fehlern wie zum Beispiel

- Division durch Null,
- Bereichsüberschreitung eines Arrays,
- Syntaxfehler bei Eingaben,
- Zugriff auf eine nichtgeöffnete Datei,
- Fehlschlag der Speicherbeschaffung oder
- Nichteinhaltung der Vorbedingung einer Methode

und der *Behandlung* von Fehlern. Die Erkennung ist in der Regel einfach, die Behandlung schwierig und häufig genug unmöglich, sodass ein Programm gegebenenfalls abgebrochen werden muss. Außer den *vorhersehbaren* Fehlern gibt es natürlich noch andere! Die Fehlerbehandlung sollte einen von zwei Wegen einschlagen:

- Den Fehler beheben und den Programmablauf fortsetzen oder, falls das nicht gelingt, das Programm mit einer aussagefähigen Meldung abbrechen *oder*
- den Fehler an das aufrufende Programm melden, das ihn dann ebenfalls auf eine dieser beiden Arten bearbeiten muss.

Für den Fall, dass Fehler in einer Funktion vom Aufrufer behandelt werden können, stellt C++ alternativ zur vierten Strategie auf Seite 302 die Ausnahmebehandlung zur Verfügung. Der Ablauf lässt sich wie folgt skizzieren:

1. Eine Funktion versucht (englisch *try*) die Erledigung einer Aufgabe.
2. Wenn sie einen Fehler feststellt, den sie nicht beheben kann, wirft (englisch *throw*) sie eine Ausnahme (englisch *exception*) aus.
3. Die Ausnahme wird von einer Fehlerbehandlungsroutine aufgefangen (englisch *catch*), die den Fehler bearbeitet.

try und catch

Die Funktion kann der Fehlerbehandlungsroutine ein Objekt eines beliebigen Datentyps »zuwerfen«, um Informationen zu übergeben. Im Unterschied zu der oben erwähnten vierten Strategie wird nach der Fehlerbehandlung *nicht* in die Funktion zurückgesprungen. Das Programm wird vielmehr mit dem der Fehlerbehandlung folgenden Code fortgesetzt.



Hinweis

Wenn aus einem Block herausgesprungen wird, werden die Destruktoren aller in diesem Block definierten automatischen Objekte aufgerufen! Diese Objekte werden dabei vom Laufzeit-Stack entfernt (englisch *stack unwinding*).

Dies kann ausgenutzt werden, um durch Exceptions verursachte Speicherlecks zu vermeiden (mit `shared_ptr`, siehe Seite 567). Aus dem vorhergehenden Absatz folgt, dass Destruktoren selbst *keine* Exceptions werfen dürfen, um das Aufräumen des Stacks nicht durch einen Sprung nach außen abrupt zu beenden. Das folgende Schema zeigt die syntaktische Struktur zum Werfen und Auffangen von Exceptions.

```
try {
    func();
    // Falls die Funktion func() einen Fehler entdeckt, wirft sie eine Ausnahme
    // aus (throw), wobei ein Objekt übergeben werden kann, um die geeignete
    // Fehlerbehandlung anzustoßen. Oder es wird in den Anweisungen ein Fehler
    // festgestellt, der zum Auswerfen einer Exception führt, etwa so:
    // weitere Anweisungen ...
    if(EsIstEinFehlerPassiert)
        throw Exception();
}

catch(Datentyp1 e) { // Syntax für Grunddatentypen, z.B. const char*
    // durch ausgeworfenes Objekt e ausgewählte Fehlerbehandlung
    // ...
}

catch(const Dentyp2& e) { // Klassenobjekte per Referenz übergeben
    // durch ausgeworfenes Objekt e ausgewählte Fehlerbehandlung
    // ...
}

// gegebenenfalls weitere catch-Blöcke
// Fortsetzung des Programms nach Fehlerbearbeitung an dieser Stelle!
// ...
```

Die Ausnahmebehandlung wird hier ausschließlich als Fehlerbehandlung verstanden. Um die Ausnahmebehandlung am Beispiel zu zeigen, wird davon leicht abgewichen, weil zum Beispiel das Erreichen des Dateiendes kein Fehler, sondern etwas Normales ist. Das Einlesen einer Zahl mit `cin >>` wird mit einer Fehlererkennung versehen, sodass fehlerhafte Eingaben ignoriert werden. Im folgenden Programm soll bei Fehleingaben die Meldung »Syntaxfehler« ausgegeben werden, ehe der Eingabestrom weiter verarbeitet wird. Das Dateiende wird bei Umleiten der Standardeingabe auf Betriebssystemebene erkannt. Die Kennung für ein Dateiende kann über die Tastatur eingegeben werden, falls die Stan-

dardeingabe nicht umgeleitet wird, in der Regel mit der Tastenkombination `Strg+Z` oder `Strg+D`.

Listing 8.1: Beispielprogramm zur Fehlerbehandlung

```
// cppbuch/k8/stream/exstream.cpp
#include<iostream>
using namespace std;

class DateiEnde : public exception {}; // Hilfsklasse (siehe unten),
// erbt von exception, siehe Seite 307

int liesZahl(std::istream& ein) {
    int i;
    ein >> i;
    // Das eof-Bit bewirkt den Auswurf eines Objekts vom Typ
    // DateiEnde, das hier durch den Aufruf des systemgenerierten
    // Konstruktors erzeugt wird, an den umgebenden Kontext. Die Eingabe von
    // falschen Zeichen setzt das fail-Bit, das durch die Funktion fail()
    // abgefragt wird und den Auswurf eines const char*-Objekts bewirkt.
    // Jedes throw führt zum Verlassen der Funktion. Details zu eof(), fail()
    // und bad() siehe Abschnitt 10.4.
    if(ein.eof()) throw DateiEnde();
    if(ein.fail()) throw "Syntaxfehler";
    if(ein.bad()) throw; // nicht behebbarer Fehler
    return i;
} // liesZahl()

void zahlen_lesen_und_ausgeben() {
    int zahl;
    while(true) { // Endlosschleife
        cout << "Zahl eingeben:";
        bool erfolgreich = true;
        try{ // Versuchsblock
            // Zahl von der Standardeingabe lesen:
            zahl = liesZahl(cin);
        }
        // Fehlerbehandlung: Der folgende Ausnahme-Handler wird
        // angesprungen, wenn ein Objekt des Typs DateiEnde im try-Block
        // ausgeworfen wurde.
        catch(const DateiEnde& e) {
            cout << "Ende der Datei erreicht! e.what() liefert : "
                 << e.what() // von exception geerbte Methode
                 << endl;
            cin.clear(); // Fehlerbits rücksetzen, siehe Abschnitt 10.4
            break; // Schleife verlassen
        }
        // Der folgende Ausnahme-Handler wird angesprungen, wenn ein Objekt des Typs
        // const char* ausgeworfen wurde. Die Funktion liesZahl() wirft einen
        // C-String aus, wenn in der Eingabe die Syntax von int-Zahlen verletzt
        // wird, zum Beispiel Buchstaben statt Ziffern.
        catch(const char* z) {
            cerr << z << endl;
            erfolgreich = false;
        }
    }
}
```

```

        cin.clear(); // Fehlerbits rücksetzen
        cin.get(); // fehlerhaftes Zeichen entfernen, s. Abschnitt 10.2
    }
    // bad() wird nicht abgefangen, ggf. Programmabbruch.
    // Fortsetzung des Programms nach der Fehlerbehandlung
    if (erfolgreich)
        cout << "Zahl = " << zahl << endl;
    }
}

int main() {
    zahlen_lesen_und_ausgeben();
}

```

Die Klasse `DateiEnde` kann von der Standardklasse `exception` (Beschreibung folgt) erben und deren Methoden nutzen, wie mit dem Aufruf von `what()` gezeigt wird. Falls `exception`-Methoden nicht benötigt werden, muss `DateiEnde` auch nicht von `exception` erben.

Für schwere Fehler, oben angezeigt durch `bad()`, ist hier keine Fehlerbehandlungsroutine definiert. Sie werden deswegen so lange an die nächsthöhere Ebene weitergereicht, bis sie auf einen geeigneten Ausnahme-Handler treffen. Ist wie hier keiner vorhanden, wird das Programm abgebrochen. Die Ausnahme-Handler werden der Reihe nach abgefragt, sodass einer, der auf alle Fehler passt, am Ende der Liste stehen muss, um die anderen nicht zu verdecken. Das obige Programm könnte entsprechend ergänzt werden. Die Folge von drei Punkten innerhalb runder Klammern wird *Ellipse* genannt, was so viel wie Auslassung bedeutet. Nichtangabe eines Datentyps durch `...` bedeutet »beliebige Anzahl beliebiger Datentypen«.

```

catch(...) { // Ellipse ... für nicht spezifizierte Fehler
    cerr << "nicht behebbarer Fehler!" << endl;
    throw; // Weitergabe an nächsthöhere Instanz
}

```

Der Vorteil der Trennung von Fehlererkennung und -behandlung wird durch einen Verlust an Lokalität erkauft, weil von der fehlerentdeckenden Stelle an eine ganz andere Stelle gesprungen wird, die auch Anlaufpunkt vieler anderer Stellen sein kann. Es lohnt sich daher, sich vor der Programmierung eine angemessene Strategie zur Fehlerbehandlung zu überlegen.

8.1.1 Exception-Spezifikation in Deklarationen

Mit Hilfe des Schlüsselworts `noexcept` kann dokumentiert werden, ob eine Funktion Ausnahmen auswerfen kann. Dabei sind drei Fälle zu unterscheiden:

1. `void zahlen_lesen_und_ausgeben();`
Es wird nichts ausgesagt, ob diese Funktion Ausnahmen auswerfen kann oder nicht.
2. `void zahlen_lesen_und_ausgeben() noexcept;`
verspricht, *keine* Ausnahmen auszuwerfen.
3. `void zahlen_lesen_und_ausgeben() noexcept(false);`
kann Ausnahmen auswerfen.

Der Benutzer einer Funktion kennt zwar ihren Prototyp, im Allgemeinen aber nicht die Implementierung der Funktion. Es kann sein, dass das Versprechen im Punkt 2 nicht eingehalten wird! In so einem Fall wird das Programm abgebrochen und damit eine spezialisierte Fehlerbehandlung unmöglich gemacht. `noexcept` sollte man selbst daher nur verwenden (wenn überhaupt), wenn man absolut sicher ist. Ohne `noexcept` hat der Aufrufer einer Funktion immer die Chance, mit `catch(...)` jede auftretende Exception abzufangen. `noexcept` wird von vielen Compilern noch nicht unterstützt.

8.1.2 Exception-Hierarchie in C++

Eigene Klassen zur Ausnahmebehandlung können sinnvoll sein, C++ stellt aber auch eine Reihe vordefinierter Exception-Klassen zur Verfügung (Abbildung 8.1).

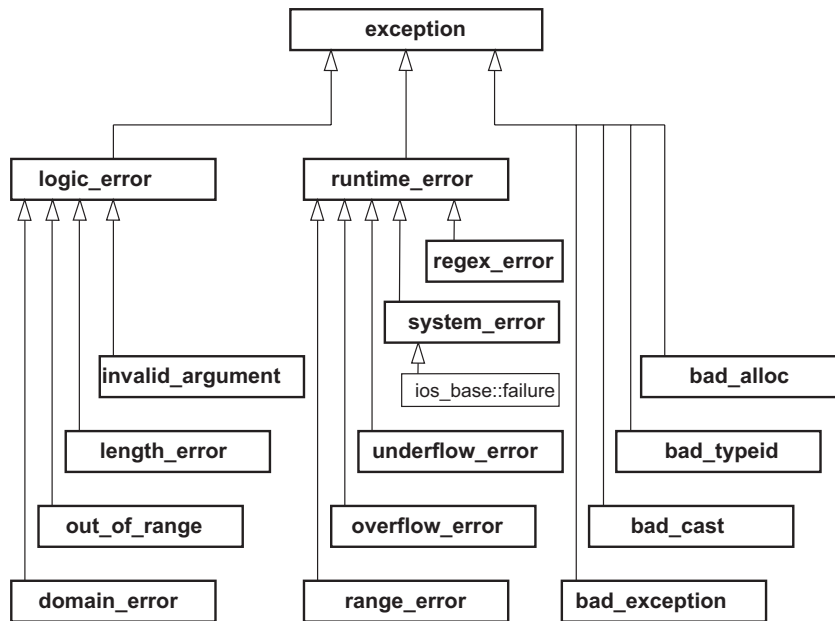


Abbildung 8.1: Exception-Hierarchie in C++

Dabei erben alle speziellen Exception-Klassen von der Basisklasse `exception`, wie Abbildung 8.1 zeigt. Die Basis-Klasse `exception` hat die folgende öffentliche Schnittstelle:

```

namespace std {
    class exception {
    public:
        exception() noexcept;
        exception(const exception&) noexcept;
        exception& operator=(const exception&) noexcept;
        virtual ~exception(); // Destruktor: wirft keine Exception
        virtual const char* what() const noexcept;
    };
}

```


Die Tabelle 8.1 zeigt die Zuständigkeit der Exception-Klassen.

Tabelle 8.1: Bedeutung der Exception-Klassen

Klasse	Bedeutung	Header
exception	Basisklasse	<exception>
logic_error	theoretisch vermeidbare Fehler, zum Beispiel Verletzung von logischen Vorbedingungen	<stdexcept>
invalid_argument	ungültiges Argument bei Funktionen	<stdexcept>
length_error	Fehler in Funktionen der Standard-C++-Bibliothek, wenn ein Objekt erzeugt werden soll, das die maximal erlaubte Größe für dieses Objekt überschreitet	<stdexcept>
out_of_range	Bereichsüberschreitungsfehler	<stdexcept>
domain_error	anderer Fehler des Anwendungsbereichs	<stdexcept>
runtime_error	nicht vorhersehbare Fehler, zum Beispiel datenabhängige Fehler	<stdexcept>
regex_error	Fehler bei regulären Ausdrücken	<regex>
system_error	Fehlermeldung des Betriebssystems	<system_error>
range_error	Bereichsüberschreitung	<stdexcept>
overflow_error	arithmetischer Überlauf	<stdexcept>
underflow_error	arithmetischer Unterlauf	<stdexcept>
bad_alloc	Speicherzuweisungsfehler (Details siehe Abschnitt 8.2)	<new>
bad_typeid	falscher Objekttyp (vgl. Abschnitt 7.10)	<typeinfo>
bad_cast	Typumwandlungsfehler (vgl. Abschnitt 7.9)	<typeinfo>

noexcept nach den Deklarationen bedeutet, dass die Methode verspricht, selbst keine Exception zu werfen, weil es sonst eine unendliche Folge von Exceptions geben könnte. Die von der Klasse exception geerbte Methode what() gibt einen Zeiger auf char zurück, der auf eine Fehlermeldung verweist. Eigene Exception-Klassen können durch Vererbung die Schnittstelle übernehmen, so wie die Klasse logic_error:

```
namespace std {
    class logic_error : public exception {
    public:
        explicit logic_error(const string& argument);
        // Zur Erinnerung: explicit verbietet implizite Typumwandlung.
    };
}
```

Dem Konstruktor kann ein string-Objekt mitgegeben werden, das eine Fehlerbeschreibung enthält, die im catch-Block ausgewertet werden kann.

8.1.3 Besondere Fehlerbehandlungsfunktionen¹

Nun kann es sein, dass mitten in einer Fehlerbehandlung selbst wieder Fehler auftreten. Die dann aufgerufene Funktion terminate() wird in diesem Abschnitt beschrieben. Das hier beschriebene Verhalten wird von manchen Compilern nur bedingt unterstützt. Daher ist es empfehlenswert, die Systemdokumentation zu Rate zu ziehen. Das ganze

¹ Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

Thema »Exception Handling« kann hier nicht annähernd vollständig dargestellt werden. Eine ausführliche und gut lesbare Auseinandersetzung damit findet sich in [ScMb]. `terminate()` ist im Header `<exception>` deklariert.

terminate()

Die Standardimplementierung von `void terminate()` beendet das Programm. Die Funktion `terminate()` wird (unter anderem) aufgerufen, wenn

- der Exception-Mechanismus keine Möglichkeit zur Bearbeitung einer geworfenen Exception findet;
- ein Destruktor während des Aufräumens (englisch *stack unwinding*) eine Exception wirft oder wenn
- ein statisches (nichtlokales) Objekt während der Konstruktion oder Zerstörung eine Exception wirft.

Benutzerdefinierte Fehlerbehandlungsfunktion

Die oben dargestellte Funktion kann bei Bedarf selbst definiert werden, um die vorgegebene zu ersetzen. Dazu ist standardmäßig ein Typ für Funktionszeiger definiert:

```
typedef void (*terminate_handler)();
```

Dazu passend gibt es eine Funktion, der ein Zeiger auf die selbst definierte Funktion dieses Typs übergeben wird, um die vorgegebene zu ersetzen:

```
terminate_handler set_terminate(terminate_handler f) noexcept;
```

Übergeben wird der Zeiger auf eine selbst definierte Funktion des Typs `terminate_handler`, die das Programm ohne Rückkehr an den Aufrufer beendet.

8.1.4 Erkennen logischer Fehler

Mit einem »logischen Fehler« ist gemeint, dass die gewünschte Semantik des Programms nicht korrekt in Programmcode umgesetzt worden ist, vermutlich durch einen »Denkfehler« des Autors. Wie können logische Fehler in einer Funktion oder allgemein in einem Programm gefunden werden, insbesondere während der Entwicklung? Dies ist möglich, indem in das Programm eine *Zusicherung* einer logischen Bedingung (englisch *assertion*) eingebaut wird. Das auf Seite 133 beschriebene Makro `assert()` dient diesem Zweck. Das Argument des Makros nimmt eine logische Annahme auf. Ist die Annahme wahr, passiert nichts; ist sie falsch, wird das Programm mit einer Fehlermeldung abgebrochen. Sämtliche logischen Überprüfungen mit `assert`-Makros werden auf einmal abgeschaltet, wenn die Compilerdirektive `#define NDEBUG` vor dem `assert()`-Makro definiert wurde. Damit entfallen aufwendige Arbeitsvorgänge mit dem Editor für auszuliefernde Software, die keinen Debug-Code mehr enthalten soll.

Der Nachteil dieses Makros besteht in dem erzwungenen Programmabbruch. Wenn an die Stelle eines Abbruchs eine besondere Fehlerbehandlung (mit oder ohne Programmabbruch) treten soll, kann einfach ein eigenes Makro unter Benutzung der Ausnahmebehandlung geschrieben werden, das zum Beispiel in der Datei `asserter.h` abgelegt wird:

Listing 8.2: Zusicherungs-Makro mit Exception

```
// cppbuch/k8/logik/assertex.h
#ifndef ASSERTEX_H
#define ASSERTEX_H
    #ifdef NDEBUG
        #define Assert(bedingung, ausnahme) ; // Leeranweisung
    #else
        #define Assert(bedingung, ausnahme) \BS
            if(!(bedingung)) throw ausnahme
        #endif
    #endif // ASSERTEX_H
```

Das folgende Programm zeigt beispielhaft, wie das Makro benutzt wird. Wenn 1 eingegeben wird, endet das Programm normal. Wenn 0 eingegeben wird, gibt es zusätzlich eine Fehlermeldung. In allen anderen Fällen wird das Programm mit einer Fehlermeldung abgebrochen. Es wird gezeigt, wie Exception-Objekten Informationen mitgegeben werden können, die innerhalb des Objekts ausgewertet werden. In diesem Fall ist es nur die Entscheidung, ob die übergebene Zahl gleich 0 oder gleich 1 ist.

Listing 8.3: Assert-Makro mit Exception

```
// cppbuch/k8/logik/main.cpp
#include<iostream>
#include<cstdlib> // exit()
// ggf. Abschalten der Zusicherungen mit NDEBUG
// #define NDEBUG
#include"assertex.h"

class GleichNull { // Exception-Klasse ohne Konstruktor-Argument
public:
    const char* what() const {
        return "Fehler GleichNull entdeckt";
    }
};

class UngleichEins { // Exception-Klasse mit Konstruktor-Argument
public:
    UngleichEins(int i) : zahl(i) {}
    const char* what() const {
        return "Fehler UngleichEins entdeckt";
    }
    int wieviel() const { return zahl;}
private:
    int zahl;
};

using namespace std;
int main() {
    int i;
    cout << "0          : GleichNull-Fehler\n"
        << "1          : normales Ende\n"
        << "! = 1       : UngleichEins-Fehler\n i = ?";
    cin >> i;
```

```

try {
    Assert(i, GleichNull()); // wirft ggf. Exception
    Assert(i == 1, UngleichEins(i)); // wirft ggf. Exception
}
catch(const GleichNull& fehlerObjekt) {
    cerr << fehlerObjekt.what() << endl
        << "keine weitere Fehlerbehandlung\n";
}
catch(const UngleichEins& fehlerObjekt) {
    cerr << fehlerObjekt.what() << endl
        << fehlerObjekt.wieviel() << '!'
        << " Abbruch" << endl;
    exit(1); // Programmabbruch
}
cout << "normales Programmende mit i = " << i << endl;
}

```

8.1.5 Arithmetische Fehler / Division durch 0

Es kann sein, dass eine arithmetische Operation nicht erlaubt ist, wie etwa die Division durch 0. Der C++-Standard sieht dafür keine Exception vor: »If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.« [ISOC++, Kap. 5] (Falls während der Auswertung eines Ausdrucks das Ergebnis mathematisch undefiniert oder nicht durch seinen Datentyp darstellbar ist, ist das Verhalten undefiniert.) Das folgende Beispiel einer Ganzzahl-Division durch 0 funktioniert daher im Allgemeinen nicht.

```

int a = 7;
int b = 0;
try {
    a = a / b;
} catch(...) { // fängt den Fehler nicht ab!
    cerr << "Division durch 0!";
}

```

Um C++-konform zu bleiben, muss man sich selbst darum kümmern, indem zum Beispiel der Nenner vor der Division auf 0 geprüft wird. Nur manche Compiler stellen Hilfsmittel zur Erkennung zur Verfügung, die von der Art der CPU abhängen. Bei Gleitkommazahlen stellt die Programmiersprache C, die ein Teil von C++ ist, einen Mechanismus zur Verfügung. Bei einer fehlerhaften Operation wird ein Flag gesetzt, das abgefragt werden kann. Nach der Abfrage sollte es gleich wieder gelöscht werden, um für nachfolgende Operationen bereit zu sein. Das folgende Programm zeigt nur die Arbeit mit dem Flag für die Division durch 0. Weitere Einzelheiten sind dem Abschnitt 7.6 des C-Standards [ISOC] zu entnehmen.

Listing 8.4: Test auf Division durch 0

```

// cppbuch/k8/division0.cpp
#include<iostream>
#include<cfenv>
using namespace std;

```

```

float f(float a, float b) {
    float c = a/b;
    if(fetestexcept(FE_DIVBYZERO)) { // abfragen
        feclearexcept(FE_DIVBYZERO); // zurücksetzen
        throw("Division durch 0!");
    }
    return c;
}

int main() {
    float z;
    float n;
    cout << "Zähler :";
    cin >> z;
    cout << "Nenner :";
    cin >> n;
    try {
        cout << "Ergebnis = " << f(z, n) << endl;
    } catch(const char* ex) {
        cerr << ex << endl;
    }
}

```

8.2 Speicherbeschaffung mit new

Eine Möglichkeit, das Fehlschlagen des new-Operators festzustellen, war früher die Abfrage, ob ein Null-Zeiger zurückgegeben wird. Es ist umständlich, nach jedem new diese Prüfung durchzuführen. Das C++-Standardkomitee hat sich deshalb entschlossen, dass bei Fehlschlagen der Speicherbeschaffung eine Ausnahme ausgeworfen wird, nämlich ein Objekt vom Typ `bad_alloc`. Die Alternative, einen Nullzeiger zurückzugeben, bleibt bestehen (siehe Ende des Abschnitts). In C++ gibt es einen Zeiger namens `new_handler` auf eine Funktion, die vom `new`- oder `new[]`-Operator gerufen wird, wenn die Speicherplatzanforderung nicht erfüllt werden kann. Der Zeiger ist wie folgt definiert:

```
typedef void (*new_handler)();
```

Zunächst zeigt `new_handler` auf eine Standardfunktion, die gegebenenfalls eine Ausnahme auswirft:

```

try {
    int *vielSpeicher = new int[30000];
}
catch(const bad_alloc&) {
    cerr << "kein Speicher vorhanden!" << endl;
    exit(1);
}
// ... normale Fortsetzung des Programms

```

Ohne try- und catch-Blöcke würde das Programm bei Speichermangel mit der Fehlermeldung »abnormal program termination« oder etwas Ähnlichem beendet. Der `new_handler`-Zeiger kann jedoch auf eine selbst definierte Funktion gerichtet werden, die Speicher beschaffen soll, wenn möglich. Dann wird im Fehlerfall diese Funktion aufgerufen, entsprechend der vierten in der Einführung zu diesem Kapitel genannten Möglichkeit. Der `new`-Operator arbeitet etwa auf folgende Weise:

```
void* ergebnis;
do {
    ergebnis = beschaffe_irgendwie_Speicher();
    if(ergebnis == NULL) {        // d.h. nicht erfolgreich
        if(new_handler == NULL) { // d.h. nicht definiert
            throw bad_alloc();
        }
        else {
            new_handler();        // Aufruf
        }
    }
} while(ergebnis == NULL);      // d.h. nicht erfolgreich
return ergebnis;
```

Die Schleife wird beendet, wenn es entweder gelungen ist, Speicher zu beschaffen, oder wenn kein Rücksprung aus `(*new_handler)()` erfolgt, weil eine Ausnahme ausgeworfen wurde. Im folgenden Programm wird der `new_handler`-Zeiger mithilfe der Bibliotheksfunktion `set_new_handler()` auf eine selbst definierte Funktion `speicherfehler()` gerichtet, wobei `set_new_handler()` einen Zeiger auf die vorher zugewiesene Funktion zurückgibt. Im Verlauf des `main()`-Programms wird ein Array von Zeigern angelegt.

Jedem Zeiger soll in der Schleife ein großer Block zugewiesen werden. Irgendwann ist jedoch der verfügbare Speicherplatz erschöpft. In diesem Moment wird während der Ausführung von `new` die Funktion `(*new_handler)()`, also `speicherfehler()`, aufgerufen. Die Funktion schafft mit `delete` Platz und deaktiviert sich selbst, weil sie nicht noch mehr Speicher beschaffen kann. Dann kehrt sie nach `new` zurück, wo sofort noch einmal versucht wird, den Speicherplatz zu beschaffen, was dieses Mal gelingt. Die Schleife läuft weiter. Irgendwann stellt sich wieder das Problem des knappen Speichers. Weil `speicherfehler()` nun eine Ausnahme auswirft, wird das Programm beendet. Falls in `speicherfehler()` kein Platz beschafft werden kann, ist durch Erzeugen einer Ausnahme das Programm abzubrechen, weil es sonst eine unendliche Schleife gibt, indem abwechselnd erfolglos versucht wird, Speicherplatz zu beschaffen und die Fehlerbehandlungsfunktion aufzurufen (siehe Programmbeispiel obiges zur Arbeitsweise von `new`).

Listing 8.5: Beispiel mit `new_handler`

```
// cppbuch/k8/speicher/newhdl.cpp
#include<iostream>
#include<new> // set_new_handler() und bad_alloc
using namespace std;
const unsigned int BLOCKGROESSE = 64000, MAXBLOECKE = 50000;
int* reserveSpeicher = 0;

void speicherfehler() throw (bad_alloc) {
    cerr << "Memory erschöpft! speicherfehler() aufgerufen!\n";
```

```

    if(reserveSpeicher) {
        cerr << "Einmal Platz schaffen!\n";
        delete [] reserveSpeicher;
        reserveSpeicher = 0;           // deaktivieren
    }
    else {
        cerr << "Exception auslösen!\n";
        throw bad_alloc();
    }
}

int main() {
    // eigene Fehlerbehandlungsfunktion eintragen
    set_new_handler(Speicherfehler);
    reserveSpeicher = new int[10*BLOCKGROESSE]; // Speicher belegen
    int* ip[MAXBLOECKE] = {0};
    unsigned int blockNr = 0;
    try {
        while(blockNr < MAXBLOECKE) { // Speicher fressen
            ip[blockNr] = new int[BLOCKGROESSE];
            cout << "Block " << blockNr << " beschafft" << endl;
            ++blockNr;
        }
        if(blockNr == MAXBLOECKE-1) { // Zählung ab 0
            cout << "Block-Array erschöpft" << endl;
        }
    }
    catch(const bad_alloc& exc) {
        cerr << ++blockNr
            << " Blöcke beschafft\n"
            << "bad_alloc ausgeworfen! Grund: "
            << exc.what() << endl;
    }
}

```

Das früher übliche Standardverhalten, bei Speichermangel NULL zurückzugeben, kann durch ein `nothrow`-Argument des `new`-Operators erreicht werden:

```

// p1 ist nie Null, bei Speichermangel wird eine bad_alloc-Exception ausgeworfen:
T *p1 = new T;
// bei Speichermangel wird Null zurückgegeben, eine Exception kann nicht erzeugt werden:
T *p2 = new(nothrow) T;
if(!p2) {
    cerr << "zuwenig Speicher!" << endl;
}
else {
    // ...
}

```

8.3 Exception-Sicherheit

Ein Programm ist exception-sicher (englisch *exception safe*), wenn Laufzeitfehler keine nachteiligen Auswirkungen haben. Um den Begriff zu präzisieren, gibt es verschiedene Stufen:

1. Auf der niedrigsten Stufe gibt es keinerlei Zusicherungen, ob und wie sich Fehler auswirken.
2. In dieser Stufe kann ein Programm zwar falsche Daten erzeugen, es soll aber weder abstürzen noch Speicherlecks oder verwitwete Objekte hinterlassen.
3. Eine starke Exception-Sicherheit liegt vor, wenn eine Operation entweder vollständig oder, bei einem Fehler, gar nicht oder so ausgeführt wird, dass das betroffene Objekt anschließend im selben Zustand wie vor der fehlerhaften Operation ist. Dies entspricht einem Commit bzw. Rollback in der Datenbank-Terminologie. Ein Beispiel ist die folgende Funktion zum Ablegen eines Elements auf einem Stack (Variante der Funktion von Seite 247):

```
template<typename T>
void SimpleStack<T>::push(const T& x) {
    if(full()) {
        throw std::logic_error("Stack-Overflow!");
    }
    array[anzahl++] = x;
}
```

Wenn noch Platz auf dem Stack ist, hat `push()` die gewünschte Wirkung. Wenn nicht, bleibt der Stack in seinem vorherigen Zustand.

4. In der sichersten Stufe wird garantiert, dass keine Fehler auftreten oder aber alle auftretenden Fehler abgefangen werden, sodass kein Fehler nachteilige Auswirkungen hat. Möglicherweise auftretende Exceptions werden nicht zum Aufrufer durchgereicht.

Die sicherste Stufe wird oft nicht zu realisieren sein. Selbst ein korrektes Programm kann nicht verhindern, dass es mit falschen Daten gefüttert wird. Die zweithöchste Stufe lässt sich hingegen meistens erreichen. Ob das immer wünschenswert ist, hängt vom Einzelfall ab. So kann man sich vorstellen, dass ein Objekt eine große Anzahl von Rechenergebnissen in einem Vektor speichert oder als Datei ausgibt. Wenn mitten in einer Berechnung ein Fehler auftritt, ist es vielleicht interessant, bis zu welcher Stelle die Berechnungen fehlerfrei erfolgen konnten. Ein gelöschter Vektor oder eine gelöschte Datei wären da nicht hilfreich.

Ein anderer Aspekt ist der zu treibende Aufwand und an welcher Stelle er betrieben werden soll. Nach dem »Design by Contract«-Prinzip [Mey] gewährleistet eine Funktion die Nachbedingung, wenn der Aufrufer die Vorbedingung einhält. Die Frage ist nun, ob eine Funktion jede Verletzung der Vorbedingung in den Eingabeparametern aufspüren und melden oder auf einen korrekten Aufruf vertrauen soll. Dazu zwei Beispiele:

1. Eine Funktion, die die Quadratwurzel aus einer positiven `double`-Zahl ≥ 0 berechnet, sollte eine Exception werfen, wenn sie mit einem negativen Argument aufgerufen wird. Die Prüfung kostet nicht viel Laufzeit und ist einfach durchzuführen.

2. Mit Hilfe der binären Suche einen Eintrag in einem Array zu finden, geht sehr schnell (z.B. Algorithmus `binary_search` von Seite 681). Der mittlere Aufwand ist proportional zum Logarithmus der Arraylänge N . Um in einem Array von 1024 Elementen einen Eintrag zu finden oder festzustellen, dass er nicht vorhanden ist, werden maximal 10 Schritte benötigt ($1024 = 2^{10}$). Vorbedingung ist, dass das Array sortiert ist. Wenn innerhalb von `binary_search()` diese Vorbedingung geprüft werden sollte, wären zusätzlich N Schritte notwendig, und der Performance-Vorteil wäre gänzlich dahin.

Es kommt also auf den Einzelfall an. Wenn sich eine Methode auf `binary_search()` verlässt, ohne die Sortierung des an `binary_search()` übergebenen Arrays zu prüfen, kann das betroffene Objekt in einen fehlerhaften Zustand geraten. Eine starke Exception Safety ist hier nur möglich, wenn die Sortierung des möglicherweise von außen stammenden Arrays sichergestellt ist oder überprüft wird.



Mehr dazu lesen Sie in Abschnitt 20.3.

9

Überladen von Operatoren

Dieses Kapitel behandelt die folgenden Themen:

- Neue Bedeutungen für Operatorsymbole
- Gestaltung mathematischer und Index-Operatoren
- Unterschied zwischen ++x und x++ bei Neudefinition
- Smart Pointer
- Objekt als Funktion
- new und delete überladen / Eigene Speicherverwaltung

Den vordefinierten Operatorsymbolen in C++ kann man für Klassen neue Bedeutungen zuordnen. Damit können Operationen mit Objekten auf sehr einfache Weise ausgedrückt werden. Am bekannten Beispiel der rationalen Zahlen wird gezeigt, wie die arithmetischen Operatoren, die vorher nur für einige Grunddatentypen definiert waren, zur einfachen Formulierung von Rechenoperationen mit rationalen Zahlen benutzt werden. Um wie viel einfacher ist es, in einem Programm mit Matrizen a, b und c

```
Matrix a = b + c/2;
```

zu schreiben, anstelle verschachtelter Schleifen unter genauer Beachtung der Randbedingungen wie Zeilen- und Spaltenzahlen! Um dieses Ziel zu erreichen, können in C++ Operatoren *überladen* werden, was nichts anderes bedeutet, als ihnen zusätzliche, problemabhängige Bedeutungen zu geben. Im obigen Fall würde der Operator + für eine Klasse `matrix` die Bedeutung bekommen, die Addition für alle Elemente der Matrix durchzuführen. In der C++-Standardbibliothek wird dieses Prinzip genutzt. Der Operator << ist

mehrfach überladen. Er dient nicht nur zum Verschieben von Bits, sondern wird in Verbindung mit `cout` zur Ausgabe von Daten verschiedenen Typs verwendet, sei es `char*`, `int`, `double` oder was sonst noch in Frage kommt.

Das Überladen von Operatoren funktioniert ähnlich wie das Überladen von Funktionen, wie es bereits in Abschnitt 3.2.5 (Seite 114) beschrieben wurde. Diesem Thema wurde dennoch ein eigenes Kapitel gewidmet, weil das Überladen von Operatoren im Unterschied zu Abschnitt 3.2.5 *Klassen voraussetzt*, die dort noch nicht behandelt wurden. Zudem zeigt die Möglichkeit des Überladens von Operatoren, dass darauf aufbauende Programme viel einfacher geschrieben werden können, wie das Beispiel am Kapitelanfang zeigt. Abbildung 9.1 zeigt die Syntax einer Operatordeklaration.

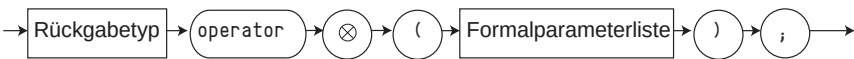


Abbildung 9.1: Syntaxdiagramm einer `operator`-Deklaration

⊗ steht für eines der möglichen C++-Operatorsymbole. Die Tabelle 9.1, auf die noch häufig Bezug genommen wird, zeigt, wie die Verwendung eines Operators vom Compiler in einen *Funktionsaufruf* umgewandelt wird (⊗ sei das Operatorsymbol). Bestimmte Operatoren können nur als Elementfunktion auftreten, andere auch als globale Funktion.

Tabelle 9.1: Operator als Funktionsaufruf

Element-Funktion	Syntax	Ersetzung durch
nein	<code>x ⊗ y</code>	<code>operator⊗(x, y)</code>
	<code>⊗ x</code>	<code>operator⊗(x)</code>
	<code>x ⊗</code>	<code>operator⊗(x, 0)</code>
ja	<code>x ⊗ y</code>	<code>x.operator⊗(y)</code>
	<code>⊗ x</code>	<code>x.operator⊗()</code>
	<code>x ⊗</code>	<code>x.operator⊗(0)</code>
	<code>x = y</code>	<code>x.operator=(y)</code>
	<code>x(y)</code>	<code>x.operator()(y)</code>
	<code>x[y]</code>	<code>x.operator[](y)</code>
	<code>x-></code>	<code>(x.operator->())-></code>
	<code>(T)x</code>	<code>x.operator T()</code>
	<code>static_cast<T>(x)</code>	<code>x.operator T()</code>
	<code>new T</code>	<code>T::operator new(sizeof(T))</code>
	<code>delete p</code>	<code>T::operator delete(p)</code>
	<code>new T[n]</code>	<code>T::operator new[](n * sizeof(T))</code>
	<code>delete[] p</code>	<code>T::operator delete[](p)</code>

T ist Platzhalter für einen Datentyp, p ist vom Typ `T*`.

Ein selbst definierter Operator *ist nichts als eine Funktion* in einer syntaktisch ansprechenderen Verkleidung! Einem in C++ vorhandenen Operator wird dabei eine neue, zusätzliche Bedeutung zugewiesen. Welche Bedeutung in einem Programmkontext die richtige ist, ermittelt der Compiler aus den Datentypen der Operanden. Eine Operatorfunktion unterscheidet sich von den bisher bekannten Funktionen nur in zweierlei Hinsicht:

1. Der Aufruf wird entsprechend der Tabelle 9.1 in einen Funktionsaufruf umgewandelt. Es macht keinen Unterschied, wenn man die Umwandlung selbst schon vornimmt und zum Beispiel `s = operator+(x, y)` statt `s = x + y` schreibt.
2. Der *Funktionsname* einer Operatorfunktion besteht aus dem Schlüsselwort `operator` und dem angehängten Operatorzeichen.

Aus Tabelle 9.1 wird ersichtlich, dass ein Überladen der unären Operatoren `++` und `--` eine Unterscheidung zwischen postfix- und präfix-Notation zulässt. Der Unterschied wird genauer auf Seite 334 beschrieben. Die Anzahl der Argumente ist um eins geringer, wenn der Operator eine Elementfunktion ist – unabhängig davon, ob es sich um einen unären oder binären Operator handelt. Sie ist um eins geringer, weil sich der Operator ohnehin schon auf ein Objekt bezieht. Es gibt einige Restriktionen für Operatorfunktionen:

- Es können die üblichen C++-Operatoren wie `=`, `==`, `+=` usw. überladen werden, nicht jedoch `.`, `*`, `::`, `?:` und andere Zeichen wie `$` usw. Eine Definition von neuen Operatoren ist *nicht* möglich, auch nicht durch Kombination von Zeichen. Die Operatoren `new` und `delete` sowie `new []` und `delete []` können überladen werden.
- Die vorgegebenen Vorrangregeln können nicht verändert werden.
- Wenigstens ein Argument der Operatorfunktion muss ein Klassen-Objekt sein oder die Operatorfunktion muss eine Elementfunktion sein. Auf diese Weise wird sichergestellt, dass niemand die vorgegebene Bedeutung von Operatoren für Grunddatentypen umdefinieren kann.

9.1 Rationale Zahlen – noch einmal

In diesem Abschnitt sollen Operatoren für rationale Zahlen überladen werden, wobei auf das Beispiel in Abschnitt 4.4 (siehe Seite 162 und folgende) Bezug genommen wird. Ziel ist es, dass man anstelle des Aufrufs der Funktionen `add()`, `eingabe()` und `ausgabe()` direkt zum Beispiel

```
Rational a, b, c;
cin >> a;           // überladener Operator
cin >> b;           // überladener Operator
c = a + b;          // überladener Operator
cout << c;          // überladener Operator
```

schreiben kann. Ähnliches gilt für die anderen arithmetischen Funktionen.

9.1.1 Arithmetische Operatoren

Weil der binäre `+`-Operator symmetrisch ist und die Argumente des Operators selbst nicht verändert werden sollen, empfiehlt sich ein überladener Additionsoperator, der *keine* Elementfunktion der Klasse `Rational` ist. Eine gemischte Verwendung der Datentypen `long` und `Rational` soll natürlich möglich sein. Es gibt zwei Möglichkeiten in Abhängigkeit vom Vorhandensein des Typumwandlungskonstruktors (siehe die Diskussion in Abschnitt 4.4).

Zunächst muss überlegt werden, welche der möglichen Operatorvarianten aus Tabelle 9.1 ausgewählt werden sollte. Die Addition ist ein binärer Operator, sodass nur die Zeilen eins und vier der Tabelle in Frage kommen. Die Addition ist kommutativ, das heißt, $x + y = y + x$, also eine symmetrische Operation. Zu einer rationalen Zahl sollte auch eine ganze Zahl addiert werden können, die ja nur ein Spezialfall einer rationalen Zahl mit dem Nenner 1 ist. Damit lassen sich drei Fälle beschreiben:

1. Addition zweier rationaler Zahlen: $z = x + y$
 Umformulierung entsprechend Tabelle 9.1:
 A) `z = x.operator+(y)`
 B) `z = operator+(x, y)`
 Der Operator muss ein Objekt vom Typ `Rational` zurückgeben, welches `z` zugewiesen wird. Die Objekte `x` und `y` sollen nicht verändert werden. Lösung A) ist nicht zu empfehlen, weil sie auf den ersten Blick eine Änderung von `x` suggeriert.
2. Addition einer rationalen und einer ganzen Zahl, zum Beispiel $z = x + 3$
 Umformulierung entsprechend Tabelle 9.1:
 A) `z = x.operator+(3)`
 B) `z = operator+(x, 3)`
 Beide Lösungen sind zunächst denkbar. Lösung A) ist aus demselben Grund wie oben abzulehnen.
3. Addition einer ganzen und einer rationalen Zahl, zum Beispiel $z = 3 + y$
 Umformulierung entsprechend Tabelle 9.1:
 A) `z = 3.operator+(y); // ?`
 B) `z = operator+(3, y);`
 Lösung A) ist ausgeschlossen, weil der selbst definierte Operator keine Elementfunktion einer ganzen Zahl sein kann! In C++ ist der Grunddatentyp `int` nicht als Klasse implementiert, und selbst wenn es so wäre, könnten wir nicht nachträglich eine neue Elementfunktion hinzufügen.

Aus dem Vergleich ergibt sich, dass die Lösung B) die einzig mögliche ist, wenn man die Symmetrie in der Schreibweise bei verschiedenen Datentypen erhalten will. Wenn entsprechend B) die Operatorfunktion `operator+()` *nicht* als Elementfunktion der Klasse `Rational` definiert werden soll, muss noch das Problem gelöst werden, dass die Operatorfunktion lesend auf die internen, privaten Daten von `Rational`-Objekten, also Zähler und Nenner, zugreifen darf. Wir benutzen die Methoden `getZaehler()` und `getNenner()`, die wegen ihrer inline-Eigenschaft schnell sind. Eine Alternative ist die Nutzung der Kurzformoperatoren, ähnlich wie schon auf Seite 167 gezeigt. Die folgende Zeile zeigt die Lösung zu Fall 3. B). Die Begründung für den `const`-Spezifizierer am Anfang der Zeile finden Sie als Hinweis auf Seite 165

```
const Rational operator+(const Rational&, const Rational&);
```

Ohne Typumwandlungskonstruktor

Um gemischte Datentypen zu erlauben, fügen wir zwei Deklarationen hinzu, mit denen Operationen wie $z = x + 3$ oder $z = 3 + y$ bearbeitet werden können:

```
const Rational operator+(Long, const Rational&);
const Rational operator+(const Rational&, Long);
```

Die Implementierung könnte wie folgt aussehen:

```
const Rational operator+(const Rational& a, const Rational& b) {
    return Rational(a.getZaehler()*b.getNenner() +
        b.getZaehler()*a.getNenner(), a.getNenner()*b.getNenner());
}

const Rational operator+(long a, const Rational& b) {
    Rational t(a, 1); // t wird zu (a/1)
    return t + b;     // Aufruf von +(Rational, Rational)
}

const Rational operator+(const Rational& a, long b) {
    return b+a;       // Aufruf von +(long, Rational)
                     // durch vertauschte Reihenfolge
}
```



Hinweis

Bei der Rückgabe temporärer Objekte wird der Kopierkonstruktor aufgerufen – wenn der Compiler den Aufruf nicht wegoptimiert. Ab Seite 589 wird auf dieses Problem und mögliche Lösungen besonders eingegangen.

Mit Typumwandlungskonstruktor

Wenn ein Typumwandlungskonstruktor wie in Abschnitt 4.3.4 oder im Beispiel auf Seite 322 vorhanden ist, könnten alle Prototypen und Implementationen des Operators entfallen, die einen long-Parameter enthalten.

Implementierung mit Ausnutzen der Kurzformoperatoren

Mit Hilfe der Kurzformoperatoren lassen sich binäre Operatoren deutlich kürzer formulieren. Dazu muss zunächst `Rational operator+=(const Rational &)` in der Klassendeklaration nachgetragen werden. Die Definition dieses Operators sei dem Leser überlassen (siehe Übungsaufgaben Seite 323). `operator+()` kann dann leicht mithilfe von `+=` implementiert werden. Weil `operator+()` nicht auf private Daten eines `Rational`-Arguments zugreift, muss `operator+()` weder Element- noch friend-Funktion sein.

```
const Rational operator+(const Rational& a, const Rational& b) {
    Rational temp = a;
    return temp += b;    // Rückgabe von temp.operator+=(b)
}
```

oder noch kürzer durch impliziten Aufruf des Kopierkonstruktors

```
const Rational operator+(const Rational& a, const Rational& b) {
    return Rational(a) += b;
}
```

Der `+=`-Operator ist in allen vorgestellten Formen sehr schnell, sofern der Compiler den Aufruf des Kopierkonstruktors bei der Rückgabe eliminiert, wie in Abschnitt 4.3.3, Seite 159, beschrieben.

9.1.2 Ausgabeoperator <<

Um eine rationale Zahl r mit einer einfachen Anweisung `cout << r`; ausgeben zu können, wird der Operator `<<` überladen. Aus der Syntax der Anweisung und der Tabelle 9.1 auf Seite 318 sehen wir, dass die Anweisung entweder als a) `cout.operator<<(r)`; interpretiert werden würde oder b) als `operator<<(cout, r)`;. Der Operator kann also keine Elementfunktion der Klasse `Rational` sein, sondern der Interpretation a) folgend höchstens eine der Klasse `ostream`, der `cout` angehört. Aber diese Möglichkeit kommt nicht in Frage, weil die Konstrukteure der Klasse `ostream` nicht jede mögliche Ausgabe einer benutzerdefinierten Klasse vorhersehen und einbauen konnten. Daher bleibt nur die Interpretation b), woraus sich ergibt, dass der Operator `<<` als globale Funktion mit zwei Argumenten formuliert werden muss:

```
std::ostream& operator<<(std::ostream& s, const Rational&);
```

Der Operator gibt eine Referenz auf das `ostream`-Objekt `s` zurück, sodass für das Ergebnis der Ausgabe wiederum `s` eingesetzt gedacht werden kann und eine Verkettung mehrerer Ausgaben möglich wird:

```
std::cout << "r = " << r << "r1 = " << r1;
```

ist dann identisch mit

```
((std::cout << "r = ") << r) << "r1 = " << r1;
```

Die Implementierung des Ausgabeoperators ähnelt der vorher benutzten Elementfunktion `ausgabe()`:

```
ostream& operator<<(ostream& ausgabe, const Rational& r) {
    ausgabe << r.getZaehler() << '/' << r.getNenner();
    return ausgabe;
}
```

Zum Abschluss wird die Definition der Klasse mit den bisher besprochenen Änderungen gezeigt, jedoch ohne die Implementierung der Methoden und Funktionen (siehe oben sowie Übungsaufgaben). Durch Operatoren ersetzte Elementfunktionen wie `add()` usw. wurden gestrichen.

Listing 9.1: Klasse `Rational` mit überladenen Operatoren

```
// cppbuch/k9/rational/ratioop.h
#ifndef RATIOOP_H
#define RATIOOP_H
#include<iostream>

class Rational {
public:
    Rational() : zaehler(1), nenner(1) {} // inline
    Rational(long z, long n);
    Rational(long); // Typumwandlungskonstruktor

    // Abfragen inline
    long getZaehler() const { return zaehler; }
    long getNenner() const { return nenner; }
```

```

// arithmetische Methoden
Rational& operator+=(const Rational&);
// weitere arithmetische Methoden weggelassen ...

// weitere Methoden
void set(long zaehler, long nenner);
void kehrwert();
void kuerzen();

private:
    long zaehler, nenner;
};

// globaler Additionsoperator (andere Operatoren siehe Übungsaufgabe)
const Rational operator+(const Rational&, const Rational&);

// globale Funktionen zur Ein- und Ausgabe
std::ostream& operator<<(std::ostream&, const Rational&);
std::istream& operator>>(std::istream&, Rational&);
#endif // RATIOOP_H

```



Übungen

9.1 Implementieren Sie den Operator `>>` zur Eingabe von Objekten der Klasse `Rational`.

9.2 Implementieren Sie den Operator `+=` für rationale Zahlen. Überlegen Sie vorher, ob der Operator als Element- oder als `friend`-Funktion realisiert werden sollte. Der Rückgabetyt könnte bei allein stehenden Operationen `void` sein. Aus dem Programmcode innerhalb `operator+()` auf Seite 321 ergibt sich aber ein Rückgabetyt `Rational` oder `Rational&`, der in der Klassendefinition oben verwendet wird. Welcher ist sinnvoll und warum?

9.3 Ersetzen Sie im Beispiel des Abschnitts 4.4 auch die Funktionen `sub()`, `mult()` und `div()` durch überladene Operatoren. Testen nicht vergessen!

9.4 Implementieren Sie den Operator `==` zum Vergleich zweier Rationalzahlen.

9.2 Eine Klasse für Vektoren

In den folgenden Abschnitten wird eine Klasse `Vektor` definiert, die ein eindimensionales Array wählbarer Größe bildet. Die Klasse soll zeigen, wie die Klasse `vector` der C++-Standardbibliothek im Prinzip arbeitet. Die Unterschiede zu einem C-Array sind:

- Der Arrayzugriff über den Index-Operator `[ ]` ist *sicher* (im Gegensatz zur Klasse `vector` der C++-Standardbibliothek, die geprüfte Zugriffe über die Methode `at()` anbietet). Eine Indexüberschreitung wird zur Laufzeit als Fehler gemeldet, es wird keine undefinierte Adresse angesprungen.

- Es wird ein Zuweisungsoperator definiert, sodass $v1 = v2$; geschrieben werden kann mit dem Ergebnis $v1[i] == v2[i]$ für alle i im Bereich 0 bis (Länge von $v2 - 1$).
- Die Klasse kann leicht um weitere Funktionen erweitert werden wie Bildung des Skalarprodukts, Addition zweier Vektoren und andere. Hier wird gezeigt, wie die Größe eines Vektors dynamisch, also zur Laufzeit, geändert werden kann. Er verhält sich dann wie ein dynamisches Array.

Die private Elemente der Klasse sind ein Zeiger `start` auf den Beginn eines Arrays der Größe `xDim`, das aus Objekten des Typs `T` besteht (Abbildung 9.2).

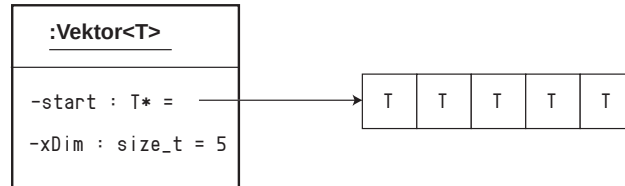


Abbildung 9.2: Ein Objekt der Klasse `Vektor<T>`

Listing 9.2: Template-Klasse `Vektor`

```

// cppbuch/k9/vektor/vektor.t
// dynamische Vektor-Klasse
#ifndef VEKTOR_T
#define VEKTOR_T
#include<stdexcept>

template<typename T>
class Vektor {
public:
    Vektor(size_t x = 0);           // Allg. Konstruktor
    Vektor(size_t n, T t);         // Allg. Konstruktor 2:
                                   // n Elemente mit Wert t
    Vektor(const Vektor<T>& v);     // Kopierkonstruktor
    virtual ~Vektor() { delete [] start; } // Destruktor

    size_t size() const { return xDim; }
    void groesseAendern(size_t);    // dynamisch ändern

    // Indexoperator inline
    T& operator[](int index) {
        if( index < 0 || index >= (int)xDim) {
            throw std::out_of_range("Indexüberschreitung!");
        }
        return start[index];
    }
    // Indexoperator für nicht-veränderliche Vektoren
    const T& operator[](int index) const {
        if( index < 0 || index >= (int)xDim) {
            throw std::out_of_range("Indexüberschreitung!");
        }
    }
}
  
```

```

        return start[index];
    }

    void swap(Vektor<T>& v); // Vektoren vertauschen
    // Zuweisungsoperator
    Vektor<T>& operator=(Vektor<T>); // Übergabe per Wert (siehe Impl. unten)

    // Zeiger auf Anfang und Position nach dem Ende für Vektoren
    // mit nichtkonstanten und konstanten Elementen
    T* begin() { return start; }
    T* end() { return start + xDim; }
    const T* begin() const { return start; }
    const T* end() const { return start + xDim; }
private:
    size_t xDim; // Anzahl der Datenobjekte
    T *start; // Zeiger auf Datenobjekte
}; // #endif siehe Seite 330

```

Die Elemente des Arrays können beliebige kopierbare Objekte desselben Datentyps sein, zum Beispiel `Rational`-Objekte. Die Klassendeklaration enthält auch die Dinge, die erst in den späteren Abschnitten über die verschiedenen Operatoren benötigt werden.

Es ist unmittelbar einsichtig, dass Programme mit Vektoren bei Vorliegen der oben beschriebenen Funktionalität lesbarer und kürzer als mit C-Arrays geschrieben werden können – weswegen in diesem Buch von Anfang an Standardvektoren verwendet werden. Das Prinzip gilt natürlich nicht nur für Vektoren. Die anwendungsbezogene C++-Programmierung wird darauf hinauslaufen, eine Vielfalt vorgefertigter Klassen aus kommerziell erhältlichen und firmenspezifischen Bibliotheken einzusetzen, um die Produktivität der Programmierer und die Qualität ihrer Produkte zu erhöhen.

Die angegebenen Elementfunktionen und Operatoren werden nachfolgend erläutert. Es folgen die Implementationen der Konstruktoren:

```

template<typename T>
Vektor<T>::Vektor(size_t x) // Allg. Konstruktor
: xDim(x), start(new T[x]) {
}

```

Der Konstruktor erzeugt zum Beispiel durch `Vektor<Rational> vRat(3);` einen `Rational`-Vektor `vRat` mit drei Elementen. Weil im Konstruktor mit `new` ein Array angelegt wird, muss im Destruktor `delete` mit eckigen Klammern angegeben werden. Der Kopier- oder Initialisierkonstruktor kopiert erst die Größe des Vektors, beschafft die benötigte Menge Speicherplatz und überträgt dann einzeln die Elemente:

```

template<typename T>
Vektor<T>::Vektor(const Vektor<T>& v) // Kopierkonstruktor
: xDim(v.xDim), start(new T[xDim]) {
    for(size_t i = 0; i < xDim; ++i) {
        start[i] = v.start[i];
    }
}

```

Man könnte fragen, ob nicht die C-Funktion `memcpy()` von Seite 881 das Kopieren des Vektors schneller als die gezeigte Schleife erledigen könnte, da der Speicherplatz für

das Array start zusammenhängend angelegt wird. Dies würde jedoch nur funktionieren, wenn der Typ `T` keine dynamischen Daten enthält, weil es sich um eine »flache« Kopie handelt (siehe auch Abbildung 6.2 auf Seite 236). Im Template ist aber nicht bekannt, ob es nur für solche Typen instanziiert werden wird. Der in Abschnitt 9.2.2 beschriebene Zuweisungsoperator stellt hingegen sicher, dass die Anweisung `start[i] = v.start[i]`, eine vollständige Kopie des Vektor-Elements bewirkt.

Der vom System bereitgestellte Kopierkonstruktor wäre nicht ausreichend, weil ein Kopieren von `start` und `xDim` nicht genügt. Der Konstruktor zur Initialisierung des Vektors mit einem bestimmten Wert enthält eine einfache Schleife, die jedem Element des Vektors den Initialisierungswert zuweist:

```
template<typename T>
Vektor<T>::Vektor(size_t n, T wert)
: xDim(n), start(new T[n]) {
    for (size_t i = 0; i < xDim; ++i) {
        start[i] = wert;
    }
}
```

Die Methode `begin()` gibt einen Zeiger auf den Anfang des Vektors zurück; `end()` gibt einen Zeiger auf die Position *nach dem letzten* Vektorelement zurück. Diese Methoden werden erst später benötigt (Abschnitt 11.2).

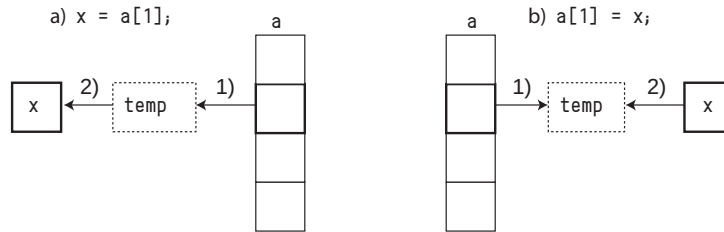
9.2.1 Index-Operator []

Die Deklaration des Indexoperators zeigt, dass nicht ein Objekt, sondern eine *Referenz* auf ein Objekt des Datentyps `T` zurückgegeben wird. Warum? Gesucht ist eigentlich *ein Element* des Vektors. Betrachten wir dazu einfach die beiden Fälle (Rückgabe eines Objekts per Wert – Rückgabe einer Referenz) im Vergleich.

Falsch: Rückgabe per Wert

Die Deklaration würde dementsprechend lauten: `T operator[](int index);`. Da die Operatorfunktion eine Funktion wie jede andere ist, die nur einen speziellen Namen hat, gelten die gleichen Mechanismen wie sonst auch bei Funktionen. Das per Wert übergebene Argument `index` wird in die Funktion hineinkopiert, und die Anweisung `return irgendwas;` bewirkt, dass eine temporäre Kopie von `irgendwas` dem Aufrufer übergeben wird. Das Kopieren wird vom compilergenerierten Kopierkonstruktor oder einem für die Klasse `T` selbst geschriebenen übernommen. Abbildung 9.3 verdeutlicht den Vorgang. Wir müssen dabei zwei verschiedene Fälle unterscheiden, nämlich dass das anzusprechende Element auf der rechten oder auf der linken Seite (als L-Wert oder lvalue) einer Zuweisung stehen kann.

Betrachten wir den ersten Fall, in Abbildung 9.3 mit a) bezeichnet. Im ersten Schritt wird durch den Indexoperator eine temporäre Kopie des Elements Nr. 1 des Vektors `a` erzeugt. Diese Kopie wird der Variablen `x` zugewiesen, und anschließend wird die Kopie verworfen – genau wie wir es wünschen. Im zweiten Fall sieht es anders aus! Wie in Abbildung 9.3, Fall b), zu sehen ist, würde auch hier zunächst eine temporäre Kopie erzeugt. Die Variable `x` würde dann *dieser Kopie* zugewiesen werden, die danach »entsorgt« wird, *das ursprüngliche Vektorelement bliebe unverändert!* Damit wäre der Sinn der Anweisung



Reihenfolge: 1) Erzeugen einer temporären Kopie, 2) Zuweisung an x bzw. von x

Abbildung 9.3: Falsch: Zuweisung von Vektorelementen per Wert

$a[1] = x;$ verfehlt. Glücklicherweise werden wir vom Compiler beim Auftreten dieser Anweisung darauf aufmerksam gemacht, dass der Operator nicht die ihm zugedachte Aufgabe erfüllt.

Richtig: Rückgabe per Referenz

Die Deklaration enthält nun den richtigen Rückgabetypp, sodass das richtige Objekt zurückgegeben wird:

```
T& operator[](int index);
```

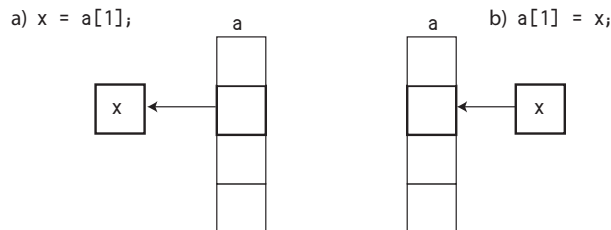


Abbildung 9.4: Richtig: Zuweisung von Vektorelementen per Referenz

Abbildung 9.4 zeigt in a) und b) die gleichen Fälle wie Abbildung 9.3, nur dass jetzt nicht mit einer temporären Kopie, sondern *direkt mit dem Original* gearbeitet wird. Eine Referenz ist ja nichts anderes als ein anderer Name, eine Alias-Bezeichnung, für ein und dasselbe Objekt. Der Kopierkonstruktor wird bei der Ergebnissrückgabe nicht aufgerufen, weil nichts zu kopieren ist und nur der Verweis auf das Original zurückgegeben wird. Das Ergebnis des Indexoperators ist nun das gewünschte Vektorelement und nicht nur eine temporäre Kopie.

Die Implementierung des Indexoperators wurde `inline` schon in der Klassendeklaration vorgenommen. Dabei wurde der Weg gewählt, dass das Programm sich mit einer Fehlermeldung beendet, wenn der zulässige Bereich für den Index über- oder unterschritten wird. Würden dem Aufrufer stattdessen ohne Abbruch irgendwelche Daten zurückgegeben, so arbeitete das Programm anschließend mit falschen Daten! Dadurch entstehende Folgefehler sind oft nur schwer auf die eigentliche Ursache der Indexüberschreitung zurückzuführen. Nach Auslieferung des Programms an einen Kunden sollte das Programm

allerdings *weder* abbrechen *noch* mit falschen Daten weiterrechnen (zur Fehlerbehandlung siehe Kapitel 8). Der zweite Indexoperator ist für den Zugriff auf konstante Objekte gedacht. Das erste `const` garantiert, dass ein per Referenz zurückgegebenes Element nicht verändert wird; das zweite `const` erlaubt die Anwendung des Operators auf konstante Vektor-Objekte ohne Beschwerden des Compilers. Wenn nun der Zugriff auf Arrayelemente ausschließlich über den Indexoperator geschieht, wird jede Bereichsüberschreitung gemeldet. Die nächsten Zeilen zeigen, wie der Operator in einem Programm eingesetzt wird.

```
Vektor<double> v(5);           // Vektor mit 5 Elementen
v[0] = 1.00; v[1] = 1.234; v[2] = 2.22; // Elemente als Linkswerte
double x = v[2];              // Element rechts

// Test auf Indexüberschreitung
v[99] = 400.2; // Fehlermeldung zur Laufzeit!
x = v[100];    // Fehlermeldung zur Laufzeit!

// Definition und Initialisierung eines anderen Vektors
Vektor<double> vc = v;      // mit Kopierkonstruktor

// Element links und rechts
v[0] = vc[1];
```

Die Schreibweise unterscheidet sich in nichts von der Benutzung eines normalen Arrays. Nach außen hin ist die Wirkung die gleiche bis auf das sichere Verhalten bei Indexfehlern. Der sichere Zugriff kann bei dieser Vektor-Klasse mittels Zeigerarithmetik umgangen werden:

```
double *vp = &v[0];
*(vp+100) = 17.9;      // nicht entdeckter Indexfehler!
```

Gegen böswillige Manipulationen kann man sich ohnehin nicht vollständig schützen, aber die konsequente Benutzung eines wie oben definierten Indexoperators anstelle der Zeigerarithmetik trägt erheblich zur Sicherheit und Testbarkeit eines Programms bei.

9.2.2 Zuweisungsoperator =

Wenn ein spezieller Zuweisungsoperator nicht definiert ist, wird automatisch vom Compiler ein Zuweisungsoperator bereitgestellt (implizit generiert), der ein Objekt elementweise kopiert. Darunter ist zu verstehen, dass für jedes Element, das selbst wieder Objekt oder aber von einem Grunddatentyp sein kann, der zugehörige Zuweisungsoperator aufgerufen wird. Der Zuweisungsoperator für Grunddatentypen bewirkt eine bitweise Kopie. Falls Konstanten oder Referenzen vorhanden sind, geschieht dies jedoch nicht, weil sie nur initialisiert, aber nicht durch Zuweisung verändert werden könnten. In diesen Fällen ist ein besonderer '='-Operator notwendig. Im folgenden Beispiel sei die Existenz zweier Klassen A und B angenommen, die in der Klasse C benutzt werden:

```
class C : public A {
public:
    // ...
private:
    B einB;
```

```
    int x;
};
```

Eine Zuweisung der Art

```
C c1, c2;
c1 = c2; // d.h. c1.operator=(c2);
```

bewirkt implizit die folgenden Aufrufe, wobei keine Aussage darüber getroffen wird, ob der Zuweisungsoperator vom System generiert oder selbst geschrieben wurde:

```
c1.A::operator=(c2.A); // Zuweisen des anonymen Subobjekts
c1.einB.operator=(c2.einB); // aggregiertes Objekt
c1.x = c2.x;           // Grunddatentyp
```

Wenn die Klassen A bzw. B selbst Elemente enthalten, gilt für diese dasselbe Prinzip.

Operatorfunktionen können wie andere Methoden vererbt werden. Die Ausnahme ist der Zuweisungsoperator '='. Der automatisch erzeugte Zuweisungsoperator würde ohnehin einen geerbten Zuweisungsoperator überschreiben. Aus diesem Grund ist ein Vererben des Zuweisungsoperators überflüssig. Typischerweise enthält eine abgeleitete Klasse zusätzliche Datenelemente. Die Restriktion sorgt dafür, dass nicht versehentlich die zusätzlichen Datenelemente beim Kopieren unter den Tisch fallen können.

Falls Zeiger als Elemente in einem Objekt vorhanden sind, werden diese ebenfalls kopiert. Dies kann zu Problemen führen, wenn die Zeiger auf dynamisch zugewiesene Speicherbereiche oder Objekte zeigen, weil dann nur Adressen kopiert werden (siehe dazu Abschnitt 4.3.3 und das Bild auf Seite 236). Wenn ein Objekt A einem Objekt B zugewiesen werden soll, ist folgende Reihenfolge einzuhalten:

1. Reservieren von Speicher für B in der Größe von Objekt A
2. Inhalt von A in den neuen Speicher kopieren
3. Freigabe des vorher belegten Speichers
4. Aktualisieren der Verwaltungsinformationen

Den 3. Schritt vor den ersten zu ziehen, um den gesamten Speicherbedarf zu reduzieren, wäre ein Fehler: Das Programm wäre nicht mehr exception-sicher (siehe dazu Abschnitt 8.3). Aber wenn wir den Swap-Trick von Seite 218 anwenden, können wir uns die Überlegungen über die Speicherbeschaffung und die Reihenfolge schenken: Der Zuweisungsoperator ist dann auf jeden Fall exception-sicher und auch viel einfacher zu schreiben. Er benutzt die Funktion `swap(Vektor<T>&)` der Klasse `Vektor`, die wiederum die Funktion `std::swap()` der Standardbibliothek nutzt. Sie vertauscht einfach nur ihre beiden Parameter.

```
// Vektoren vertauschen
template<typename T>
void Vektor<T>::swap(Vektor<T>& v) {
    std::swap(xDim, v.xDim);
    std::swap(start, v.start);
}
```

```
// Zuweisungsoperator
template<typename T>
Vektor<T>& Vektor<T>::operator=(Vektor<T> v) { // Übergabe per Wert! (siehe Text)
    swap(v);
    return *this;
}
```

Die Übergabe per Wert erspart eine Zeile zur Konstruktion eines temporären Objekts. Außerdem hat der Compiler so die Möglichkeit, den Kopierkonstruktor wegzuoptimieren – nämlich dann, wenn auf der rechten Seite der Zuweisung ein temporäres Objekt steht, etwa bei der Zuweisung eines Additionsergebnisses: $v1 = v2 + v3$.

Damit können Vektoren mit *einem* Befehl im Programm zugewiesen werden, zum Beispiel $v2 = v$. Bisher bestehende Referenzen auf Elemente des linksseitigen Vektors werden bedeutungslos, weil der Adressbereich des Vektors nach der `new`-Operation im Kopierkonstruktor ganz woanders liegen kann! Eine syntaktisch mögliche Zuweisung auf sich selbst (zum Beispiel $v1 = v1$) wird wohl kaum jemand vornehmen – wenn doch, gäbe es nur einen leichten Performance-Verlust wegen der überflüssigen Kopie. Aufgrund dieser Überlegungen ergibt sich der folgende



Tipp

Verwenden Sie diese Form des Zuweisungsoperators für Objekte mit dynamischen Anteilen (mit `new` erzeugt). In den meisten anderen Fällen wird gar kein eigener Zuweisungsoperator gebraucht. Wenn doch, ist dann die Übergabe per `const&` vorzuziehen. Dieser Tipp gilt nicht für Zuweisungsoperatoren einer Vererbungshierarchie (vgl. Abschnitt 9.9).

Dynamisches Ändern der Vektorgröße

Nachdem die Wirkungsweisen von Kopierkonstruktor und Zuweisungsoperator bekannt sind, macht es nun keine Schwierigkeiten, eine Methode zum Ändern der Größe eines Vektors zu schreiben:

```
template<typename T>
void Vektor<T>::groesseAendern(size_t neueGroesse) {
    T *pTemp = new T[neueGroesse]; // neuen Speicherplatz besorgen
    // die richtige Anzahl von Elementen kopieren
    size_t kleinereZahl = neueGroesse > xDim ? xDim : neueGroesse;
    for(size_t i = 0; i < kleinereZahl; ++i) {
        pTemp[i] = start[i];
    }
    delete [] start; // alten Speicherplatz freigeben
    start = pTemp; // Verwaltungsdaten aktualisieren
    xDim = neueGroesse;
}
#endif // VEKTOR_T
```



Übungen

9.5 Schreiben Sie einen Zuweisungsoperator für die Klasse `NummeriertesObjekt` aus Abschnitt 6.2. Überlegen Sie vorher genau, was er eigentlich tun soll.

9.6 Schreiben Sie für die Klasse `MeinString` aus Abschnitt 6.1.1 Operatoren, die den Methoden `assign()`, `at()` und `ausgabe()` entsprechen.

9.7 Schreiben Sie für die Klasse `MeinString` die Operatoren `+=` und `+` zum Verketten von `MeinString`-Objekten.



Hinweis

Bei der Konstruktion des Zuweisungsoperators in einer Vererbungshierarchie sind besondere Bedingungen zu beachten. Ab Seite 365 wird darauf eingegangen.

9.2.3 Mathematische Vektoren

Die oben beschriebene Klasse `Vektor` kann beliebige kopierbare Objekte eines vorgegebenen Typs aufnehmen. Nun konstruieren wir uns eine maßgeschneiderte Klasse `MathVektor` für Objekte der Datentypen `int`, `float`, `complex`, `Rational` etc., die alle Eigenschaften der Klasse `Vektor` hat und noch ein paar mehr, nämlich dass mathematische Operationen mit Objekten der Klasse einfach möglich sind. Alternativ gibt es die Klasse `valarray` der C++-Standardbibliothek (siehe Seite 857 ff). Sie ist eine Klasse speziell für numerische Arrays und den mathematischen Operationen dazu, nur ist sie um einiges komplizierter. `MathVektor` ist ein Vektor, und diese Beziehung drückt sich in C++ durch Vererbung aus. Daher wird `MathVektor` von der Klasse `Vektor` des vorhergehenden Abschnitts abgeleitet. Normalerweise soll die Klasse `Vektor` nur einen Einblick in die Wirkungsweise einer Template-Klasse für Vektoren geben, nicht aber zum Standardwerkzeug werden – dafür ist im Allgemeinen die Klasse `vector` der C++-Standardbibliothek besser geeignet. Hier wird davon abgewichen, weil der Indexoperator den übergebenen Index prüfen soll. Als Beispiel für mathematische Operatoren sei hier nur der `*=`-Operator gezeigt.

Listing 9.3: Klasse für Vektoren mit mathematischen Operationen

```
// cppbuch/k9/mathvek/mvektor.t
#ifndef MVEKTOR_T
#define MVEKTOR_T
#include<iostream>
#include "../vektor/vektor.t"

template<typename T>
class MathVektor : public Vektor<T> {
public:
    typedef Vektor<T> super; // Abkürzung für Oberklassentyp
    MathVektor(size_t=0);
    MathVektor(size_t n, T wert);
    void init(const T&); // alle Elemente setzen
    MathVektor& operator*=(const T&); // Operator *=
    // weitere Operatoren und Funktionen ...
}; // #endif folgt am Dateiende (nach der Implementierung)
```

Die Implementation der Konstruktoren ist entsprechend einfach. Sie initialisieren nur ein Basisklassensubobjekt vom Typ `Vektor`, sonst ist nichts weiter zu tun.


```

template<typename T>
MathVektor<T>::MathVektor(size_t x)
: Vektor<T>(x) {
}

template<typename T>
MathVektor<T>::MathVektor(size_t n, T wert)
: Vektor<T>(n, wert) {
}

template<typename T>
void MathVektor<T>::init(const T& wert) {
    for(size_t i = 0; i < super::size(); ++i) {
        super::operator[](i) = wert;
    }
}

```

Weil `MathVektor` keinerlei Attribute außer den geerbten hat, sind weder Destruktor, noch Kopierkonstruktor oder Zuweisungsoperator notwendig! Die vom System implizit generierten Funktionen Destruktor, Kopierkonstruktor und Zuweisungsoperator genügen, weil die entsprechenden Operationen für das Basisklassensubobjekt von der Klasse `Vektor` erledigt werden. Der Indexoperator wird von der Basisklasse geerbt, sodass die Klassendeklaration dadurch sehr einfach wird.

Bei dieser Diskussion wird vorausgesetzt, dass in einem Programm – wie in den Beispielen oben – Methoden über den Namen eines definierten Objekts aufgerufen werden, nicht über Referenzen oder Zeiger. Polymorphismus wird nicht betrachtet. Polymorphes Verhalten ist bei vektorähnlichen Objekten nicht üblich. Sollte es dennoch gewünscht sein, müsste der Zuweisungsoperator weiteren Bedingungen genügen, die in Abschnitt 9.9 erläutert werden. Auch wäre dann die Klasse `vector` als Oberklasse nicht geeignet, weil sie keinen virtuellen Konstruktor hat.

9.2.4 Multiplikationsoperator

Dieser Operator soll auf einfache Weise einen Vektor mit einem Skalar multiplizieren, indem zum Beispiel `v1 *= 1.23;` geschrieben wird. Der Operator muss dafür sorgen, dass alle Elemente von `v1` mit 1.23 multipliziert werden.

```

template<typename T>
MathVektor<T>& MathVektor<T>::operator*=(const T& zahl) {
    for(size_t i = 0; i < super::size(); ++i) {
        super::operator[](i) *= zahl; // elementweise Multiplikation
    }
    return *this;
}

```

Vielleicht fragen Sie sich, was die Angabe von `super::` soll, wo doch `MathVektor` von `vector` abgeleitet ist und `operator[]` bekannt sein sollte? Die Antwort: Es gibt beim Aufruf von `operator[]` keine Argumente, die von einem Template-Parameter abhängen. Damit ist eine Instanziierung der Funktion `operator[]` bezogen auf `T` nicht möglich. Der Compiler würde ohne `super::` eine Deklaration von `operator[]` vermissen. Mit `super::` ist jedoch klar, dass `operator[]` ein Element der Oberklasse ist; dort existiert die Deklaration. Natur-

lich könnte auch `Vektor<T>::operator[](i)` geschrieben werden. Eine weitere Alternative wäre die Verwendung von `this`, zum Beispiel

```
typedef Vektor<T> super;
// Verwendung:
super::operator[](i) *= zahl;      // gleichbedeutend mit:
Vektor<T>::operator[](i) *= zahl;  // gleichbedeutend mit:
this->operator[](i) *= zahl;       // gleichbedeutend mit:
(*this)[i] *= zahl;
```

Jetzt kann die Multiplikation eines beliebig großen Vektors mit einem Skalar sehr einfach geschrieben werden, zum Beispiel:

```
MathVektor<double> v1(100), v2(200);
//...
v1 *= 1.234567; // alle Elemente von v1 multiplizieren
```

Aber was ist mit Fällen wie `v1 = v2 * 1.234567`; oder `v1 = 1.234567 * v2`;, in denen drei Beteiligte anstatt nur zwei vorhanden sind? Am Ende des Kapitels 9.1 wurde darauf hingewiesen, dass ein binärer Operator mithilfe des Kurzformoperators implementiert werden kann. Dabei müssen die binären Operatoren nicht einmal Elementfunktionen der Klasse sein, sofern sie nicht direkt auf private Daten zugreifen. Die Implementierung für die beiden Fälle ergibt in Analogie zum Vorschlag am Ende von Abschnitt 9.1:

```
// * Operator für den Fall v1 = zahl*v2;
template<typename T>
MathVektor<T> operator*(const T& zahl, MathVektor<T> mv) {
    return mv *= zahl; // Aufruf von operator*=()
}

// * Operator für den Fall v1 = v2*zahl; (vertauschte Reihenfolge der Argumente)
template<typename T>
MathVektor<T> operator*(const MathVektor<T>& v, const T& zahl) {
    return zahl*v; // Aufruf des obigen operator*()
}
```

Auf die Erweiterung mit zusätzlichen mathematischen oder anderen Operatoren wird nicht weiter eingegangen. Man beachte übrigens, dass sich die Länge eines Vektors durch Zuweisung ändern kann. In dem Beispiel

```
v1 = 1.234567 * v2;
```

spielt die Länge von `v1` vor der Zuweisung keine Rolle, und nach der Zuweisung haben `v1` und `v2` dieselbe Anzahl von Elementen. Dafür sorgt der Zuweisungsoperator der Klasse `Vektor`. Dieser Operator ist nicht vererbbar, er wird hier bei der elementweisen Kopie der Elemente eines `MathVektors` wirksam, d.h. bei der Kopie des Basisklassensubjekts.

9.3 Inkrement-Operator ++

Der Inkrement-Operator ++ tritt in der Präfix- oder Postfix-Form auf, je nachdem, ob das Hochzählen vor oder nach der Verwendung einer Variablen geschehen soll. Um nicht nur ein ganz triviales Erhöhen um eins zu zeigen, wird eine Klasse `Datum` vorgestellt, in der das Hochzählen mit ++ das Weiterschalten des Tages auf das nächste Datum bedeutet. Dabei müssen Monatsübergänge, Schaltjahre und mehr berücksichtigt werden.

Nach Tabelle 9.1 (Seite 318) ist eine Unterscheidung von Präfix- und Postfix-Version durch ein Argument 0 bei der letzteren realisiert. Aus diesem Grund wird der Postfix-Inkrementoperator von der Präfix-Version durch einen zusätzlichen `int`-Parameter im Argument unterschieden, obwohl der Parameter sonst nicht weiter benötigt wird – ein vielleicht nicht besonders eleganter Kunstgriff. Um bereits *vor* der Konstruktion eines Datums Tag, Monat und Jahr überprüfen zu können, zum Beispiel nach einer Dialogeingabe, wird eine globale Funktion zur Überprüfung bereitgestellt. Die Funktion wird auch innerhalb der Methoden der Klasse benutzt. Sie benutzt die globale Funktion `istSchaltjahr()`, die auch von der Methode `Datum::istSchaltjahr()` gerufen wird.

Listing 9.4: Klasse `Datum`

```
// cppbuch/k9/datum/datum.h
#ifndef DATUM_H
#define DATUM_H

class Datum {
public:
    Datum(); // Standardkonstruktor
    Datum(int t, int m, int j); // allgemeiner Konstruktor
    void set(int t, int m, int j); // Datum setzen
    void aktuell(); // Systemdatum setzen
    bool istSchaltjahr() const;
    Datum& operator++(); // Tag hochzählen, präfix
    Datum operator++(int); // Tag hochzählen, postfix
    int tag() const; // lesende Methoden
    int monat() const;
    int jahr() const;
private:
    int tag_, monat_, jahr_;
};

// Prototyp der globalen Schaltjahr-Funktion
bool istSchaltjahr(int jahr); // Implementation s.u.
// inline Methoden
// Um Code-Duplikation zu vermeiden, wird set() aufgerufen
inline Datum::Datum(int t, int m, int j) { set(t, m, j); }
inline int Datum::tag() const { return tag_; }
inline int Datum::monat() const { return monat_; }
inline int Datum::jahr() const { return jahr_; }
inline bool Datum::istSchaltjahr() const {
    return ::istSchaltjahr(jahr_); // globale Funktion mit ::-Operator
}
```

```
// globale Funktionen
bool istGueltigesDatum(int t, int m, int j);

inline bool istSchaltjahr(int jahr) {
    return (jahr % 4 == 0 && jahr % 100) || jahr % 400 == 0;
}
#endif // DATUM_H
```

Implementierung zur Klasse Datum

Die folgend beschriebene Implementierung der Operatormethoden ist in der Datei *cpp-buch/k9/datum/datum.cpp* der Beispiele zu finden. Der Standardkonstruktor initialisiert das Datum mit dem Systemdatum:

```
Datum::Datum() {
    aktuell(); // siehe unten
}

void Datum::set(int t, int m, int j) { // Datum neu setzen
    // Damit nur korrekte Datum-Objekte möglich sind:
    if(!istGueltigesDatum(t, m, j)) {
        // siehe Aufgabe am Ende des Abschnitts
    }
    tag_ = t;
    monat_ = m;
    jahr_ = j;
}
```

In der Funktion `aktuell()` treten vordefinierte Datentypen und Funktionen auf, die auf jedem C/C++-System zu finden sind (Einzelheiten siehe Seite 881). Die Datentypen und Funktionen `time_t`, `time()`, `tm` und `localtime()` sind in `<ctime>` deklariert, `time_t` beispielsweise als `long`. `time()` liefert die Anzahl der Sekunden, die seit dem 1. Januar 1970 bis zum Aufruf verstrichen sind, als Zahl vom Typ `time_t`. Die Funktion `localtime()` gibt einen Zeiger auf eine statische Struktur des Typs `tm` zurück, die ausgewertet wird.

```
// Systemdatum eintragen
void Datum::aktuell() {
    time_t now = time(NULL); // aktuelle Zeit in s seit 1.1.1970
    tm *z = localtime(&now); // Zeiger auf eine vordefinierte Struktur des Typs tm.
                                // Umwandlung mit localtime().
    jahr_ = z->tm_year + 1900;
    monat_ = z->tm_mon + 1; // localtime() liefert 0..11
    tag_ = z->tm_mday;
}
```

Die folgende Methode dient schaltet den Tag weiter. Falls ein ungültiges Datum wie der 31. April gebildet wird, muss es sich um den Übergang zum nächsten Monat handeln:

```
Datum& Datum::operator++() { // Präfix (kein int-Argument)
    ++tag_;
    // Monatsende erreicht?
    if(!istGueltigesDatum(tag_, monat_, jahr_)) {
        tag_ = 1;
    }
}
```

```

    if (++monat_ > 12) {
        monat_ = 1;
        ++jahr_;
    }
}
return *this;
}

```

Der Postfix-Inkrementierungsoperator folgt einem Muster, das für alle nachgestellten ++-Operatoren gültig ist. Zuerst wird der alte Zustand des Objekts »gerettet«, dann wird das Objekt verändert und der gerettete Wert zurückgegeben:

```

Datum Datum::operator++(int) { // Datum um 1 Tag erhöhen
                               // Parameter wird nicht gebraucht
    Datum temp(*this);
    ***this;                  // Präfix ++ aufrufen
    return temp;
}

```

Anstelle von ***this; kann alternativ der Aufruf operator++(); stehen. Die Angabe eines Variablennamens in der Parameterliste des Postfix-Operators ist nicht notwendig, weil die Variable nicht benötigt wird. Die Implementierung des Postfix-Operators ändert den Wert des Objekts mithilfe des Präfix-Operators, sodass eine Code-Duplikation vermieden wird, gibt aber mithilfe der Variablen temp den vorherigen Wert entsprechend der gewünschten Semantik zurück. Der Aufruf in einem Programm wird wie folgt interpretiert:

```

++dat1;          Aufruf wie dat1.operator++();
dat1++;          Aufruf wie dat1.operator++(0);

```

Damit ist das Rüstzeug zur Implementierung eigener Inkrement-Operatoren vorhanden. Die verzögerte Objektänderung in der Postfix-Form muss allerdings, wie gezeigt, selbst programmiert werden, sie wirkt nicht automatisch. Manchmal ist es von der Semantik her gleichgültig, ob ein Prä- oder Postfix-Operator genommen wird, wie etwa bei einer allein stehenden Anweisung x++; für ein beliebiges Objekt x. Jedoch kann eine temporäre Variable beim Postfix-Operator nicht vermieden werden, sodass in solchen Fällen aus Effizienzgründen ein Präfix-Operator vorzuziehen ist, das heißt, ++x; ist besser. Alles in diesem Abschnitt über den Inkrement-Operator Gesagte gilt entsprechend natürlich auch für den Dekrement-Operator --.

Zum Schluss folgt die globale Funktion zur Überprüfung, ob ein Tag/Monat/Jahr-Tripel ein gültiges Datum darstellt. Die Jahreszahl soll größer als 1582 sein, weil in dem Jahr der bis heute gültige Gregorianische Kalender eingeführt worden war.

```

bool istGueltigesDatum(int t, int m, int j) {
    // Tage pro Monat (static vermeidet Neuinitialisierung):
    static int tmp[]={31,28,31,30,31,30,31,31,30,31,30,31};
    int letzterTagImMonat = tmp[m-1];
    if(m == 2 && istSchaltjahr(j)) {
        letzterTagImMonat = 29;
    }
    return ( m >= 1 && m <= 12 && j >= 1583 && j <= 2399 // oder mehr
            && t >= 1 && t <= letzterTagImMonat);
}

```



Übungen

9.8 Der Rückgabetyt der Präfix-Form des Inkrementoperators ist als Referenz formuliert. Warum?

9.9 Wäre eine Referenz als Rückgabetyt in der Postfix-Form sinnvoll?

9.10 Schreiben Sie einen Ausgabeoperator << zur Ausgabe von Datum-Objekten in der üblichen Schreibweise, d.h. Tag, Monat und Jahr durch einen Punkt getrennt (zum Beispiel 31.12.2011).

9.11 Implementieren Sie die relationalen Operatoren ==, != und < für Objekte der Klasse Datum. Letzterer gibt an, welches Datum von zweien früher liegt.

9.12 Die Differenz zweier Datum-Objekte in Tagen soll durch eine globale Funktion mit der Schnittstelle `int DatumDifferenz(const Datum& a, const Datum& b)` berechnet werden.

9.13 Berechnen Sie, welches maximale Systemdatum auf Ihrem System möglich ist. Hinweis: siehe die obige Methode `aktuell()`.

9.14 Die Methode `set()` von Seite 334 soll eine von Ihnen geschriebene `UngueltigesDatumException` werfen, wenn das Datum nicht korrekt ist. `UngueltigesDatumException` soll von `runtime_error` erben. Prüfen Sie das Verhalten mit einem Testprogramm, indem Sie ein falsches Datum erzeugen, zum Beispiel

```
Datum einDatum;
try {
    einDatum.set(30, 2, 2012);
}
catch(const UngueltigesDatumException& e) {
    cout << e.what() << endl;
}
```

Die Ausgabe des Programms soll zum Beispiel lauten: »30.2.2012 ist ein ungültiges Datum« oder ähnlich.

9.4 Typumwandlungsoperator

Erinnern Sie sich an die `if`- und die `while`-Anweisung, in deren Bedingung nicht nur ein Ausdruck mit einem logischen oder arithmetischen Ergebnis, sondern auch ein Zeiger stehen darf? Ein Zeiger muss dann in einen äquivalenten Wert (0 oder 1, entsprechend `false` oder `true`) umgewandelt werden. Sie wissen, dass der Compiler Typumwandlungen automatisch vornimmt, wenn zum Beispiel einer `float`-Zahl eine `int`-Zahl zugewiesen wird. Bedingung für die Umwandlung von Klassenobjekten ist ein Typumwandlungskonstruktor (siehe Abschnitt 4.3.4) oder ein Typumwandlungsoperator, der implizit durch den Compiler oder explizit durch Angabe des Datentyps angewendet werden kann:

```
float f;
int i = 1234;
f = i; // implizit
f = static_cast<float>(i); // explizit
```

Die Typumwandlung mit einem Konstruktor haben Sie in Abschnitt 4.3.4 kennen und in Abschnitt 4.4 anwenden gelernt. C++ erlaubt darüber hinaus die Definition von Typumwandlungsoperatoren für selbst geschriebene Klassen, um ein Objekt der Klasse in einen anderen Datentyp abzubilden. *Typumwandlungsoperatoren sind Elementfunktionen und haben weder eine Parameterliste noch einen Rückgabetyt* (siehe Syntax in Abbildung 9.5).

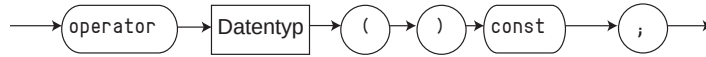


Abbildung 9.5: Syntax eines Typumwandlungsoperators

Hier wird als Beispiel die Umwandlung eines Datum-Objekts in einen String gezeigt, der dann weiterverarbeitet oder ausgegeben werden kann. Die Deklaration in der Klasse Datum sowie die Implementierung lauten:

```
// datum.h
#include<string>

class Datum {
    // ... wie vorher
    operator std::string() const;
};

// in datum.cpp:
Datum::operator std::string() const {
    std::string temp("tt.mm.jjjj");
    temp[0] = tag_/10 + '0';    // implizite Umwandlungen
    temp[1] = tag_%10 + '0';
    temp[3] = monat_/10 + '0';
    temp[4] = monat_%10 + '0';

    int pos = 9;                // letzte Jahresziffer
    int j = jahr_;
    while(j > 0) {
        temp[pos] = j % 10 + '0'; // letzte Ziffer
        j /= 10;                // letzte Ziffer abtrennen
        --pos;
    }
    return temp;
}
```

Es ist sowohl die implizite als auch die explizite Umwandlung möglich, wie die folgenden Zeilen zeigen.

```
using namespace std;
Datum d;
string s1 = d;                // implizit oder
string s2 = (string)d;        // explizit
string s3 = static_cast<string>(d); // explizit
cout << s1 << endl;          // Ausgabe
```

**Empfehlung**

Um die Typprüfung des Compilers nicht unnötig zu umgehen, sollte man Typumwandlungsoperatoren nur in begründeten Fällen einsetzen. Alternativ bieten sich Methoden an. In diesem Beispiel wäre eine Methode namens `toString()` anstelle des Typumwandlungsoperators die bessere Lösung, weil dann eine implizite Typumwandlung von vornherein unmöglich ist.

**Übung**

9.15 Ersetzen Sie den Typumwandlungsoperator durch eine Methode `toString()`.

9.5 Smart Pointer: Operatoren -> und *

Ein Schlüsselkonzept von C++ ist der Zugriff auf Objekte und deren Elementfunktionen über Zeiger, *Indirektion* genannt. Wer kennt die Fallstricke nicht, die mit der Verwendung von Zeigern einhergehen! Typische Fehler sind die Dereferenzierung von nicht initialisierten Zeigern oder versehentliche Versuche, `delete` mehr als einmal auf einen Zeiger anzuwenden. Andere Fehler sind hängende Zeiger und verwitwete Objekte (siehe Seite 203) – alles Fehler, die manchmal schwer zu finden sind. Gegen diese typische C-Krankheit bietet C++ eine Medizin: »intelligente« Zeiger (englisch *smart pointers*).

Die Vielseitigkeit von C++ erlaubt es, auch die Operatoren `->` und `*` zu überladen, sodass damit eine Klasse von »smart pointers« geschaffen werden kann, mit der Indirektion auf direkte, sichere und effiziente Art darstellbar ist. Weil Zeiger typischerweise für dynamische Objekte verwendet werden, werden im Folgenden nur smarte Pointer für Heap-Objekte betrachtet. Das Problem, versehentlich `delete` zu vergessen, existiert nicht für Objekte, die auf dem Stack abgelegt werden.

Was erwarten wir von einem intelligenten Zeiger?

1. Die Syntax der Benutzung soll möglichst der bekannten Schreibweise von »normalen« Zeigern gleichen.
2. Ein intelligenter Zeiger sollte ohne viel Aufwand für verschiedene Klassen möglich sein, ohne an Typsicherheit einzubüßen.
3. Gegenüber einem normalen Zeiger sollte kein Laufzeitmehrbedarf entstehen.
4. Ein smarter Pointer hat niemals einen undefinierten Wert. Er verweist entweder auf ein Objekt oder definitiv auf nichts, um hängende Zeiger zu vermeiden.
5. Die Dereferenzierung von nicht existenten Objekten soll nicht zu einem Programmabsturz, sondern zu einer Exception führen.
6. Das Objekt, auf das der Zeiger verweist, soll gelöscht werden, sobald der Zeiger nicht mehr gültig ist, um verwitwete Objekte zu vermeiden.

Des Weiteren sind noch viele Dinge denkbar, die ein intelligenter Zeiger bei der Dereferenzierung erledigen könnte. Beispiele: ein Objekt erst bei Bedarf vom Massenspeicher holen, in Abhängigkeit vom Zugriff auf konstante oder nichtkonstante Objekte unterschiedlich reagieren und anderes mehr. Um das folgende Beispiel nicht zu überfrachten, bleibt es auf die oben angeführten Punkte beschränkt. Bei der Verwendung ist zu beachten, dass eine gemischte Verwendung von »intelligenten« mit normalen C-Zeigern nicht sinnvoll ist, weil die Sicherheitsmaßnahmen der Klasse unterlaufen werden können.

Der bisher bekannte Operator `->` ist für jede Adresse verwendbar, die vom Typ »Zeiger auf `struct` oder `class`« ist. Der überladene Operator verhält sich genauso: Seine Benutzung ist nichts anderes als der Aufruf der Funktion `operator->()` (siehe Tabelle 9.1, Seite 318), die eben so einen Zeiger zurückgibt – nur dass sie vorher noch etliche andere Dinge erledigen kann! Für Grunddatentypen ist ein Überladen allerdings nicht möglich.

Die Forderung nach Typsicherheit und gleichzeitiger Anwendbarkeit für verschiedene Klassen wird durch ein Klassen-Template erfüllt. Durch `inline`-Definitionen entsteht, verglichen mit einem normalen Zeiger, kein zusätzlicher Laufzeitaufwand für die Funktionalität. Nur darüber hinausgehende Extras wie die Überprüfung auf Gültigkeit kosten Laufzeit – nicht alles ist umsonst. Falls auf einen nicht-gültigen Zeiger zugegriffen wird, soll eine `NullPointerException` geworfen werden:

```
// cppbuch/k9/smartptr/NullPointerException.h
#ifndef NULLPOINTEREXCEPTION_H
#define NULLPOINTEREXCEPTION_H
#include<stdexcept>

class NullPointerException : public std::runtime_error {
public:
    NullPointerException()
        : std::runtime_error("NullPointerException!") {
    }
};
#endif
```

Im folgenden Beispiel wird ein Klassen-Template `SmartPointer` für »intelligente« Zeiger auf *Heap-Objekte* vorgestellt. Die Klasse soll nicht für Zeiger gelten, die auf ein Array von Objekten verweisen.

Listing 9.5: Template für `SmartPointer`

```
// cppbuch/k9/smartptr/smartptr.t
#ifndef SMARTPTR_T
#define SMARTPTR_T
#include "NullPointerException.h"

template<typename T>
class SmartPointer {
public:
    SmartPointer(T *p = 0);
    ~SmartPointer(); // nicht virtual: Vererbung ist nicht geplant
    T* operator->() const;
    T& operator*() const;
    SmartPointer& operator=(T *p);
```

```

    operator bool() const;
private:
    T* zeigerAufObjekt;           // Ergänzungen siehe unten
    void check() const;          // Prüfung auf nicht-Null
};
// ... hier die unten beschriebenen inline-Implementierungen einfügen
#endif // SMARTPTR_T

```

Die Variable `zeigerAufObjekt` verweist auf ein Objekt eines beliebigen Typs `T`. Alle Zugriffe geschehen über die Operatoren `->` und `*`. Der Konstruktor stellt sicher, dass `zeigerAufObjekt` den Wert 0 bei Initialisierung erhält, wenn keine Objektadresse übergeben wird.

```

template<typename T>
inline SmartPointer<T>::SmartPointer(T *p)
: zeigerAufObjekt(p) {
}

template<typename T>
inline SmartPointer<T>::~~SmartPointer() {
    delete zeigerAufObjekt;
}

template<typename T>
inline void SmartPointer<T>::check() const {
    if(!zeigerAufObjekt) {
        throw NullPointerException();
    }
}

```

Der Operator `->` gibt einen Zeiger auf das Objekt zurück, wobei im Unterschied zum bisher bekannten Pfeiloperator geprüft wird, ob er auf ein Objekt verweist. Falls nicht, gibt es eine Exception.

```

template<typename T>
inline T* SmartPointer<T>::operator->() const {
    check();
    return zeigerAufObjekt;
}

```

Der Operator `*` gibt eine Referenz auf das Objekt zurück, wobei im Unterschied zum bisher bekannten Dereferenzierungsoperator geprüft wird, ob `zeigerAufObjekt` tatsächlich auf ein Objekt verweist. Falls nicht, gibt es eine Exception.

```

template<typename T>
inline T& SmartPointer<T>::operator*() const {
    check();
    return *zeigerAufObjekt;
}

```

Schließlich soll die Abfrage möglich sein, ob ein smarter Pointer auf ein Objekt zeigt. Dazu wird ein Typumwandlungsoperator benutzt, sodass Abfragen in `if-` oder `while-` Bedingungen mit der vertrauten Syntax `if(smartPointerObjekt) ...` möglich sind.

```
template<typename T>
inline SmartPointer<T>::operator bool() const {
    return bool(zeigerAufObjekt);
}
```

Die Aufgabe, ein Objekt zu löschen, sobald der zugehörige Zeiger nicht mehr gültig ist, übernimmt der Destruktor des smarten Pointers. Es muss in diesem Beispiel vermieden werden, dass mehrere SmartPointer-Objekt auf dasselbe Objekt verweisen. Bei dynamischen Objekten gäbe es andernfalls das Problem, dass das erste ungültig werdende SmartPointer-Objekt das referenzierte Objekt löscht mit der Folge, dass die anderen nicht mehr darauf zugreifen könnten, obwohl die internen Zeiger (`zeigerAufObjekt`-Variablen) ungleich 0 sind.

Es gibt Möglichkeiten, beliebig viele smarte Pointer auf ein Objekt verweisen zu lassen, ohne dass diese Schwierigkeiten auftreten. Sie setzen eine Buchführung voraus, die mitzählt, wie viele Zeiger auf ein Objekt verweisen. Die Benutzungszählung (englisch *reference counting*) wird von der Klasse `shared_ptr` (siehe Abschnitt 9.5.1) der C++-Standardbibliothek geleistet, übersteigt aber den Rahmen dieses einfachen Beispiels, weswegen hier schlicht verboten wird, dass mehr als ein smarter Pointer auf ein Objekt verweist. Das geschieht am einfachsten dadurch, dass sowohl der Zuweisungsoperator als auch der Initialisierungs- oder Kopierkonstruktor durch eine `private`-Deklaration unzugänglich gemacht werden:

```
private: // Ergänzung der Klassendefinition:
void operator=(const SmartPointer& ); // Zuweisung p1 = p2; verbieten
SmartPointer(const SmartPointer&); // Initialisierung mit SmartPointer verbieten
```

Damit ist zwar nicht jede, aber wenigstens eine Fehlerquelle gestopft. Eine Parameterübergabe per Wert ist die Erzeugung und Initialisierung eines Objekts, bei der dem Kopierkonstruktor eine Referenz auf das Original mitgegeben wird. Ein privater Kopierkonstruktor sorgt dafür, dass eine Übergabe per Wert nicht möglich ist und ein SmartPointer nur noch per Referenz übergeben werden kann. Obwohl sich da die Benutzung von SmartPointer-Objekten von der normaler Zeiger unterscheidet, ist dies ein erwünschter Effekt, denn: Eine Übergabe per Wert erzeugt eine temporäre Kopie des Parameters, deren Destruktor am Ende der Funktion aufgerufen wird. Der Destruktor würde das zugehörige Objekt löschen, obwohl es im aufrufenden Programm noch gebraucht wird!

Das folgende Programm zeigt einige Anwendungen von smarten Pointern. Es werden Objekte der Klassen A und B erzeugt, auf die mit smarten Pointern zugegriffen wird. Dabei erbt B von A, damit polymorphes Verhalten gezeigt werden kann. Die Destruktoren dokumentieren das Vergehen der Objekte, wenn die Zeiger ungültig werden.

Listing 9.6: Beispiel mit SmartPointer-Objekten

```
// cppbuch/k9/smartptr/main.cpp
#include "smartptr.h"
#include <iostream>
using namespace std;

class A {
public:
    virtual void hi() { cout << "hier ist A::hi()" << endl; }
```

```

    virtual ~A()    { cout << "A::~Destruktor" << endl; }
};

class B : public A {
public:
    virtual void hi() { cout << "hier ist B::hi()" << endl; }
    virtual ~B()    { cout << "B::~Destruktor" <<endl; }
};

// Parameterübergabe per Wert ist hier nicht möglich, wohl aber per Referenz:
template<typename T>
void perReferenz(const SmartPointer<T>& p) {
    cout << "Aufruf: perReferenz(const SmartPointer<T>&):";
    p->hi();    // d.h. (p.operator->())->hi();
}

int main() {
    cout << "Zeiger auf dynamische Objekte:" << endl;
    cout << "Konstruktoraufruf" << endl;
    SmartPointer<A> spA(new A);

    cout << "Operator ->" << endl;
    spA->hi();

    cout << "Operator *" << endl;
    (*spA).hi();

    cout << "Polymorphismus:" << endl;
    SmartPointer<A> spAB(new B); // zeigt auf B-Objekt
    spAB->hi();                  // B::hi()

    // Parameterübergabe eines SmartPointer
    perReferenz(spAB);

    // Die Wirkung der Sicherungsmaßnahmen im Vergleich zu einfachen
    // C++-Zeigern zeigen die folgenden Zeilen:
    SmartPointer<B> spUndef;
    try {
        if (!spUndef)          // = if(!(spUndef.operator bool()))
            cout << "undefinierter Zeiger:" << endl;
        // Zugriff auf nichtinitialisierten Zeiger bewirkt Laufzeitfehler:
        spUndef->hi();          // Laufzeitfehler!
        (*spUndef).hi();        // Laufzeitfehler!
    } catch(const NullPointerException& ex) {
        cout << "Laufzeitfehler: " << ex.what() << endl;
    }

    // alle folgenden Anweisungen bewirken Fehlermeldungen des Compilers!
    // Typkontrolle: ein A ist kein B!
    SmartPointer<B> spTyp(new A);    // Fehler!
    // keine Initialisierung mit Kopierkonstruktor
    SmartPointer<A> spY = spA;        // Fehler!
    // Zuweisung ist nicht möglich (privater -=Operator):

```

```
SmartPointer<A> spA1;
spA1 = spA;                // Fehler!
}
```

9.5.1 Smart Pointer und die C++-Standardbibliothek

Ein `SmartPointer`, wie oben beschrieben, behält ein Objekt solange, bis der Destruktor beim Verlassen des Gültigkeitsbereichs wirksam wird. Abschnitt 33.1 beschreibt vordefinierte Klassen der C++-Standardbibliothek für smarte Zeiger. Eine davon ist die schon erwähnte Klasse `shared_ptr`. Es können mehrere Objekte dieser Klasse auf ein Objekt verweisen, weil sie eine Benutzungszählung implementiert. Im Gegensatz zur Klasse `SmartPointer` ist eine Parameterübergabe per Wert nicht schädlich, weil die Kopie nur den Benutzungszähler erhöht. Der Destruktor eines `shared_ptr`-Objekts wendet nur dann `delete` auf das referenzierte Objekt an, wenn der Benutzungszähler 1 ist, also kein anderes `shared_ptr`-Objekt (mehr) darauf verweist.



Mehr zu `shared_ptr` und Beispiele lesen Sie in Abschnitt 33.1.

9.6 Objekt als Funktion

In einem Ausdruck wird der Aufruf einer Funktion durch das von der Funktion zurückgegebene Ergebnis ersetzt. Die Aufgabe der Funktion kann von einem Objekt übernommen werden, eine Technik, die in den Algorithmen und Klassen der C++-Standardbibliothek häufig eingesetzt wird. Dazu wird der Funktionsoperator `()` mit der Operatorfunktion `operator()()` überladen. Ein Objekt kann dann wie eine Funktion aufgerufen werden. Ein algorithmisches Objekt dieser Art wird *Funktionsobjekt* oder *Funktor* genannt.

Funktoren sind Objekte, die sich wie Funktionen verhalten, aber alle Eigenschaften von Objekten haben. Sie können erzeugt, als Parameter übergeben oder in ihrem Zustand verändert werden. Die Zustandsänderung erlaubt einen flexiblen Einsatz, der mit Funktionen nur über zusätzliche Parameter möglich wäre. Wie funktioniert ein Funktor? Betrachten wir dazu eine Klasse, deren Objekte den Sinus eines Winkels berechnen. Den Objekten wird bei der Erzeugung bereits mitgeteilt, in welchen Einheiten der Winkel angegeben wird.

Das syntaktisch entscheidende Element ist der überladene Klammeroperator `operator()()`. Durch ihn ist es möglich, dass ein Objekt wie eine Funktion aufgerufen werden kann.

Listing 9.7: Beispiel für ein Funktionsobjekt

```
// cppbuch/k9/funktor/sinus.h
#ifndef SINUS_H
#define SINUS_H
#include <cmath> // sin(), Konstante M_PI für  $\pi$ 
// manche Compiler stellen Konstanten wie M_PI nicht zur Verfügung
#ifndef M_PI
```

```

#define M_PI 3.14159265358979323846
#endif

class Sinus {
public:
    enum Modus { bogenmass, grad, neugrad};
    Sinus(Modus m = bogenmass)
    : berechnungsart(m) {
    }

    double operator()(double arg) const {
        double erg;
        switch(berechnungsart) {
            case bogenmass : erg = std::sin(arg);
                            break;
            case grad      : erg = std::sin(arg/180.0*M_PI);
                            break;
            case neugrad   : erg = std::sin(arg/200.0*M_PI);
                            break;
            default       : ;      // kann hier nicht vorkommen
        }
        return erg;
    }
private:
    Modus berechnungsart;
}
#endif // SINUS_H

```

Die Algorithmen der C++-Standardbibliothek benutzen häufig Funktionsobjekte. Ein Beispiel dafür ist `setprecision` zum Festlegen der Anzahl ausgegebener Stellen bei der Ausgabe von Zahlen (siehe Abschnitt 10.3). Im Beispielprogramm werden Objekte für drei verschiedene Winkleinheiten angelegt und als Funktionsobjekt benutzt. Ferner wird eins von den drei Objekten als Parameter übergeben. Was unterscheidet Funktionsobjekte von Funktionen?

- Sie erfüllen die Aufgabe einer Funktion, haben aber alle Eigenschaften von Objekten.
- Funktionsobjekte sind flexibler als Funktionen, weil virtuelle Funktionen und Vererbung möglich sind.

Listing 9.8: Funktor

```

// /cppbuch/k9/funktor/main.cpp
#include<iostream>
#include"sinus.h"
using namespace std;

void sinusAnzeigen(double arg, const Sinus& funktor) {
    cout << funktor(arg) << endl;
}

int main() {
    Sinus sinrad;           // Funktoren anlegen
    Sinus sinGrad(Sinus::grad);
    Sinus sinNeuGrad(Sinus::neugrad);
}

```

```
// Aufruf der Objekte wie eine Funktion
cout << "sin(" << M_PI/4.0 << " rad) = " //  $\pi/4 = 0.785398...$ 
    << sinrad(M_PI/4.0) << endl; // = sinrad.operator() (...);
cout << "sin(0.7854 rad) = " << sinrad(0.7854) << endl;
cout << "sin(45 Grad) = " << sinGrad(45.0) << endl;
cout << "sin(50 Neugrad) = " << sinNeuGrad(50.0) << endl;

// Übergabe eines Funktors an eine Funktion
sinusAnzeigen(50.0, sinNeuGrad);
// ...
}
```

- Sie ermöglichen eine einfachere Schnittstelle: Wenn ein Funktionsobjekt per Wert oder per Referenz an eine Funktion übergeben wird, kann es die verschiedensten Daten intern mit sich tragen. Im Beispiel oben ist es die Berechnungsart. Die vergleichbare Übergabe eines Funktionszeigers mit denselben Daten erfordert eine viel breitere Schnittstelle.

Funktionsobjekte sind durchaus nicht auf arithmetische Operationen beschränkt. Ihr Einsatz sollte immer dann überlegt werden, wenn Zeiger auf Funktionen verwendet werden.

9.6.1 Lambda-Funktionen

Mit Lambda-Ausdrücken können einfach namenlose Funktionsobjekte realisiert werden. Lambda-Funktionen oder -Ausdrücke sind anonyme (unbenannte) Funktionen, die direkt an der Stelle ihrer Verwendung definiert werden. Sie sind daher geeignet, wenn die Funktion nur einmal benötigt wird. Die Funktionen haben ihren Namen vom sogenannten Lambda-Kalkül, einem formalen System zur Erforschung einiger theoretischer Grundlagen der Mathematik und Informatik. Lambda-Funktionen werden wie ein Funktionsobjekt benutzt, nur dass man sich das Schreiben einer Klasse und der zugehörigen Funktion `operator()` sparen kann. Der Compiler wandelt einen Lambda-Ausdruck in ein anonymes Funktionsobjekt um. Im folgenden Beispiel werden Vektorelemente nach dem Absolutbetrag sortiert. Gezeigt wird die Schreibweise ohne und mit Lambda-Funktion.

Listing 9.9: Nach Absolutbetrag sortieren, ohne Lambda-Funktion

```
struct Absolutbetrag {
    bool operator()(int x, int y) {
        return abs(x) < abs(y);
    }
};

int main() {
    vector<int> v = {-11, 3, 4, -7, 8, 1, 2, -4};
    sort(v.begin(), v.end(), Absolutbetrag());
}
```

Listing 9.10: Nach Absolutbetrag sortieren, mit Lambda-Funktion

```
// Auszug aus cppbuch/k9/funktor/lambda/main.cpp
vector<int> v = {-11, 3, 4, -7, 8, 1, 2, -4};
sort(v.begin(), v.end(), [](int x, int y) { return abs(x) < abs(y); });
```

Eine Lambda-Funktion beginnt mit dem Symbol `[]` am Anfang, dann folgen die Parameterliste in runden Klammern und abschließend eine durch geschweifte Klammern begrenzte Verbundanweisung. Das vom Compiler erzeugte Funktionsobjekt hat Zugriff nicht nur auf interne Variablen, sondern auch auf Variablen, die im selben Sichtbarkeitsbereich (*scope*) liegen. Im folgenden Beispiel wird die Summe aller Elemente eines Vektors berechnet:

```
// Auszug aus cppbuch/k9/funktor/lambda/main.cpp
double summe = 0.0;
for_each(v.begin(), v.end(), [&summe](double el) { summe += el; });
```

Wenn die Lambda-Funktion mit dem Symbol `[&]` beginnt, kann innerhalb der Verbundanweisung auf alle sichtbaren Variablen zugegriffen werden. Hier wird der Zugriff auf `summe` beschränkt. Die Menge der zugreifbaren Daten heißt in diesem Zusammenhang auch *Closure*. Die in den eckigen Klammern angegebenen externen Referenzen werden auf entsprechende Attribute des generierten temporären Funktionsobjekts abgebildet, so dass *Closure* und Objekt sich entsprechen. Wenn die Lambda-Funktion mit dem Symbol `[=]` beginnt, werden im Funktionsobjekt Referenzen angelegt; wenn stattdessen `[=]` genommen wird, werden alle Variablen kopiert.

9.7 new und delete überladen¹

Wenn Sie für eine Klasse ein eigenes Memory-Management benötigen, können Sie die Operatoren `new` und `delete` (und natürlich `new[]` und `delete[]`) überladen. Um die Klasse der Anwendung nicht damit zu überfrachten, wird das Memory-Management in eine eigene Klasse ausgelagert, von der die Anwendungsklasse erbt. Die Anwendungsklasse sei hier eine einfache Klasse `Person`. Das `main`-Programm erzeugt zum Vergleich sowohl Objekte, die vom Laufzeitsystem auf dem Laufzeit-Stack abgelegt werden, als auch mit `new` erzeugte Objekte, die auf dem Heap landen.

Listing 9.11: `main`-Programm

```
// cppbuch/k9/new/new/main.cpp
#include "Person.h"
using namespace std;

int main() {
    Person person("Lena");           // Stack-Objekt
    cout << "Name : " << person.getName() << endl;
    Person* ptr1 = new Person("Jens"); // Heap-Objekt
    cout << "Name : " << ptr1->getName() << endl;
    delete ptr1;                     // Löschen des Heap-Objekts

    size_t anz = 2;
    Person* arr = new Person[anz];   // dynamisches Array anlegen
```

¹ Dieser Abschnitt kann beim ersten Lesen übersprungen werden.


```

    for(size_t i = 0; i < anz; ++i) {
        cout << i << ": " << arr[i].getName() << endl;
    }
    delete[] arr;                                // dynamisches Array löschen
}                                                  // Destruktor-Aufruf des Stack-Objekts

```

Um das Überladen kennenzulernen, wird ein Ansatz für ein eigenes Memory-Management zurückgestellt und zunächst nur dokumentiert, was beim Überladen von `new` und `delete` geschieht. Aus diesem Grund gibt es eine Meldung des Destruktors der Klasse `Person`:

Listing 9.12: Klasse `Person`

```

// cppbuch/k9/new/new/Person.h
#ifndef PERSON_H_
#define PERSON_H_
#include<iostream>
#include<string>
#include"Objekt.h"

class Person : public Objekt {
public:
    Person(const std::string& n = "N.N.")
        : name(n) {
    }

    ~Person() {
        std::cout << "Person-Destruktor aufgerufen ("
                    << name << ")" << std::endl;
    }

    const std::string& getName() const {
        return name;
    }
private:
    std::string name;
};
#endif

```

Wenn `Person` nicht von `Objekt` erben würde, kämen die Standardoperatoren `new` und `delete` zum Einsatz. So aber wählt der Compiler die passenden Operatoren der Klasse `Objekt`:

Listing 9.13: Klasse `Objekt`

```

// cppbuch/k9/new/new/Objekt.h
#ifndef OBJEKT_H_
#define OBJEKT_H_
#include<iostream>

class Objekt {
public:
    virtual ~Objekt() {
        std::cout << "Objekt-Destruktor aufgerufen ("
                    << this << ")" << std::endl;
    }
}

```

```

static void *operator new(size_t size) {
    std::cout << "new aufgerufen. size=" << size << std::endl;
    return ::operator new(size);
}

static void operator delete(void* ptr, size_t size) {
    std::cout << "delete aufgerufen. size=" << size << std::endl;
    ::operator delete(ptr);
}

static void *operator new[](size_t size) {
    std::cout << "new[] aufgerufen. size=" << size << std::endl;
    return ::operator new[](size);
}

static void operator delete[](void* ptr, size_t size) {
    std::cout << "delete[] aufgerufen. size=" << size << std::endl;
    ::operator delete[](ptr);
}
};
#endif

```

Die Funktionen der Klasse dokumentieren die Aufrufe und greifen dann auf die Standardoperatoren zurück. Dass andere Speicherbeschaffungsfunktionen möglich sind, wird unten gezeigt. Die Klasse `Objekt` hat einige Besonderheiten:

- Der Destruktor ist `virtual`. Dies ist zwingend erforderlich, soll es keine Speicherlecks geben, wie das Beispiel in Abschnitt 7.6.3 beweist.
- Alle Funktionen sind `static`. Das Schlüsselwort steht hier zur Dokumentation; tatsächlich kann es weggelassen werden. Aber ob es da steht oder nicht – `new` und `delete` sind grundsätzlich `static`. Der Grund liegt auf der Hand: Es wird kein Objekt modifiziert oder abgefragt, sondern es wird erzeugt bzw. gelöscht. Insofern kann auch ein Konstruktor als statische Funktion aufgefasst werden.
- Bei den `delete`-Funktionen kann der zweite Parameter `size` weggelassen werden. Ich habe ihn belassen, um die Größe des zu löschenden Bereichs zu dokumentieren.
- Der Rückgriff auf die Standardoperatoren wird durch den Scope-Operator `::` bewerkstelligt. Fehlt er, handelt man sich eine Kette rekursiver Aufrufe ein.
- Der Compiler wandelt zur Speicherbeschaffung die Anweisung `new Person("Jens")`; im `main`-Programm in die Form `Objekt::operator new(sizeof(Person))`; um. Das Objekt `Person("Jens")` wird an dieser Stelle konstruiert.
- Die Anweisung `new Person[anz]`; im `main`-Programm wird vom Compiler in die Form `Objekt::operator new[](anz*sizeof(Person) + x)`; umgewandelt. Das Array wird an dieser Stelle konstruiert. `x` ist eine implementationsabhängige Größe, um Verwaltungsinformationen abzulegen. Unten sehen Sie, wie `x` bestimmt wird.

Der Aufruf des `main`-Programms ergibt die Ausgabe (ohne die Zeilennummern), die auf Ihrem System nicht identisch sein muss:

```

1 Name : Lena
2 new aufgerufen. size=8
3 Name : Jens
4 Person-Destruktor aufgerufen (Jens)

```

```

5 Objekt-Destruktor aufgerufen (0x804c038)
6 delete aufgerufen. size=8
7 new[] aufgerufen. size=20
8 0: N.N.
9 1: N.N.
10 Person-Destruktor aufgerufen (N.N.)
11 Objekt-Destruktor aufgerufen (0x804c02c)
12 Person-Destruktor aufgerufen (N.N.)
13 Objekt-Destruktor aufgerufen (0x804c024)
14 delete[] aufgerufen. size=20
15 Person-Destruktor aufgerufen (Lena)
16 Objekt-Destruktor aufgerufen (0xbfd7e0c)

```

Die Zeile 2 zeigt, dass ein Person-Objekt 8 Bytes benötigt. Davon sind 4 Bytes für den Zeiger auf die Tabelle virtueller Funktionen. Dies zeigt der Vergleich, wenn der Destruktor von Objekt fälschlicherweise *nicht* virtual sein sollte.

Die `delete ptr1`-Anweisung bewirkt die Destruktor-Aufrufe von Unter- und Oberklasse (Zeilen 4 und 5). Hier ist zu sehen, dass *zuerst* die Destruktoren aufgerufen werden und *danach* die dokumentierende Anweisung zum Zug kommt (Zeile 6)! Der Grund: Bei der Konstruktion wird zuerst der Speicher beschafft und dann das Objekt konstruiert. Bei der Destruktion muss der Ablauf umgekehrt sein: Erst der Destruktor-Aufruf und dann die Speicherfreigabe. Andernfalls würde der Destruktor in freigegebenem Speicher ablaufen – das darf nicht sein.

Obwohl das Array nur aus zwei Elementen besteht, werden 20 Bytes statt $2 * 8 = 16$ angefordert (Zeile 7). Genau diese Menge wird wieder freigegeben (Zeile 14). Daraus ergibt sich der Wert von 4 Bytes für die oben genannte Hilfsgröße `x`.

Die Zeilen 10 bis 13 dokumentieren die Destruktion der beiden Array-Elemente. Auch hier sieht man (Zeile 14), dass die dokumentierende Anweisung *nach* den Destruktoren aufgerufen wird. Die Zeilen 15 und 16 zeigen die Destruktion des anfangs in `main()` angelegten Stack-Objekts.

9.7.1 Speichermanagement mit `malloc` und `free`

Oben wird bei den überladenen `new`- und `delete`-Operatoren auf die globalen Standardoperatoren zurückgegriffen. Eine andere Art ist die Speicherbeschaffung mit der C-Funktion `malloc(size_t size)` und die Freigabe mit `free(void* ptr)`. `malloc()` gibt einen Zeiger auf den Beginn des Speicherbereichs der Größe `size` Bytes zurück bzw. `NULL`, wenn kein Speicher zugewiesen werden kann. `free(ptr)` gibt den Speicher, auf den `ptr` verweist, wieder frei. `ptr` muss auf einen Speicherbereich zeigen, der vorher mit `malloc()`, `calloc()` oder `realloc()` (siehe unten) beschafft worden und noch nicht wieder freigegeben ist. Falls `ptr` jedoch gleich `NULL` ist, geschieht nichts. Um `malloc()` und `free()` in die Klasse Objekt einzubauen, müssen nur kleine Änderungen vorgenommen werden:

Listing 9.14: Klasse Objekt mit `malloc/free`

```

// cppbuch/k9/new/malloc/Objekt.h
#ifndef OBJEKT_H
#define OBJEKT_H

```

```

#include<iostream>
#include<cstdlib>           // malloc(), free()
#include<new>              // bad_alloc

class Objekt {
public:
    virtual ~Objekt() {
        std::cout << "Objekt-Destruktor aufgerufen ("
                    << this << ")" << std::endl;
    }

    static void *operator new(size_t size) {
        std::cout << "new aufgerufen. size=" << size << std::endl;
        void* ptr = malloc(size);
        if(!ptr) {
            throw std::bad_alloc();
        }
        return ptr;
    }

    static void operator delete(void *ptr, size_t size) {
        std::cout << "delete aufgerufen. size=" << size << std::endl;
        free(ptr);
    }

    static void *operator new[](size_t size) {
        std::cout << "new[] aufgerufen. size=" << size << std::endl;
        void* ptr = malloc(size);
        if(!ptr) {
            throw std::bad_alloc();
        }
        return ptr;
    }

    static void operator delete[](void *ptr, size_t size) {
        std::cout << "delete[] aufgerufen. size=" << size << std::endl;
        free(ptr);
    }
};
#endif

```

Alles andere kann bleiben. Falls die Speicherbeschaffung schief geht, wirft `malloc()` im Gegensatz zu `::new` keine Exception, sondern gibt einen Null-Zeiger zurück. Um dasselbe Verhalten wie mit `::new` zu erreichen, wird der zurückgegebene Wert überprüft und gegebenenfalls `bad_alloc` geworfen. Außer `malloc()` und `free()` gibt es weitere C-Funktionen zur Speicherverwaltung:

- `void* calloc(size_t n, size_t size)` beschafft Speicher für ein Array mit `n` Elementen jeweils der Größe `size`. Es wird ein Zeiger auf den Beginn des Arrays zurückgegeben bzw. `NULL`, wenn kein Speicher zugewiesen werden kann.
- `void* realloc(void* ptr, size_t size)` beschafft neuen Speicher der Größe `size`, kopiert maximal `size` Bytes des Objekts, das sich an der Adresse `ptr` befindet, in den

neuen Speicherbereich, gibt den Speicher bei `ptr` frei und gibt einen Zeiger auf den neu angelegten Speicher zurück. Anders gesagt, für den typischen Fall, dass `size` der Größe des Objekts entspricht: Das Objekt wird im Speicher verschoben, und der vorher belegte Platz wird freigegeben.

9.7.2 Unterscheidung zwischen Heap- und Stack-Objekten

Kann ein Objekt wissen, ob es sich auf dem Heap oder auf dem Stack befindet? Normalerweise nicht, aber wenn `new` und `delete` überladen werden, ergibt sich eine Möglichkeit. Im folgenden Beispiel werden die Objekte gezählt, wobei Heap- und Stack-Objekte unterschieden werden. Der »Trick« besteht darin, dass der `new`-Operator den erhaltenen Zeiger in eine Liste ein- und `delete` ihn wieder austrägt.



Wozu eine eigene Buchführung über Heap-Objekte?

Die werden Sie im Allgemeinen nicht benötigen. Die hier vorgestellte Technik ist jedoch interessant, weil sie in ähnlicher Form von manchen Bibliotheken eingesetzt wird. Die Qt-Bibliothek für grafische Benutzungsoberflächen (Abschnitt 14.2) verwaltet Heap-Objekte in einer Baumstruktur (nicht über den `new`-Operator, sondern über Konstruktoren) und sorgt dafür, dass bei Löschen eines Knotens mit `delete` alle davon abhängigen Knoten (Kind-Knoten) automatisch mit gelöscht werden. Ein Aufruf von `delete` für Kind-Knoten ist damit nicht mehr notwendig.

Im folgenden Beispiel wird statt einer Liste die Klasse `set` von Seite 787 verwendet. Die Feststellung, ob sich ein Element in einem Container befindet, wird von einem `set`-Container schneller beantwortet. Die Methode `count(x)` gibt die Anzahl der enthaltenen Elemente `x` zurück; bei einem Set kann das Ergebnis nur 0 oder 1 sein. Das Attribut `istHeapObjekt` wird entsprechend gesetzt. Die Klasse `Person` unseres Beispiels erbt nun nicht mehr von der Klasse `Objekt`, sondern von der Klasse `ZaehlendesObjekt`:

Listing 9.15: Klasse `ZaehlendesObjekt`

```
// cppbuch/k9/new/zaehle/ZaehlendesObjekt.h
#ifndef ZAEHLENDESOBJEKT_H
#define ZAEHLENDESOBJEKT_H
#include<iostream>
#include<string>
#include<set>

class ZaehlendesObjekt {
public:
    ZaehlendesObjekt() {
        ++gesamt;
        istHeapObjekt = (1 == objekte.count(this));
        std::cout << (istHeapObjekt ? "Heap" : "Stack")
                  << "-Objekt " << this << " erzeugt." << std::endl;
    }

    virtual ~ZaehlendesObjekt() {
        --gesamt;
        std::cout << (istHeapObjekt ? "Heap" : "Stack")
```

```

        << "-Objekt " << this << " zerstört." << std::endl;
    }

    static void status() {
        std::cout << "Es gibt " << gesamt << " Objekt(e), davon "
            << objekte.size() << " Heap-Objekt(e)" << std::endl;
    }

    static void *operator new(size_t size) {
        void* ptr = ::operator new(size);
        objekte.insert(ptr);
        return ptr;
    }

    static void operator delete(void *ptr) {
        objekte.erase(ptr);
        ::operator delete(ptr);
    }
private:
    bool istHeapObjekt;
    static std::set<void*> objekte;
    static int gesamt;
};
#endif

```

Die Instanziierung der statischen Variablen ist in der Implementierungsdatei:

Listing 9.16: Instanziierung statischer Variablen

```

// cppbuch/k9/new/zaehlend/ZaehlendesObjekt.cpp
#include "ZaehlendesObjekt.h"
std::set<void*> ZaehlendesObjekt::objekte;
int ZaehlendesObjekt::gesamt = 0;

```

Der `new`-Operator trägt den zurückzugebenden Zeiger in den Set `objekte` ein. Der direkt anschließend ausgeführte Konstruktor zählt die Gesamtzahl aller Objekte hoch und stellt fest, ob die eigene Adresse `this` im Set enthalten ist.

Der Destruktor vermindert die Gesamtzahl um eins. Der `delete`-Operator trägt den Zeiger des zu löschenden Heap-Objekts aus dem Set aus. Die Methode `status()` gibt die Gesamtzahl aller Objekte und den Anteil der Heap-Objekte aus. Letzter ergibt sich durch die Abfrage des Sets nach der Anzahl noch enthaltener Elemente mit der Methode `size()`.

In der Klasse `ZaehlendesObjekt` wird auf die Operatoren `new[]` und `delete[]` verzichtet. Man könnte die Anzahl der `new[]`-Aufrufe auf dieselbe Art ermitteln und auch ein fehlendes `delete[]` entdecken (siehe unten). Der Konstruktor könnte aber nicht mehr wie oben herausfinden, ob das Objekt ein Heap-Objekt ist, weil in dem Set nur der Beginn des Arrays abgelegt wäre, nicht jedoch die Adressen der einzelnen Array-Elemente.

9.7.3 Fehlende delete-Anweisung entdecken

Die Zählung der Heap-Objekte entsprechend dem vorhergehenden Abschnitt erlaubt es, fehlende `delete`-Anweisungen und damit verwitwete Objekte zu entdecken. Dazu wird

ein globales Objekt des Typs `Waechter` hinzugefügt (siehe Beispiel im Verzeichnis `cppbuch/k9/new/waechter`):

```
// cppbuch/k9/new/waechter/ZaehlendesObjekt.cpp
#include "ZaehlendesObjekt.h"
#include "Waechter.h"                // neu
std::set<void*> ZaehlendesObjekt::objekte;
int ZaehlendesObjekt::gesamt = 0;
Waechter w;                          // neu
```

Weil dieses Objekt global ist, wird sein Destruktor erst nach dem Ende von `main()` ausgeführt. Der Destruktor überprüft, ob der Set der Zeiger auf Heap-Objekte leer ist:

Listing 9.17: Klasse `Waechter`

```
// cppbuch/k9/new/waechter/Waechter.h
#ifndef WAECHTER_H
#define WAECHTER_H
#include "ZaehlendesObjekt.h"

class Waechter {
public:
    ~Waechter() {
        if (ZaehlendesObjekt::objekte.size() > 0) {
            std::cerr << "Es fehlen " << ZaehlendesObjekt::objekte.size()
                << " delete-Anweisungen!" << std::endl;
        }
    }
};
#endif
```

Falls nicht, gibt es eine Fehlermeldung, und man kann sich auf die Suche begeben, an welcher Stelle die `delete`-Anweisung fehlt. Weil die Klasse `Waechter` auf das private Attribut `objekte` zugreift, wird der Zugriff erlaubt, indem `friend class Waechter;` in die Klassendefinition von `ZaehlendesObjekt` eingefügt wird. Die Wirkung wird sichtbar, wenn die Anweisung `delete ptr1;` in `main()` entfernt oder auskommentiert wird.

Listing 9.18: Fehlendes `delete` entdecken

```
// cppbuch/k9/new/waechter/main.cpp
#include "Person.h"
using namespace std;

int main() {
    Person person("Lena");
    cout << "Name : " << person.getName() << endl;
    ZaehlendesObjekt::status();
    Person* ptr1 = new Person("Jens");
    cout << "Name : " << ptr1->getName() << endl;
    ZaehlendesObjekt::status();
    delete ptr1;                // ggf. kommentieren, siehe Text
    ZaehlendesObjekt::status();
}
```

9.7.4 Eigene Speicherverwaltung

Eine universelle eigene Speicherverwaltung zu schreiben, die effizienter als die vorhandene ist, ist ein schwieriges Unterfangen. Weil die normale C++-Speicherverwaltung sehr gut ist, ist so ein Vorhaben auch *nicht empfehlenswert*. Nur für spezielle Zwecke kann eine eigene Speicherverwaltung sinnvoll sein, um zum Beispiel Fehler zu finden oder unter günstigen Umständen sehr schnell sein zu können. Im Folgenden werden solche günstigen Umstände vereinfachend vorausgesetzt. Die maximal von einem Programm benötigte Anzahl von Objekten (d.h. die Menge an Speicher) sei von vornherein bekannt. In so einem Fall kann der Speicher vorher statisch allokiert werden. Dies ist besonders bei eingebetteten Systemen (englisch *embedded systems*) wichtig, denen nur ein beschränkter Speicher zur Verfügung steht.

Ausrichtung an Speichergrenzen

Oben werden bestimmte Fälle wie das Löschen eines Null-Zeigers oder das Ausrichtung (englisch *alignment*) an Speichergrenzen von den benutzten Systemfunktionen automatisch gehandhabt. Mit der Ausrichtung an Speichergrenzen ist gemeint, dass Zeiger systemabhängig nur auf Speicheradressen mit bestimmten Eigenschaften verweisen dürfen. So kann es sein, dass eine Speicheradresse, an der ein `int`-Wert beginnt, durch 4 teilbar sein muss (falls `4 = sizeof(int)` ist). Ein `double`-Wert kann systemabhängig an einer durch 8 oder durch 4 teilbaren Adresse beginnen. Der Grund liegt in der gewünschten hohen Verarbeitungsgeschwindigkeit. Wenn ein `int`-Wert erst aus einem Bereich »krummer« Byte-Adressen extrahiert werden muss, kostet das viel Laufzeit. Der Compiler sorgt dafür, dass Objekte diesen Anforderungen entsprechend angelegt werden. Sie können das leicht sehen, indem Sie sich `sizeof(char)` und `sizeof(double)` ausgeben lassen (auf meinem System 1 und 8), und dann eine aus beiden zusammengesetzte Struktur bilden:

```
struct SizeofTest {  
    char x;  
    double d;  
};
```

`sizeof(SizeofTest)` ergibt auf meinem System 12 statt 9 – ein Hinweis, dass einige nach `x` liegende Bytes zugunsten eines schnelleren Zugriffs auf `d` verschenkt werden. Bei der eigenen Verwaltung von Speicherplatz müssen diese Anforderungen berücksichtigt werden, sofern der Compiler nicht dafür sorgt. In C++ ist eine Funktion `alignof(Typ)` vorgesehen, die die Stückelung des Speichers zurückgibt. `alignof(SizeofTest)` ergibt auf meinem System 4. Es könnten daher vier statt einer `char`-Variablen in der Struktur ohne weiteren Speicherbedarf untergebracht werden. Erst bei einer fünften `char`-Variablen würde der Speicherbedarf der Struktur um 4 auf 16 Bytes wachsen.

Konsequenzen für die Verwaltung des Speichers

Die oben vorgeschlagene Oberklasse `Objekt` zur Speicherverwaltung weiß nicht, von welchem Typ die Unterklasse ist, die die geerbten Funktionen `new` und `delete` aufruft. Damit gibt es keine Möglichkeit zur Bestimmung der Speicherausrichtung. Die Konsequenz: Die überladenen Operatoren `new` und `delete` müssen in die Anwendungsklasse verlegt werden, weil dort das Alignment bestimmt werden kann. Von dieser Klasse darf aus dem genannten Grund nicht geerbt werden. Dies sind natürlich Einschränkungen. Anderer-

seits sollte die Verwaltung selbst bereitgestellten Speichers ohnehin nur eine Ausnahme bleiben; normalerweise genügen `::new` und `::delete`. Zur Demonstration wird die Klasse `Person` entsprechend angepasst:

- Das Attribut `name` ist jetzt vom Typ `char[15]`. Damit liegt der Speicherbedarf fest, wie für ein eingebettetes System sinnvoll. Der Typ `string` impliziert `new`-Operationen und wird deshalb nicht gewählt. Der beschriebene Speicherverwaltungsmechanismus würde mit `string` auch funktionieren, nur wäre zusätzlicher Heap-Speicher notwendig, dessen Menge wegen der unvorhersehbaren Länge eines Strings nicht bezifferbar ist. Im Beispiel wird jeder String auf 14 Zeichen + Null-Byte gekürzt.
- Der Destruktor ist nicht `virtual`, weil von dieser Klasse nicht geerbt werden soll.
- Die Speicherverwaltung wird in eine eigene Klasse verlegt, die als Template gestaltet und damit auch für andere Typen nutzbar ist. Alle Funktionen sind `static`, weil die Klasse für alle `Person`-Objekte zuständig ist.
- Basis für die Speicherstückelung sind nicht die einzelnen Attribute der Klasse, sondern `sizeof(Person)`. Damit sorgt der Compiler für die richtige Ausrichtung.

Listing 9.19: Klasse `Person` mit eigener Speicherverwaltung

```
// Auszug aus cppbuch/k9/new/eigene/Person.h
#ifndef PERSON_H_
#define PERSON_H_
#include<algorithm>    // std::min()
#include<string>
#include<cstring>
#include"Speicherverwaltung.t"

class Person {
public:
    typedef Speicherverwaltung<Person, 100> Speicher;

    Person(const std::string& n = "N.N.") {
        strncpy(name, n.c_str(), sizeof(name)-1);
        size_t ende = std::min(n.length(), sizeof(name)-1);
        name[ende] = '\0';
    }

    const std::string getName() const {
        return name;
    }

    static void *operator new(size_t size) {
        return Speicher::getMemory(size);
    }

    static void operator delete(void* ptr) {
        Speicher::freeMemory(ptr);
    }

    static int freiePlaetze() { // gibt die Anzahl noch verfügbarer Plätze zurück
        return Speicher::freiePlaetze();
    }
}
```

```
private:
    char name[15];
};
#endif
```

Mit typedef werden Typ und maximale Anzahl der Objekte an einer Stelle festgelegt. Der Alias Speicher kann an jeder Stelle abkürzend anstelle des vollständigen Templatenamens angegeben werden. Der Konstruktor kopiert maximal `sizeof(name)-1` Zeichen und setzt das abschließende Null-Byte.

Die Klasse Speicherverwaltung hat nur statische Variablen. Wie aus Abschnitt 6.2 bekannt, geschieht die Initialisierung und Definition der klassenspezifischen Variablen außerhalb der Klassendeklaration. Bei statischen Template-Attributen braucht es keine Implementierungsdatei; die Definitionen werden nach der Klassendeklaration aufgeführt, wie unten gezeigt.

- `speicher` ist ein fixes char-Array, in dem die Objekte abgelegt werden. Weil `T` den Typ und `N` die maximale Anzahl der Objekte darstellen, beträgt die Anzahl der Arrayelemente `sizeof(T)*N`.
- `belegt` ist das Array zur Speicherverwaltung. Für jedes der `N` möglichen Objekte wird dort seine Adresse eingetragen. Das Array ist durch die Art der Speicherverwaltung stets dicht belegt; es gibt keine Lücken (siehe unten).
- `ersterFreierPlatz` gibt die Stelle im Array `speicher` an, wo das nächste Objekt eingetragen werden kann. `ersterFreierPlatz` verweist stets auf die Position direkt nach dem Ende des belegten Bereichs.

Listing 9.20: Klasse Speicherverwaltung

```
// cppbuch/k9/new/eigene/Speicherverwaltung.h
#ifndef SPEICHERVERWALTUNG_T
#define SPEICHERVERWALTUNG_T
#include<new> // bad_alloc
#include<stdexcept> // invalid_argument

template<typename T, int N>
class Speicherverwaltung {
public:
    static void* getMemory(size_t size) {
        if(ersterFreierPlatz >= N) {
            throw std::bad_alloc();
        }
        if(size != sizeof(T)) {
            throw std::invalid_argument("getMemory(): Aufruf fuer falschen Typ!");
        }
        belegt[ersterFreierPlatz] = &speicher[sizeof(T)*ersterFreierPlatz];
        return belegt[ersterFreierPlatz++];
    }

    static void freeMemory(void* ptr) {
        if(ptr) { // Null-Zeiger ignorieren
            for(int i = ersterFreierPlatz-1; i >=0; --i) { // Stelle suchen
                if(belegt[i] == ptr) {
                    // 'freimachen' durch Kopieren des letzten Eintrags
```

```

        // an diese Stelle und Anpassen von ersterFreierPlatz
        belegt[i] = belegt[--ersterFreierPlatz];
        break;
    }
}
}
}

static int freiePlaetze() {
    return N - ersterFreierPlatz;
}

private:
    static int ersterFreierPlatz;
    static void* belegt[N];
    static char speicher[sizeof(T)*N];
};

// Definitionen
template<typename T, int N>
int Speicherverwaltung<T, N>::ersterFreierPlatz = 0;

template<typename T, int N>
void* Speicherverwaltung<T, N>::belegt[N];

template<typename T, int N>
char Speicherverwaltung<T, N>::speicher[sizeof(T)*N];
#endif

```

Die Funktion `getMemory()` überprüft zunächst, ob überhaupt noch Platz vorhanden ist. Falls nicht, wird `bad_alloc` geworfen. Falls unzulässigerweise von der Klasse `T` abgeleitet und `new` für die abgeleitete Klasse aufgerufen wird, reagiert `getMemory()` mit einer `invalid_argument`-Exception. Voraussetzung für die Erkennung ist, dass die abgeleitete Klasse zusätzliche Attribute zu den geerbten hat. Wenn alles klar geht, wird die nächste freie Adresse in das Feld `belegt` eingetragen und zurückgegeben. `ersterFreierPlatz` wird auf die nächste Position hochgezählt. Weil nicht erst eine freie Stelle gesucht werden muss, ist `getMemory()` sehr schnell.

`getMemory()` dauert hingegen geringfügig länger, weil die Stelle gesucht wird, wo die Adresse `ptr` eingetragen wurde. Zuletzt angelegte Objekte werden nicht immer, aber häufig zuerst wieder gelöscht. Deshalb beginnt die Suche am Ende des Bereichs. Wenn der Eintrag gefunden wird, kann er gelöscht werden. Statt einen Null-Zeiger einzutragen, ist es besser, den letzten Verweis an diese Stelle zu kopieren und `ersterFreierPlatz` herunterzuzählen. Dadurch werden zukünftige Suchvorgänge verkürzt, und für `new` ist gar kein Suchvorgang erforderlich.

9.7.5 Empfehlungen im Umgang mit `new` und `delete`

`new` und `delete` gehören zusammen. Daraus ergeben sich einige Empfehlungen:

- Überladen Sie `new` und `delete` *nicht*, es sei denn, Sie haben gute Gründe dafür. Wenn die mangelnde Performance wegen vieler `new`-Aufrufe ein Grund sein sollte, wäre zuerst zu überlegen, ob die Anzahl nicht reduziert werden kann, zum Beispiel

durch Wiederverwendung bereits angelegter Objekte. Sie könnten mit einer passenden Funktion neu initialisiert werden.

- Wenn Sie `new` überladen, überladen Sie auch `delete` und umgekehrt. Dasselbe gilt für `new[]` und `delete[]`.
- Überladene `delete`- und `delete[]`-Operatoren dürfen keine Exception werfen. Beide Funktionen werden oft innerhalb eines Destruktors aufgerufen und rufen selbst Destruktoren auf. Wenn im ersten Fall eine Exception geworfen würde, wäre der Destruktor nicht mehr exception-sicher. Einzelheiten dazu finden Sie in Abschnitt 20.2.7.
- Verwenden Sie nur zusammenpassende Operatoren und Funktionen. Das heißt, dass ein mit `new` angelegtes Objekt nicht mit `free()` freigegeben werden darf. `delete` darf nicht auf einen mit `malloc()` erzeugten Speicherbereich angewendet werden.
- Falls Sie die Speicherbeschaffung in eine Oberklasse verlagern, vergessen Sie nicht, den Destruktor der Oberklasse als `virtual` zu deklarieren.
- Bei der Verwaltung selbst bereitgestellten Speichers darf die Ausrichtung an Speichergrenzen nicht vergessen werden, ebenso nicht, dass `delete` `NULL` keine Auswirkung haben darf.

9.8 Mehrdimensionale Matrizen²

Neben eindimensionalen Feldern sind 2- und 3-dimensionale Matrizen in mathematischen Anwendungen verbreitet. Der Sprache C entsprechende ein- und mehrdimensionale Arrays in C++ sind aus den Abschnitten 5.4 und 5.7 bekannt. Mathematische Matrizen sind Spezialfälle von Arrays mit Elementen, die vom Datentyp `int`, `float`, `complex`, `rational` oder ähnlich sind. Die Klasse `MathVektor` (Abschnitt 9.2.3) ist eine eindimensionale Matrix in diesem Sinne, wobei die Klasse im Unterschied zu einem normalen C-Array einen sicheren Zugriff über den Indexoperator erlaubt, wie wir dies auch für zwei- und mehrdimensionale Matrixklassen erwarten. Der Zugriff auf Elemente eines ein- oder mehrdimensionalen Matrixobjekts sollte

- sicher sein durch eine Prüfung aller Indizes und
- über den Indexoperator (bzw. `[]`, `[] []` ...) erfolgen, um die gewohnte Schreibweise beizubehalten.

Das Überladen des Klammeroperators für runde Klammern `()` wäre alternativ möglich, entspräche aber nicht der üblichen Syntax. Nun kann man sich streiten, ob es ästhetischer ist, `a(5,6)` zu schreiben anstatt `a[5][6]`. Sicherlich ist es bei neu zu schreibenden Programmen gleichgültig, doch wenn man für die Wartung und Pflege von existierenden großen Programmen verantwortlich ist, die die `[]`-Syntax verwenden? Ein weiteres Argument: Eine Matrixklasse soll sich möglichst ähnlich wie ein konventionelles C-Array

² Dieser Abschnitt kann beim ersten Lesen übersprungen werden. Es geht darum, wie die Aufeinanderfolge verketteter Indexoperatoren `[] [] []`, wie sie bei Matrizen üblich ist, mit dem überladenen Indexoperator realisiert wird, nicht um maximal effiziente Matrizen.

verhalten. Die auftretenden verketteten Aufrufe des Indexoperators werden im Einzelnen analysiert. Auch sollen die Indexoperatoren sicher sein. Etwas effizientere zweidimensionale Arrays, das heißt ohne Indexprüfung, werden an anderer Stelle behandelt (Seiten 215 und besonders 697 ff.).

Der Verzicht auf die erste Anforderung (Indexprüfung) wird oft mit Effizienzverlusten begründet. Dieses Argument ist nicht immer stichhaltig:

- Es ist wichtiger, dass ein Programm korrekt anstatt schnell ist. Indexfehler, das zeigt die industrielle Praxis, treten häufig auf. Das Finden der Fehlerquelle ist schwierig, wenn mit falschen Daten weitergerechnet und der eigentliche Fehler erst durch Folgefehler sichtbar wird.
- Der erhöhte Laufzeitbedarf durch einen geprüften Zugriff ist oft durchaus vergleichbar mit den weiteren Operationen, die mit dem Array-Element verbunden sind, und manchmal vernachlässigbar. Im Bereich der Natur- und Ingenieurwissenschaften gibt es einige Programme, bei denen sich die Indexprüfung deutlich nachteilig auswirkt, andererseits kommt es auf den Einzelfall an. Nur wenn ein Programm *wegen* der Indexprüfung zu langsam ist, sollte man nach gründlichen Tests erwägen, die Prüfung herauszunehmen.

Für rechenzeitoptimierte Vektoren und Matrizen sei auf entsprechende Bibliotheken verwiesen, von denen einige auf [OON] gelistet sind. C++ stellt die Klasse `valarray` zur Verfügung (Abschnitt 34, Seiten 857 ff). Zugunsten starker Optimierungsmöglichkeiten wurden beim Entwurf der Klasse `valarray` bewusst Einschränkungen bezüglich der Verständlichkeit und leichter Benutzbarkeit hingenommen. Eine schnelle 2-dimensionale Matrix wird in Abschnitt 24.10.1 diskutiert.

9.8.1 Zweidimensionale Matrix als Vektor von Vektoren

Was ist eine zweidimensionale Matrix, deren Elemente vom Typ `int` sind? Eine `int`-Matrix *ist ein* Vektor von mathematischen `int`-Vektoren! Diese Betrachtungsweise erlaubt eine wesentlich elegantere Formulierung einer Matrixklasse im Vergleich zur Aussage: Die Matrix *hat* beziehungsweise *besitzt* mathematische `int`-Vektoren. Die Formulierung der *ist-ein*-Relation als Vererbung zeigt die Klasse `Matrix`, wobei die bereits bekannten Klassen `vector` und `MathVektor` eingesetzt werden:

Listing 9.21: Matrix-Template

```
// cppbuch/k9/matrix/matrix.t
#ifndef MATRIX_T
#define MATRIX_T
#include "../mathvek/mvektor.t"

// Matrix als Vektor von MathVektoren
template<typename T>
class Matrix : public Vektor<MathVektor<T> > {
protected:
    size_t yDim;
public:
    typedef Vektor<MathVektor<T> > super;           // Oberklassentyp
    Matrix(size_t = 0, size_t = 0);                  // Zeilen, Spalten
    size_t zeilen() const {return super::size(); } // Zu super siehe Seite 332.
```

```

    size_t spalten() const {return yDim; }
    void init(const T&);
    // mathematische Operatoren und Funktionen
    Matrix<T>& I(); // als Einheitsmatrix initialisieren
    Matrix<T>& operator*=(const T&);
    Matrix<T>& operator*=(const Matrix<T>&);
    void swap(Matrix<T>& rhs); // Vertauschen
    // ... weitere Operatoren und Funktionen
};

template<typename T> // Implementierung des Konstruktors:
Matrix<T>::Matrix(size_t x, size_t y)
: Vektor<MathVektor<T> >(x, yDim(y) {
    MathVektor<T> temp(y);
    for(size_t i = 0; i < x; ++i) {
        super::operator[](i) = temp;
    }
} // ... Fortsetzung folgt unten

```

Die Klasse `Matrix` erbt also von der Klasse `Vektor`, wobei jetzt der Datentyp der Vektorelemente durch ein Template beschrieben wird. Der Konstruktor initialisiert ein Basisklassensubobjekt des Typs `vector<MathVektor<T> >` mit der richtigen Größe `x`. Weil dem Basisklassensubobjekt in der Initialisierungsliste die zweite Dimension `y` nicht bekannt ist, legt es `xDim` eigene Vektorelemente vom Typ `MathVektor<T>` mit der Länge 0 an. Die korrekte Länge `yDim` aller Vektorelemente wird in der nachfolgenden Schleife durch Zuweisung eines temporären Objekts `temp` mit der richtigen Länge `y` erreicht.

`Matrix` hat keinerlei dynamische Daten außerhalb des Basisklassensubobjekts. Deshalb sind weder ein besonderer Destruktor, Kopierkonstruktor noch ein eigener Zuweisungsoperator notwendig. Die entsprechenden Operationen für das Basisklassensubobjekt werden von der `vector`-Klasse erledigt. Allerdings soll die Klasse `Matrix` nicht über Zeiger benutzt werden, zum Beispiel:

```

Matrix<double> *pMatrix = new Matrix<double>(10, 10);
// ... Benutzung ...
delete pMatrix; // Speicherleck!

```

Der Grund liegt darin, dass die Klasse `vector` wie alle Container-Klassen der C++-Standardbibliothek keinen virtuellen Destruktor hat.



In Abschnitt 7.6.3 (Seite 280) finden Sie eine Begründung für den Fehler mit Beispiel.

Der Indexoperator wird geerbt; wie er funktioniert, wird unten beschrieben. Die Initialisierung der `Matrix` mit Werten (siehe Übungsaufgabe), der Kurzformoperator `*=` für die Multiplikation mit einem Skalar und die Erzeugung einer Einheitsmatrix sind ebenfalls einfach zu implementieren, wobei im `*=`-Operator geerbter Code wiederverwendet wird:

```

template<typename T> // Multiplikationsoperator
Matrix<T>& Matrix<T>::operator*=(const T& faktor) {
    for(size_t i = 0; i < super::size(); ++i) {
        super::operator[](i) *= faktor; // MathVektor<T>::operator*=(
    }
}

```

```

    return *this;
}

template<typename T>          // Einheitsmatrix
Matrix<T>& Matrix<T>::I() { // keine Prüfung auf size() == yDim
    for(size_t i = 0; i < super::size(); ++i) {
        for(size_t j = 0; j < yDim; ++j) {
            super::operator[](i)[j] = (i==j) ? T(1) : T(0);
        }
    }
    return *this;
}
// weitere Funktionen ...
#endif          // MATRIX_T

```

Nach diesem Schema können weitere Operatoren und Funktionen gebaut werden (siehe Übungsaufgaben). Nun fragt man sich, wie der Elementzugriff und die Indexprüfung in diesem Fall funktionieren? Betrachten wir folgendes Beispiel:

```

Matrix<float> matrix2D(100,55);
matrix2D[3][7] = 1.0162;

```

Der Zugriff ist sehr einfach, beide Indizes werden überprüft. Die Erklärung der Wirkungsweise ist jedoch nicht ganz so einfach. Um zu sehen, was geschieht, schreiben wir `matrix2D[3][7]` um und lösen dabei die Funktionsaufrufe auf:

```

(matrix2D.operator[](3)).operator[](7)

```

Die erste, von der Oberklasse geerbte Operatormethode ist

```

Vektor<MathVektor<float>> >::operator[](int)

```

Das anonyme Basisklassensubobjekt ist ein Vektor, dessen `[]`-Operator mit dem Argument 3 aufgerufen wird. Die Elemente des Vektors sind vom Typ `mathVektor<float>`; zurückgegeben wird also eine Referenz auf den dritten (d.h. eigentlich vierten wegen der Zählung ab 0) `mathVektor` des Vektors. Bezeichnen wir den Rückgabewert zur Vereinfachung mit `x`, dann wird jetzt

```

x.operator[](7)

```

ausgeführt, was nichts anderes bedeutet, als die Indexoperation `operator[]()` für einen `mathVektor<float>` mit dem Ergebnis `float&` – also einer Referenz auf das gesuchte Element – auszuführen. In jedem dieser Aufrufe von Indexoperatoren werden Grenzen auf einheitliche Weise geprüft. Abgesehen von der äquivalenten Definition für konstante Objekte existiert nur eine *einzige* Definition des Indexoperators! Auf die Definition weiterer Operatoren und sinnvoller Elementfunktionen soll nicht eingegangen werden.



Übungen

9.16 Implementieren Sie einen Operator

```

Matrix<T>& Matrix::operator*=(const Matrix<T>&)

```

zur Multiplikation einer Matrix mit einer anderen.

9.17 Implementieren Sie einen binären Operator zur Multiplikation zweier zweidimensionaler Matrizen unter der Annahme, dass bereits ein Elementoperator `*` existiert.

9.18 Schreiben Sie eine Methode `void init(const T&)` zur Initialisierung aller Matrixelemente. Sind dabei Methoden der Oberklasse einsetzbar?

9.8.2 Dreidimensionale Matrix

Das für zweidimensionale Matrizen benutzte Schema lässt sich nun zwanglos für Matrizen beliebiger Dimension erweitern. Hier sei nur noch abschließend das Beispiel für die dritte Dimension gezeigt. Was ist eine dreidimensionale Matrix, deren Elemente vom Typ `int` sind? Die Frage lässt sich leicht in Analogie zum vorhergehenden Abschnitt beantworten. Eine dreidimensionale `int`-Matrix *ist ein* Vektor von mathematischen zweidimensionalen `int`-Matrizen! Die Formulierung der *ist-ein*-Relation als Vererbung zeigt die Klasse `Matrix3D`:

Listing 9.22: 3-dimensionale Matrix

```
template<typename T>
class Matrix3D : public Vektor<Matrix<T> > {
private:
    size_t yDim, zDim;
public:
    typedef Vektor<Matrix<T> > super; // Oberklassentyp
    Matrix3D(size_t = 0, size_t = 0, size_t = 0);
    size_t xDIM() const { return super::size(); }
    size_t yDIM() const { return yDim; }
    size_t zDIM() const { return zDim; }
    void init(const T&); // Initialisierung
    Matrix3D<T>& I(); // Einheitsmatrix
    void swap(Matrix3D<T>& rhs); // Vertauschen
    // mathematischer Operator:
    Matrix3D<T>& operator*=(const T&); // Multiplikation
    // weitere Operatoren und Funktionen ...
};

template<typename T> // Implementierung des Konstruktors
Matrix3D<T>::Matrix3D(size_t x, size_t y, size_t z)
: Vektor<Matrix<T> >(x), yDim(y), zDim(z) {
    Matrix<T> temp(y, z);
    for(size_t i = 0; i < x; ++i) {
        super::operator[](i) = temp;
    }
}
```

Der Konstruktor initialisiert ein Basisklassensubobjekt des Datentyps `Vektor<Matrix<T> >` mit der richtigen Größe `x`. Weil dem Basisklassensubobjekt in der Initialisierungsliste die zweite und dritte Dimension nicht bekannt sind, legt es `xDim` eigene Elemente vom Typ `Matrix<T>` mit der Größe 0 mal 0 an. Die korrekte Größe aller Subobjekte wird in der nachfolgenden Schleife durch Zuweisung eines temporären Objekts `temp` erreicht. Weil `Matrix3D` (wie `Matrix`) keinerlei dynamische Daten außerhalb des Basisklassensubobjekts hat, ist weder ein besonderer Destruktor, Kopierkonstruktor noch ein eigener Zuweisungsope-

rator notwendig. Die entsprechenden Operationen für das Basisklassensubobjekt werden von der Klasse Vektor selbst erledigt. Der Indexoperator wird geerbt. Die Initialisierung, der Kurzformoperator *= und die Erzeugung einer Einheitsmatrix sind ähnlich wie oben beschrieben zu implementieren, wobei im *=-Operator geerbter Code wieder verwendet wird:

```
template<typename T>
void Matrix3D<T>::init(const T& wert) { // Initialisierung
    for(size_t i = 0; i < super::size(); ++i)
        for(size_t j = 0; j < yDim; ++j)
            for(size_t k = 0; k < zDim; ++k)
                super::operator[](i)[j][k] = wert;
}
```

Eine Alternative zur Initialisierung der einzelnen Elemente wie oben ist die Ausnutzung einer Methode der Klasse Matrix:

```
template<typename T>
void Matrix3D<T>::init(const T& wert) { // alternative Initialisierung
    for(size_t i = 0; i < super::size(); ++i) {
        super::operator[](i).init(wert);
        // operator[](i) ist vom Typ Matrix<T>
    }
}
```

Das kann ebenso vorteilhaft für die Bildung einer Einheitsmatrix ausgenutzt werden:

```
template<typename T> // Einheitsmatrix
Matrix3D<T>& Matrix3D<T>::I() { // keine Prüfung auf gleiche Dimensionen
    for(size_t i = 0; i < super::size(); ++i) {
        super::operator[](i).I();
    }
    return *this;
}

template<typename T> // Multiplikationsoperator
Matrix3D<T>& Matrix3D<T>::operator*=(const T& faktor) {
    for(size_t i = 0; i < super::size(); ++i)
        super::operator[](i) *= faktor; // Matrix<T>::operator*=(T)
    return *this;
}
#endif // matrix_t
```

Nun können auf einfache Art dreidimensionale Matrizen definiert und benutzt werden, zum Beispiel:

```
Matrix3D<int> matrix3D(3, 15, 2);
for(size_t i = 0; i < matrix3D.xDIM(); ++i)
    for(size_t j = 0; j < matrix3D.yDIM(); ++j)
        for(size_t k = 0; k < matrix3D.zDIM(); ++k) {
            matrix3D[i][j][k] = was_auch_immer();
            cout << matrix3D[i][j][k];
        }
```

```
// Benutzung
matrix3D *= 1020;
// .... usw.
```

Die Wirkungsweise des Indexoperators ist in Analogie zur Klasse `Matrix` beschreibbar, es gibt nur einen verketteten Operatoraufruf mehr. Wir formulieren `matrix3D[1][2][3]` um und erhalten: `matrix3D.operator[] (1).operator[] (2).operator[] (3)`

Der erste Operator gibt etwas vom Typ `Matrix<int>&` zurück oder genauer eine Referenz auf das erste Element des Vektor-Subobjekts von `matrix3D` (die dreidimensionale Matrix ist ein Vektor von zweidimensionalen Matrizen). Das zurückgegebene »Etwas« kürzen wir der Lesbarkeit halber mit `Z` ab und erhalten `Z.operator[] (2).operator[] (3)`

Sie wissen, dass eine Referenz nur ein anderer Name ist, sodass `Z` letztlich eine Matrix des Typs `Matrix<int>` repräsentiert. Sie sahen bereits, dass eine `Matrix<int>` ein Vektor ist, nämlich ein `Vektor< MathVektor<int> >`, von dem `operator[] ()` geerbt wurde. Genau dieser Operator wird nun mit dem Argument 2 aufgerufen und gibt ein Ergebnis vom Typ `MathVektor<int>&` zurück, das hier der Kürze halber `x` heißen soll: `x.operator[] (3)`

Der Rest ist leicht, wenn Sie an das Ende des Abschnitts über zweidimensionale Matrizen schauen. Auch hier ist wie bei der Klasse `Matrix` der Zugriff auf ein Element simpler als die darunterliegende Struktur.

Schlussbemerkung

Die Methode zur Konstruktion der Klassen für mehrdimensionale Matrizen kann leicht verallgemeinert werden: Eine n -dimensionale Matrix kann als Vektor von $(n - 1)$ -dimensionalen Matrizen aufgefasst werden. Die Existenz einer Klasse für $(n - 1)$ -dimensionale Matrizen sei dabei vorausgesetzt. In der Praxis werden jedoch vier- und höherdimensionale Matrizen selten eingesetzt. Indexoperator, Zuweisungsoperator, Kopierkonstruktor und Destruktor brauchen nicht geschrieben zu werden, sie werden von der Klasse `Vektor` zur Verfügung gestellt. Geschrieben werden müssen jedoch der Konstruktor, die Methoden zur Initialisierung und die gewünschten mathematischen Operatoren. Die beschriebenen Klassen zeigen die Kombination von Templates mit Vererbung. Beim Zugriff auf einzelne Matrix-Elemente wird kontrolliert, ob der Index im zulässigen Bereich liegt – das kostet natürlich Laufzeit.

9.9 Zuweisung bei Vererbung³

Wenn kein spezieller Zuweisungsoperator definiert ist, wird automatisch vom Compiler ein Zuweisungsoperator bereitgestellt, der ein Objekt elementweise kopiert. Elementweise Kopie meint, dass für jedes Element der zugehörige Zuweisungsoperator aufgerufen wird, sei er nun selbst definiert oder implizit generiert worden. Insbesondere wird bei einem Objekt einer abgeleiteten Klasse das anonyme Subobjekt der Oberklasse in diesem Sinn als Element aufgefasst. Dies ist alles einfach, wenn Polymorphismus aus dem Spiel bleibt.

³ Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

Der Tipp auf Seite 273 soll sicherstellen, dass sich die Bedeutung eines Methodenaufrufs nicht ändert, wenn auf eine Methode statt über den Objektnamen über Basisklassenzeiger bzw. -referenzen zugegriffen wird. Wie sieht es aus, wenn polymorphe Objekte einander zugewiesen werden sollen? Anhand der Klasse *Ober* und der abgeleiteten Klasse *Unter* soll das Problem verdeutlicht werden:

```
// Klassendefinitionen
class Ober {
public:
    // ....
private:
    int i0;           // Daten der Klasse Ober
};

class Unter : public Ober {
public:
    // ....
private:
    int iU;           // Daten der Klasse Unter
};
```

```
// main()-Programm, Auszug:
Unter U1, U2;
U1 = U2;           // kein Problem: Unter::operator=(const Unter&)
// polymorphe Objekte (statischer ≠ dynamischer Typ)
Ober& r01 = U1;    // polymorphe Zuweisung
Ober& r02 = U2;
```

Falls der Zuweisungsoperator nicht virtuell ist, werden nur statische Typen betrachtet:

```
Ober O1;
Ober O2;
O1 = O2;           // Ober::operator=(const Ober&) ok
```

Aus der Zuweisung

```
r01 = U2;           // Ober::operator=(const Unter&) ?
```

versucht der Compiler, einen Zuweisungsoperator zu finden. In einer Oberklasse darf es so einen Operator nicht geben, weil nicht garantiert ist, dass in der Oberklasse Informationen über eine Unterklasse existieren. Der Compiler nimmt den nächst passenden Operator, nämlich `Ober::operator=(const Ober&)`, weil `r01` vom statischen Typ `Ober` ist. Der Fehler: Dem Objekt `U1`, auf das die Referenz `r01` verweist, wird *nur der Oberklassen-anteil* von `U2` zugewiesen, das heißt, `U1` und `U2` unterscheiden sich nach der Zuweisung! Um den richtigen Zuweisungsoperator aufzurufen, wenn statischer und dynamischer Typ verschieden sind, muss der Zuweisungsoperator virtuell sein. Weil der implizit generierte Zuweisungsoperator niemals virtuell ist, heißt dies, dass sich der Softwareentwickler selbst dieser Mühe unterziehen muss.

Bei virtuellen Methoden müssen Name und Parameterlisten übereinstimmen, nur der Rückgabotyp darf etwas unterschiedlich sein: Wenn der Rückgabotyp einer virtuellen Funktion eine Referenz auf die Klasse ist, dann darf der Rückgabotyp der entsprechenden

Funktion in der abgeleiteten Klasse eine Referenz auf die abgeleitete Klasse sein. Ein Beispiel sind die Funktionen `vf4()` und `vf5()` auf Seite 274.

```
// Annahme: virtueller Zuweisungsoperator
O1 = O2;    // Ober::operator=(const Ober&) wie vorher
U1 = rO2;   // Unter::operator=(const Ober&)
rO1 = U2;   // Unter::operator=(const Ober&)
```

Leider tauchen bei der Implementierung nun weitere Probleme auf:

```
virtual Unter& Unter::operator=(const Ober& rs) {
    // rs = rechte Seite der Zuweisung
    // geerbte Attribute der Oberklasse zuweisen:
    iO = rs.iO;    // Fehler! iO ist private.
    // Attribut der Klasse Unter zuweisen:
    iU = rs.iU;    // Fehler! rs.iU ist unbekannt!
    return *this;
}
```

Der erste Fehler kann leicht behoben werden: Wenn die Attribute der Oberklasse `protected` sind, ist die Zuweisung möglich. Nun wird dadurch der Zugriffsschutz aufgeweicht, und deshalb ist es besser, wenn die Oberklasse eine Methode zur lokalen Zuweisung ihrer Attribute bereitstellt, die die privaten Daten kopiert. Der zweite Fehler resultiert aus der Tatsache, dass der Compiler nur eine statische Analyse vornehmen kann: Danach hat das Objekt `rs`, das zur Klasse `Ober` gehört, kein Attribut `iU`. Benötigt wird jedoch zur Laufzeit die Information über den dynamischen Typ von `rs`, der sowohl `Ober` als auch `Unter` sein kann. Nur in letzterem Fall ist eine Zuweisung überhaupt sinnvoll.

Mit den bisher bekannten Mitteln ist dem Problem der Zuweisung bei Polymorphismus nicht beizukommen. Eine brutaler Downcast, also eine Typumwandlung der Art

```
iU = static_cast<Unter&>(rs).iU; // gefährlich
```

verhindert zu erkennen, dass `rs` vielleicht doch den falschen Typ hat. Aus diesem Grund wurde der `dynamic_cast`-Operator eingeführt, der zur Laufzeit eine Typumwandlung vornimmt und dabei prüft, ob diese Typumwandlung erlaubt ist (siehe Abschnitt 7.9).

Man könnte denken, dass ein virtueller Zuweisungsoperator genügen würde. Das ist nicht der Fall, weil der Zuweisungsoperator anders als »normale« virtuelle Funktionen aufgerufen wird. Wenn nur ein virtueller Zuweisungsoperator geschrieben wurde, wird bei der Compilation der nicht-virtuelle automatisch auch noch erzeugt – und ggf. statt des virtuellen aufgerufen (siehe Abschnitt 13.5.3 in [ISOC++]). Das ist kein Problem, wenn die abgeleitete Klasse keinerlei dynamische Daten hat. Wenn es jedoch Zeigerattribute gibt, denen mit `new` etwas zugewiesen wird, ist ein selbstgeschriebener nicht-virtueller Zuweisungsoperator unumgänglich, weil der compilergenerierte nur den Zeiger kopieren würde, ohne Speicherplatz zu beschaffen. Im Zusammenspiel von nicht-virtuellem und virtuellem Zuweisungsoperator ergeben sich weitere Schwierigkeiten, auf die hier nicht näher eingegangen werden soll. Es gibt nur eine funktionierende Strategie, die im Folgenden beschrieben wird.

Bei dieser Strategie bekommt jede Klasse eine virtuelle Methode `assign()` und einen nicht-virtuellen `operator=()`. Bei polymorpher Zuweisung ruft der Zuweisungsoperator

`assign()` auf. Weil die Methode virtuell ist, wird sie automatisch passend zum Objekt aufgerufen.

Wenn dynamische Daten vorhanden sind, sind stets ein Kopierkonstruktor, ein Zuweisungsoperator und ein Destruktor erforderlich. Zur Demonstration hat jede Klasse des folgenden Beispiels sowohl ein Attribut vom Typ `int` als auch ein dynamisches Attribut vom Typ `char*` (C-String). Dem letzten Attribut wird mit `new` Speicherplatz zugewiesen, der im Destruktor freigegeben wird. Das Beispiel hat weitere Besonderheiten:

- Es gibt drei Klassen A, B und C. Dabei erbt B von A, und C erbt von B.
- Wegen der dynamischen Daten existiert für jede Klasse ein Kopierkonstruktor, ein Zuweisungsoperator und ein Destruktor.
- Der Zuweisungsoperator ruft in den abgeleiteten Klassen die virtuelle Methode `assign()`, in der die eigentliche Kopierarbeit geleistet wird. Damit der übergebene Typ stimmt, wird er mit `static_cast` umgewandelt. Ein `dynamic_cast` ist nicht notwendig, weil durch den polymorphen Aufruf der Methode `assign()` gewährleistet ist, dass der Typ zur Klasse passt.
- Jede Klasse hat eine `swap`-Methode, die nach bekanntem Muster innerhalb der Methode `assign()` eingesetzt wird.

Das `main`-Programm demonstriert mögliche Testfälle. Die Gleichheit der beteiligten Objekte wird mit `assert` geprüft. Dabei kommt der virtuelle Gleichheitsoperator `operator==()` zum Einsatz. Wegen der `virtual`-Eigenschaft ist er als Mitgliedsfunktion konzipiert. Das hat den Vorteil des direkten Zugriffs auf die Attribute, sodass `getter`-Funktionen in diesem Beispiel entfallen können.

Listing 9.23: `main`-Programm zum Testen der Zuweisungen

```
// cppbuch/k9/zuweisung_vererbung/zuassign.cpp
#include<cassert>
#include<iostream>
#include"C.h" // schließt A.h, B.h ein
using namespace std;

int main() {
    // Klasse A
    cout << "\nTest 1" << endl;
    A a1(1, "einsA");
    A a2(2, "zweiA");
    a1.ausgabe(); cout << endl;
    a1 = a2;
    cout << "a1 nach Zuweisung a1=a2:\n";
    a1.ausgabe(); cout << endl;
    assert(a1 == a2);

    // Klasse B
    cout << "\nTest 2" << endl;
    B b1(1, "einsA", 2, "einsB");
    B b2(3, "zweiA", 4, "zweiB");
    b1.ausgabe(); cout << endl;
    b1 = b2;
    cout << "b1 nach Zuweisung b1=b2:\n";
    b1.ausgabe(); cout << endl;
```

```

assert(b1 == b2);

cout << "\nTest 3 polymorphe Zuweisung"<< endl;
B b3(5, "dreia", 6, "dreib");
A& ar = b1;           // Oberklassenreferenz
ar = b3;
cout << "ar nach Zuweisung ar=b3:\n";
ar.ausgabe(); cout << endl;
assert(ar == b3);

// Klasse C
cout << "\nTest 4" << endl;
C c1(1, "einsA", 2, "einsB", 3, "einsC");
C c2(4, "zweiA", 5, "zweiB", 6, "zweiC");
c1.ausgabe(); cout << endl;
c1 = c2;
cout << "c1 nach Zuweisung c1=c2:\n";
c1.ausgabe(); cout << endl;
assert(c1 == c2);

cout << "\nTest 5 : polymorphe Zuweisung A&t = C"<< endl;
C c3(7, "dreia", 8, "dreib", 9, "dreic");
A& arc = c2;          // Oberklassenreferenz
arc = c3;
cout << "arc nach Zuweisung arc=c3:\n";
arc.ausgabe(); cout << endl;
assert(arc == c3);

cout << "\nTest 6 : polymorphe Zuweisung B&t = C"<< endl;
B& brc(c2);           // Oberklassenreferenz
brc = c3;
cout << "brc nach Zuweisung brc=c3:\n";
brc.ausgabe(); cout << endl;
assert(brc == c3);

cout << "\nTest 7 : falscher Typ: b1 = c3"<< endl;
try {
    b1 = c3;
}
catch(const bad_typeid& e) {
    cout << "Falscher Typ! Exception: " << e.what() << endl;
}

cout << "\nTest 8 : falscher Typ: a1 = c3"<< endl;
try {
    a1 = c3;
}
catch(const bad_typeid& e) {
    cout << "Falscher Typ! Exception: " << e.what() << endl;
}
cout << "Test-Ende" << endl;
}

```

Es folgen die verwendeten Klassen, beginnend mit der obersten Basisklasse:

Listing 9.24: Basisklasse

```
// cppbuch/k9/zuweisung_vererbung/A.h
#ifndef A_H
#define A_H
#include<algorithm> // swap
#include<cstring>
#include<iostream>
#include<typeinfo>

using std::cout;
using std::endl;

class A {
public:
    A(int a_, const char* as_) : a(a_), as(new char[strlen(as_)+1]) {
        strcpy(as, as_);
    }

    A(const A& rhs) : a(rhs.a), as(new char[strlen(rhs.as)+1]) {
        strcpy(as, rhs.as);
    }

    virtual ~A() { delete [] as; }

    virtual A& assign(const A& rhs) {
        cout << "virtual A& A::assign(const A&)" << endl;
        if(typeid(*this) != typeid(rhs)) { // siehe Text unten
            throw std::bad_typeid();
        }
        A temp(rhs);
        swap(temp);
        return *this;
    }

    A& operator=(const A& rhs) {
        cout << "A& operator=(const A& rhs)" << endl;
        return assign(rhs);
    }

    virtual void ausgabe() const {
        cout << "A.a = " << a << ", A.as = " << as << " ";
    }

    void swap(A& rhs) {
        std::swap(a, rhs.a);
        std::swap(as, rhs.as);
    }

    virtual bool operator==(const A& arg) const {
        return typeid(*this) == typeid(arg) // siehe Text unten
            && a == arg.a && strcmp(as, arg.as) == 0;
    }
};
```

```

    }
private:
    int a;
    char* as;
};
#endif

```

In C++ ist die Zuweisung `base = derived`, erlaubt. Dabei sei `base` ein Objekt der Basis-klasse und `derived` ein Objekt einer abgeleiteten Klasse. Weil nur der Basisklassenanteil zugewiesen wird, wird so eine Anweisung nicht erwünscht sein. Aus diesem Grund wird mit `typeid` geprüft, ob der linke und der rechte Typ zur Laufzeit übereinstimmen. Falls nicht, wird eine Exception geworfen.

Ähnliches gilt für den Vergleichsoperator `==`. Bei dem Vergleich `base == derived`, wird normalerweise nur geprüft, ob die Basisklassenanteile im Objekt `derived` mit denen im Objekt `base` übereinstimmen. Um nicht zu ignorieren, dass es sich möglicherweise trotzdem um ganz verschiedene Typen handelt, wird `typeid` eingesetzt.

Weil der Vergleich bei abgeleiteten Klassen die Basisklasse einschließt, muss die Prüfung mit `typeid` in abgeleiteten Klassen nicht wiederholt werden. Das gilt nicht für `assign()`.

Listing 9.25: Abgeleitete Klasse B

```

// cppbuch/k9/zuweisung_vererbung/B.h
#ifndef B_H
#define B_H
#include "A.h"

class B : public A {
public:
    B(int a_, const char* as_, int b_, const char* bs_)
        : A(a_, as_), // Oberklassen-Subobjekt
          b(b_), bs(new char[strlen(bs_)+1]) { // lokale Daten
        strcpy(bs, bs_);
    }

    B(const B& rhs)
        : A(rhs), // Oberklassen-Subobjekt
          b(rhs.b), bs(new char[strlen(rhs.bs)+1]) { // lokale Daten
        strcpy(bs, rhs.bs);
    }

    ~B() { delete [] bs; }

    virtual B& assign(const A& rhs) {
        cout << "virtual B& B::assign=(const A&)" << endl;
        if(typeid(*this) != typeid(rhs)) {
            throw std::bad_typeid();
        }
        B temp(static_cast<const B&>(rhs));
        swap(temp);
        return *this;
    }
}

```



```

B& operator=(const B& rhs) {
    cout << "B& operator=(const B& rhs)" << endl;
    return assign(rhs);
}

virtual void ausgabe() const {
    A::ausgabe();
    cout << "B.b = " << b << ", B.bs = " << bs << " ";
}

void swap(B& rhs) {
    A::swap(rhs);           // Oberklassendaten
    std::swap(b, rhs.b);    // lokale Daten
    std::swap(bs, rhs.bs);  // lokale Daten
}

bool operator==(const A& arg) const {
    const B& rarg = static_cast<const B&>(arg);
    return A::operator==(arg) &&
        b == rarg.b && strcmp(bs, rarg.bs) == 0;
}
private:
    int b;
    char* bs;
};
#endif

```

Listing 9.26: Abgeleitete Klasse C

```

// cppbuch/k9/zuweisung_vererbung/C.h
#ifndef C_H
#define C_H
#include "B.h"

class C : public B {
public:
    C(int a_, const char* as_, int b_, const char* bs_,
        int c_, const char* cs_)
        : B(a_, as_, b_, bs_),           // Oberklassen-Subobjekt
          c(c_), cs(new char[strlen(cs_)+1]) { // lokale Daten
        strcpy(cs, cs_);
    }

    C(const C& rhs)
        : B(rhs),                       // Oberklassen-Subobjekt
          c(rhs.c), cs(new char[strlen(rhs.cs)+1]) { // lokale Daten
        strcpy(cs, rhs.cs);
    }

    ~C() { delete [] cs; }

    virtual C& assign(const A& rhs) {
        cout << "virtual C& C::assign=(const A&)" << endl;
    }
}

```

```

        if(typeid(*this) != typeid(rhs)) {
            throw std::bad_typeid();
        }
        C temp(static_cast<const C&>(rhs));
        swap(temp);
        return *this;
    }

    C& operator=(const C& rhs) {
        cout << "C& operator=(const C& rhs)" << endl;
        return assign(rhs);
    }

    virtual void ausgabe() const {
        B::ausgabe();
        cout << "C.c = " << c << ", C.cs = " << cs << " ";
    }

    void swap(C& rhs) {
        B::swap(rhs);           // Oberklassendaten
        std::swap(c, rhs.c);    // lokale Daten
        std::swap(cs, rhs.cs);  // lokale Daten
    }

    bool operator==(const A& arg) const {
        const C& rarg = static_cast<const C&>(arg);
        return B::operator==(arg) &&
            c == rarg.c && strcmp(cs, rarg.cs) == 0;
    }
private:
    int c;
    char* cs;
};
#endif

```

Aus diesem Abschnitt ergeben sich einige Empfehlungen, die beim Einsatz von Vererbung beherzigt werden sollten, es sei denn, Polymorphismus wird mit deutlichen Hinweisen und ausführlicher Dokumentation allen Benutzern einer Oberklasse verboten:

- Der nicht virtuelle Zuweisungsoperator muss existieren. Er ruft eine spezielle virtuelle Methode `assign()` auf, die die Zuweisung erledigt. Beispiel für eine Klasse `X`:
`X& operator=(const X& rhs) { return assign(rhs); }`
- Jede Klasse hat praktischerweise eine `swap()`-Methode, die in der Zuweisung benutzt wird. Sie sorgt für die Vertauschung der Oberklassendaten und der lokalen Daten.
- Jede abgeleitete Klasse Unterklasse, egal an welcher Stelle der Vererbungshierarchie, sollte eine (in der obersten Basisklasse deklarierte) virtuelle Methode `assign()` mit der folgenden Struktur haben:

```

virtual Unterklasse& assign(const obersteBasisklasse& rhs) {
    Unterklasse temp(static_cast<const Unterklasse&>(rhs));
    swap(temp);
    return *this;
}

```

Die Methode `assign()` ist verantwortlich für die Zuweisung eines vollständigen Objekts der betreffenden Klasse und hat als Rückgabotyp eine Referenz dieser Klasse. Der Aufruf bewirkt bei dieser Struktur die Zuweisung aller anonymen Subobjekte der Oberklassen bis hin zur (obersten) Basisklasse.

- Nur wenn die Operationen `base = derived`, als Fehler betrachtet wird, müssen in den `assign()`-Methoden die Typen mit `typeid` geprüft werden.
- Nur wenn der Vergleich `(base == derived)` `false` ergeben soll, auch wenn die Basis-klassenanteile links und rechts übereinstimmen, müssen im Vergleichsoperator der obersten Basisklasse die Typen mit `typeid` geprüft werden. Hier genügt die oberste Basisklasse, weil ein Vergleichsoperator einer Klasse den seiner direkten Oberklasse aufruft. Ohne Typprüfung ist der Vergleichsoperator nicht symmetrisch, d.h. `(base == derived)` kann `true` sein und das umgekehrte `false` (wobei dann ohne Typprüfung auf undefinierte Bereiche zugegriffen wird).

Manche empfehlen, vorher auf Identität zu prüfen, etwa:

```
virtual Unterklasse& assign(const obersteBasisklasse& rhs) {
    if(this != &rhs) {    // wirklich notwendig?
        Unterklasse temp(dynamic_cast<const Unterklasse&>(rhs));
        swap(temp);
    }
    return *this;
}
```

Eine Zuweisung der Art `x = x`, würde dann den Kopieraufwand vermeiden. Selbstzuweisung ist aber so selten, dass meiner Meinung nach gut auf die zusätzliche Abfrage verzichtet werden kann.

Schlussbemerkung

Das Beispiel zeigt auch, dass der Gleichheitsoperator virtuell sein kann, und dass ein nicht-virtueller Gleichheitsoperator nicht benötigt wird.

Der Zuweisungsoperator ist bei Vererbung und polymorpher Nutzung etwas komplizierter als ohne Vererbung. Bei Mehrfachvererbung ist noch zu beachten, dass die Subobjekte verschiedener Oberklassen richtig zugewiesen werden, und dass im Fall einer obersten Basisklasse (Stichwort virtuelle Vererbung) darauf geachtet werden muss, dass das Basis-klassensubobjekt nur einmal zugewiesen wird. Dieses Thema ist aber zu speziell, um es hier zu vertiefen, und es gibt vermutlich nur sehr wenige Anwendungsfälle dafür. Wer trotzdem etwas darüber wissen möchte, findet in [\[Br\]](#) Informationen dazu.

10

Dateien und Ströme

Dieses Kapitel behandelt die folgenden Themen:

- Formatierung der Ausgabe
- Funktionsweise der Standardeingabe
- Manipulatoren - Wirkungsweise und Konstruktion
- Fehlerbehandlung bei der Ein- und Ausgabe
- Status einer Datei
- Ansteuern bestimmter Positionen einer Datei
- Nutzung von `cin` für eigene Klassen
- Ausgabe in einen String umleiten

In Kapitel 2 wird die Ein- und Ausgabe einführend beschrieben. In diesem Kapitel wird das Thema vertieft, indem weitere nützliche Funktionen und Operatoren angegeben werden sowie die Formatierung von Daten besprochen und auf die Fehlerbehandlung eingegangen wird. Die C++-Standardbibliothek stellt die notwendigen Mechanismen für die Ein- und Ausgabe von Grunddatentypen zur Verfügung, erlaubt aber auch die Konstruktion eigener Ein- und Ausgabefunktionen für selbst definierte Klassen. Auf der untersten Ebene wird ein *stream* als Strom oder Folge von Bytes aufgefasst. Das Byte ist die Dateneinheit des Stroms, andere Datentypen wie `int`, `char*` oder `vector` erhalten erst durch die Bündelung und Interpretation von Bytesequenzen auf höherer Ebene ihre Bedeutung. Die Basisklasse heißt `ios_base`; aus ihr werden die anderen Klassen abgeleitet, wie Abbildung 10.1 zeigt. Die dort mit dem Wort `basic` beginnenden Klassen sind Templates, die für beliebige Zeichentypen geeignet sind, zum Beispiel Unicode-Zeichen oder andere »Wide Characters«, die auf Seite 51 erwähnt werden. Für den am häufigsten benötigten Datentyp `char` sind die Klassen durch Typdefinitionen wie

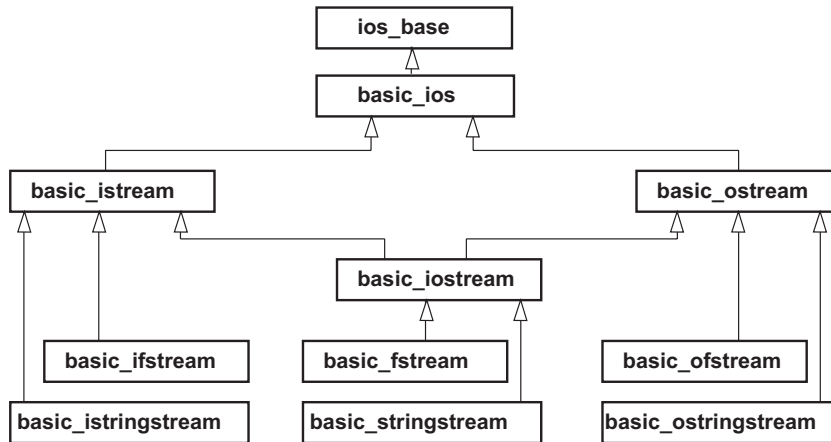


Abbildung 10.1: Hierarchie der Klassen-Templates für die Ein- und Ausgabe (Auszug)

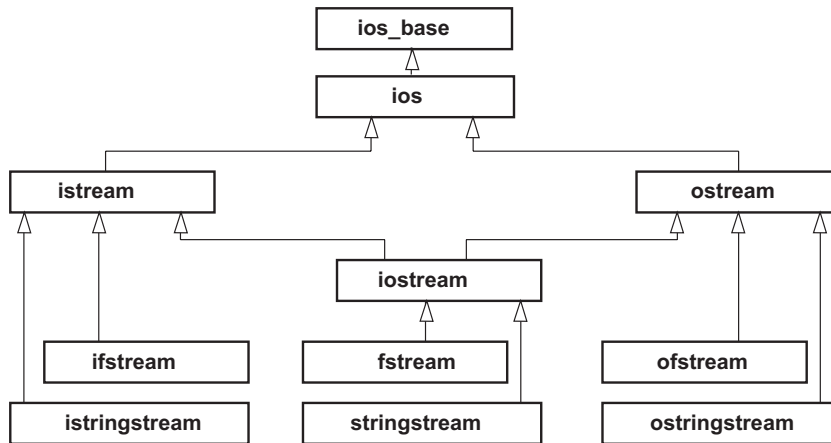


Abbildung 10.2: Spezialisierte Klassen für `char`

```
typedef basic_ofstream<char> ofstream;
```

spezialisiert worden, wie Abbildung 10.2 zeigt. Im Folgenden liegt der Schwerpunkt auf der Benutzung der Klassen. Die im Header `<iostream>` deklarierten Standardstreams sind Objekte dieser Klassen:

```
namespace std {
    // Auszug aus <iostream>
    extern istream cin;      // Standardeingabe
    extern ostream cout;    // Standardausgabe
    extern ostream cerr;    // Standardfehlerausgabe
    extern ostream clog;    // gepufferte Standardfehlerausgabe
}
```

10.1 Ausgabe

Die Klasse `ostream` umfasst überladene Operatoren für eingebaute Datentypen. Der Operator `<<` dient dazu, ein Objekt eines internen Datentyps in eine Folge von ASCII-Zeichen zu verwandeln.

```
ostream& operator<<(const char*); // C-Strings
ostream& operator<<(char);
ostream& operator<<(int);
ostream& operator<<(float);
ostream& operator<<(double);
//... usw.
```

Der Rückgabtyp des Operators ist eine *Referenz* auf `ostream`. Das Ergebnis des Operators ist das `ostream`-Objekt selbst, sodass ein weiterer Operator darauf angewendet werden kann. Damit ist die Hintereinanderschaltung von Ausgabeoperatoren möglich:

```
cerr << "x= " << x;
```

wird interpretiert als

```
(cerr.operator<<("x= ")).operator<<(x);
```

Ausgabe benutzerdefinierter Typen

Überladen von Operatoren ermöglicht die Ausgabe beliebiger Objekte benutzerdefinierter Klassen. Ein ausführliches Beispiel dazu wird auf den Seiten 322 ff. gezeigt, wo es darum geht, Objekte der Klasse `rational` auszugeben. Nach diesem Muster sind eigenen Kreationen des `<<`-Operators keine Grenzen gesetzt.

Ausgabefunktionen

In der Klasse `ostream` sind weitere Elementfunktionen für `ostream`-Objekte definiert:

```
ostream& put(char);
```

gibt ein Zeichen aus. Wir haben `put()` bereits in Abschnitt 2.2 kennengelernt, wo die Funktion in einem Beispielprogramm zum Kopieren von Dateien eingesetzt wird. Die Funktion

```
ostream& write(const char*, size_t);
```

wird in Abschnitt 5.8 zur binären Ausgabe verwendet. Es gibt eine Menge zusätzlicher Funktionen, zum Beispiel zum Positionieren des Ausgabestroms auf eine bestimmte Stelle, zum Herausfinden der aktuellen Stelle, und weitere, auf die hier zum Teil eingegangen werden soll. Die Deklarationen dieser Funktionen finden Sie im Header `<iostream>`. Eine ausführliche Beschreibung findet sich in dem englischsprachigen Buch [KL]).

10.1.1 Formatierung der Ausgabe

Oft ist es notwendig, die Ausgabe besonders aufzubereiten, sei es als Tabelle, in der alle Spalten die gleiche Breite haben müssen, oder sei es, dass spezielle Zahlenformate

verlangt werden. Dieser Abschnitt stellt die wichtigsten Möglichkeiten zur Formatierung der Ausgabe vor.

Weite und Füllzeichen

Die Methode `width()` bestimmt die Weite der *unmittelbar folgenden* Zahlen- oder C-Stringausgabe. Dabei entstehende Leerplätze werden mit Leerzeichen aufgefüllt. Anstelle der Leerzeichen können mittels der Funktion `fill()` andere Füllzeichen definiert werden. Das Programmstück

```
cout.width(6);
cout.fill('0');
cout << 12 << ' ' << 34 << ' ';
```

erzeugt die Ausgabe 000012(34). Die Weite wurde auf 6 gesetzt, mit 0 als Füllzeichen. Weil die Weite aber bei jeder Ausgabe auf 0 zurückgesetzt wird, erscheint (34) ohne Füllzeichen. Es ist erkennbar, dass mindestens die notwendige Weite genommen wird, dass also eine zu kleine Angabe für die Weite die Ausgabe nicht beschränkt:

```
cout.width(4);
cout << 123456;
```

ergibt 123456, und nicht etwa 1234 oder ****.

Steuerung der Ausgabe über Flags

Ein *Flag* (englisch für *Flagge*, *Fahne*) ist ein Zeichen für ein Merkmal, das entweder vorhanden (Flag ist gesetzt) oder nicht vorhanden ist (Flag ist nicht gesetzt). Zur Formatsteuerung sind Flags des (compilerabhängigen) Datentyps `ios_base::fmtflags` definiert, der als Bitmaske benutzt wird. Die Tabelle 10.1 zeigt eine Aufstellung.

Weil die Klasse `ios` von `ios_base` erbt und im Folgenden nur die auf den Typ `char` spezialisierten Klassen der Ein- und Ausgabe betrachtet werden, wie die Abbildung 10.2 zeigt, können alle öffentlichen Attribute und Methoden auf die Klasse `ios` bezogen werden. `ios::fmtflags` ist dasselbe wie `ios_base::fmtflags`, weil `ios_base` kein Template ist. Statt `ios_base` wird im Folgenden nur noch `ios` geschrieben. Dieses Vorgehen hat den Vorteil der weitgehenden Kompatibilität mit früheren C++-Versionen und der einfacheren Schreibweise.

Im folgenden Programmbeispiel wird gezeigt, wie die Flags gelesen und gesetzt werden können. Die typische Voreinstellung mit Zweierpotenzen im Typ `fmtflags` definiert eine Bitleiste, deren einzelne Bits durch Oder-Operationen gesetzt werden. Die Funktion `flags()` dient zum Lesen und Setzen *aller* Flags, und `setf()` wird zum Setzen *eines* Flags verwendet.

```
ostream Ausgabe;
ios::fmtflags altesFormat;
ios::fmtflags neuesFormat = ios::left|ios::oct|ios::showpoint|ios::fixed;
// gleichzeitiges Lesen und Setzen aller Flags
altesFormat = Ausgabe.flags(neuesFormat);
// Setzen eines Flags
Ausgabe.setf(ios::hex);    // ungünstig, siehe unten
// gleichwertig damit ist:
```

Tabelle 10.1: Formateinstellungen für die Ausgabe

Name	Bedeutung
boolalpha	true/false alphabetisch ausgeben oder lesen
skipws	Zwischenraumzeichen ignorieren
left	linksbündige Ausgabe
right	rechtsbündige Ausgabe
internal	zwischen Vorzeichen und Wert auffüllen
dec	dezimal
oct	oktal
hex	hexadezimal
showbase	Basis anzeigen
showpoint	nachfolgende Nullen ausgeben
uppercase	E,X statt e,x
showpos	+ bei positiven Zahlen anzeigen
scientific	Exponential-Format
fixed	Gleitkomma-Format
	Puffer leeren (flush):
unitbuf	– nach jeder Ausgabeoperation
stdio	– nach jedem Textzeichen

```
Ausgabe.flags(Ausgabe.flags() | ios::hex);
// Zurücksetzen eines Flags
Ausgabe.unsetf(ios::hex);
```

Wenn ein Flag gesetzt werden soll, ist es besser, sicherheitshalber möglicherweise kollidierende Flags zurückzusetzen. Die Funktion `setf()` mit *zwei* Parametern sorgt dafür:

```
Ausgabe.setf(dieFlags, Maske);
```

bewirkt, dass zuerst alle Bits der Maske zurückgesetzt und danach die Flags gesetzt werden. Es gibt drei vordefinierte Masken für diesen Zweck, die in der Tabelle 10.2 zusammengefasst sind. Wenn die hexadezimale Ausgabe eingestellt werden soll, ist es daher besser, `Ausgabe.setf(ios::hex, ios::basefield)` zu schreiben, um gleichzeitig die möglicherweise gesetzten Flags `oct` oder `dec` zurückzusetzen.

Tabelle 10.2: Vordefinierte Bitmasken

Name	Wert
adjustfield	left right internal
basefield	oct dec hex
floatfield	fixed scientific

Wenn in Ihrem System gelegentlich die Ausgabe nicht zum erwarteten Zeitpunkt geschieht, liegt es wahrscheinlich daran, dass `unitbuf` nicht voreingestellt ist. Manchmal ist es sinnvoll, den Benutzer eines Programms auf eine Wartezeit vorzubereiten:

```
// unitbuf zu Demonstrationszwecken zurücksetzen
cout.unsetf(ios::unitbuf);
cout << "langwierige Berechnung, bitte warten..\n";
// ... hier folgt die Berechnung
cout << "Ende der Berechnung" << endl;
```


Falls `unitbuf` nicht gesetzt ist, wie im Beispiel erzwungen, schlummert die Aufforderung zum Warten im Ausgabepuffer und erscheint erst *nach dem Ende* der Berechnung. `endl` bewirkt das Anhängen von `\n` sowie ein anschließendes Leeren des Ausgabepuffers mit `cout.flush()`. Das gewünschte Verhalten einer nicht gepufferten Ausgabe wird mit `cout.setf(ios::unitbuf)` erreicht.

Weite von Fließkommazahlen

Die Weite von Fließkommazahlen wird mit der Funktion `precision()` gesteuert, die die Anzahl der Ziffern bei Fließkomma-Ausgabe festlegt, sofern `fixed` oder `scientific` nicht gesetzt sind. Andernfalls legt `precision` die Anzahl der Nachkommastellen fest. Die eingestellte Anzahl ist gültig bis zum nächsten `precision()`-Aufruf. Mit dieser Funktion kann auch die aktuell eingestellte Ziffernzahl festgestellt werden. Das Programm

```
int vorherigeAnzahlDerZiffern = cout.precision();
cout.precision(8);
cout << 1234.56789 << " ";
cout << 1234.56789 << " "; // precision bleibt erhalten
cout.precision(4);
cout << 1234.56789 << endl;
cout.precision(vorherigeAnzahlDerZiffern); // Wiederherstellung
```

erzeugt die Ausgaben (mit automatischer Rundung)

```
1234.5679 1234.5679 1235
```

Falls die Anzahl der Ziffern vor dem Komma den Wert von `precision` überschreitet, wird auf die wissenschaftliche Notation, also Ausgabe mit Exponent, umgeschaltet.

Anzahl der Nachkommastellen festlegen

Falls `fixed` oder `scientific` gesetzt ist, legt `precision()` die Anzahl der Nachkommastellen fest. Das Beispiel spricht für sich:

```
double f = 1234.123456789012345;
cout.setf(ios::scientific, ios::floatfield);
cout.precision(4);
cout << f << endl; // 1.2341e+03
cout.setf(ios::fixed, ios::floatfield);
cout.precision(8);
cout << f << endl; // 1234.12345679
```

10.2 Eingabe

Die Klasse `istream` enthält Operatoren für die in C++ eingebauten Grunddatentypen, die für eine Umwandlung der eingelesenen Zeichen in den richtigen Datentyp sorgen:

```
istream& operator>>(char*); // C-Strings
istream& operator>>(char&);
```

```
istream& operator>>(float&);
istream& operator>>(int&);
//... usw.
```

Der Operator `>>` ist als überladener Operator definiert, wie wir ihn im Prinzip schon kennengelernt haben. Für einen Datentyp `T` (`T` steht für einen der Grunddatentypen) sei hier das Schema der Definition gezeigt:

```
istream& istream::operator>>(T& var) {
    // überspringe Zwischenraumzeichen
    // lies die Variable var des Typs T aus dem istream ein
    return *this;
}
```

Wie bei der Ausgabe bewirkt die Rückgabe einer Referenz auf `istream`, dass mehrere Eingaben verkettet werden können:

```
cin >> a >> b;           // ist gleichbedeutend mit
(cin >> a) >> b;         // und wird interpretiert als
(cin.operator>>(a)).operator>>(b);
```

Falls Zeichen oder Bytes eingelesen und Zwischenraumzeichen *nicht* ignoriert werden sollen, kann die Elementfunktion `istream& get()` genommen werden, die in mehreren überladenen Versionen bereitsteht, von denen eine ein einzelnes Zeichen einliest, wie das Beispiel zeigt. Zur Auswertung der `while`-Bedingung siehe Abschnitt 10.5.

```
// zeichenweises Kopieren der Standardeingabe (vgl. Beispiel Seite 97)
char c;
while(cin.get(c)) {
    cout << c;
}
// ebenfalls möglich ist:
while(cin.get(c)) {
    cout.put(c);
}
```

Zum sicheren Einlesen einer Zeichenkette in einen Pufferbereich wird einer anderen überladenen Version von `get()` der Zeiger auf den Pufferbereich und die Puffergröße mitgegeben, die wegen des abschließenden `'\0'`-Zeichens um eins größer als die Anzahl der maximal einzulesenden Elemente sein muss. Die Deklaration im Header `<iostream>` lautet `istream& get( char*, unsigned int, char t='\n')`. Die Zeichenkette wird bis zu einem festzulegenden Zeichen `t` übernommen (`t` steht für »Terminator«), wobei das Terminatorzeichen im Eingabestrom verbleibt. Es ist mit der Zeilenendekennung vorbesetzt. Beispiel:

```
// Zeile einlesen (max. N-1 Zeichen):
const unsigned int N = 100;
char buf[N];
cin >> buf;           // unsicher und Abbruch beim ersten Zwischenraumzeichen
cin.get(buf, N);      // sicher wegen Längenbegrenzung auf N-1 Zeichen
// ... hier ggf. Terminatorzeichen lesen
```

Wegen der Vorbesetzung des Terminatorzeichens ist der Aufruf gleichbedeutend mit `cin.get(buf, N, '\n')`. Als letztes Zeichen trägt `get()` in `buf[]` die Stringendekennung

'\0' ein. Darüber hinaus gibt es einige weitere `istream`-Funktionen, von denen ein kleiner Teil hier aufgeführt ist:

```
istream& getline(char*, unsigned int, char t='\n');
```

Diese Funktion wirkt wie `get()`, aber das Terminatorzeichen wird gelesen (jedoch nicht mit in den Puffer übernommen).

```
istream& ignore(size_t n = 1, int t = EOF);
```

Diese Funktion liest und verwirft alle Zeichen des Eingabestroms, bis entweder das Terminatorzeichen oder `n` andere Zeichen gelesen worden sind. Das Terminatorzeichen verbleibt nicht im `istream`. `EOF` ist ein vordefiniertes Makro, das das Dateiende (englisch *end of file*) kennzeichnet und das häufig durch den Wert `-1` repräsentiert wird. Ein Aufruf könnte zum Beispiel `cin.ignore(max_Zeilenlaenge, '\n');` lauten.

```
istream& putback(char c);
```

Diese Funktion gibt ein Zeichen `c` an den `istream` zurück. Die Funktion

```
int get();
```

holt das nächste Zeichen und gibt es als `int`-Wert entsprechend seiner Position in der ASCII-Tabelle zurück. Bei `EOF` wird `-1` zurückgegeben. Man könnte die Standardeingabe daher auf folgende Art kopieren:

```
int i;
while(cin.good() && (i = cin.get()) != EOF) { // d.h. weder EOF noch Lesefehler
    cout.put(static_cast<char>(i));
}
```

Eine andere Möglichkeit wäre die Abfrage mit `eof()`, eine Funktion, die `true` zurückgibt, wenn ein Leseversuch wegen Erreichens des Dateiendes erfolglos bleibt:

```
char c;
while(!cin.eof()) { // besser: while(cin.good()) ...
    cin.get(c);
    if(!cin.fail()) { // EOF ist möglich, deswegen nicht if(cin.good())
        cout.put(c);
    }
}
```

`cin.good()` ist vorzuziehen, weil nicht nur `EOF`, sondern auch Lesefehler berücksichtigt werden. Eine Vorschau auf das nächste Zeichen im Eingabestrom wird durch die Funktion

```
int peek();
```

erlaubt. Die Wirkung von `c = cin.peek()` ist wie die Hintereinanderschaltung der Anweisungen (mit impliziter Typumwandlung)

```
c = cin.get(); cin.putback(c);
```

10.3 Manipulatoren

Manipulatoren sind Operationen, die direkt in die Ausgabe oder Eingabe zur Erledigung bestimmter Funktionen, zum Beispiel zur Formatierung, eingefügt werden. In diesem Abschnitt werden Manipulatoren nur für die Ausgabe beschrieben, das Prinzip lässt sich jedoch gleichermaßen auch für die Eingabe anwenden. Um eine `int`-Zahl oktalauszugeben, schreibt man

```
cout << oct << zahl << endl;
```

Die Wirkung ist genauso, als ob ein Flag zur Formatänderung gesetzt worden wäre:

```
cout.setf(ios::oct, ios::basefield);
cout << zahl << endl;
```

Sowohl `oct` als auch `endl` sind Manipulatoren. `endl` haben wir bereits kennengelernt. Wie funktioniert so ein Manipulator? Es gibt spezielle überladene Formen des Ausgabeoperators und verschiedene Funktionen, zum Beispiel `oct()` und `endl()`:

```
// Funktionen zur Klasse ostream
ostream& operator<<(ostream& (*fp)(ostream&));
ostream& operator<<(ios& (*fp)(ios&));
// .....
ostream& endl(ostream&);
// Funktion zur Klasse ios
ios& oct(ios&);
```

Der überladene Operator erwartet ein Argument vom Typ »Zeiger auf eine Funktion, die eine Referenz auf einen `ostream` als Parameter hat und eine Referenz auf einen `ostream` als Ergebnis zurückliefert«. Die Funktion `endl()` erfüllt genau dieses Kriterium, und die Funktion `oct()` entsprechend für die Klasse `ios`. Es werden in der einfachen Schreibweise `cout << oct << zahl << endl;` Funktionsnamen, also Zeiger auf eine Funktion, übergeben. Der Operator führt diese Funktionen aus, sodass die Einstellung auf die oktale Zahlenbasis beziehungsweise die Ausgabe einer neuen Zeile bewirkt werden. Die uns nicht vorliegende Implementierung zum Beispiel von `endl()` könnte wie folgt aussehen:

```
ostream& endl(ostream& os) {
    os.put('\n');
    os.flush();
    return os;
}
```

Der Ausgabeoperator ruft die übergebene Funktion auf:

```
ostream& ostream::operator<<(ostream& (*fp)(ostream&)) {
    *fp(*this);           // Funktionsaufruf
    return *this;
}
```

Die Anweisung

```
cout << zahl << endl;
```

wird ausgewertet zu

```
(cout.operator<<(zahl)).operator<<(endl);
```

In Kenntnis dieses Mechanismus können Sie nun selbst die tollsten Manipulatoren schreiben! Aber ehe Sie loslegen, schauen Sie sich erst die bereits vorhandenen an (Tabellen 10.3 bis 10.5), von denen einige in den Headern `<ios>` und `<iostream>` deklariert sind, andere (die mit Argumenten) jedoch in `<iomanip>` (*iomanip* = *input output manipulator*). Einschließen von `<iostream>` impliziert Inkludieren von `<ios>`.

Tabelle 10.3: ios-Manipulatoren

Name	Bedeutung
boolalpha	true/false alphabetisch ausgeben oder lesen
noboolalpha	true/false numerisch (1/0) ausgeben oder lesen
showbase	Basis anzeigen
noshowbase	keine Basis anzeigen
showpoint	nachfolgende Nullen ausgeben
noshowpoint	keine nachfolgenden Nullen ausgeben
showpos	+ bei positiven Zahlen anzeigen
noshowpos	kein + bei positiven Zahlen anzeigen
skipws	Zwischenraumzeichen ignorieren
noskipws	Zwischenraumzeichen berücksichtigen
uppercase	E,X statt e,x
lowercase	e,x statt E,X
unitbuf	Puffer nach jeder Ausgabe leeren
nounitbuf	Ausgabe puffern
adjustfield:	
internal	zwischen Vorzeichen und Wert auffüllen
left	linksbündige Ausgabe
right	rechtsbündige Ausgabe
basefield:	
dec	dezimal
oct	oktal
hex	hexadezimal
floatfield:	
fixed	Gleitkomma-Format
scientific	Exponential-Format

Tabelle 10.4: iostream-Manipulatoren

Name	Bedeutung	Typ
endl	neue Zeile ausgeben	ostream&
ends	Nullzeichen ('\0') ausgeben	ostream&
flush	Puffer leeren	ostream&
ws	Zwischenraumzeichen aus der Eingabe entfernen	istream&

Manipulatoren sind sehr einfach anzuwenden – auch wenn die Erklärung ihrer Wirkungsweise vielleicht nicht so einfach wie die Anwendung ist. Die Beispiele mit `cout.precision()` und `cout.width()` können umgeschrieben werden:

Tabelle 10.5: iomanip-Manipulatoren

Name	Bedeutung
<code>resetiosflags(ios::fmtflags M)</code>	Flags entsprechend der Bitmaske M zurücksetzen
<code>setiosflags(ios::fmtflags M)</code>	Flags entsprechend M setzen
<code>setbase(int B)</code>	Basis 8, 10 oder 16 definieren
<code>setfill(char c)</code>	Füllzeichen festlegen
<code>setprecision(int n)</code>	Fließkommaformat (siehe Seite 380)
<code>setw(int w)</code>	Weite setzen (entspricht <code>width()</code>)

```
cout << setw(6) << setfill('0') << 999 << ' ';
cout << setprecision(8) << 1234.56789 << endl;
```

Das Ergebnis ist: `000999 1234.5679` (Annahme: `fixed` und `scientific` sind nicht gesetzt). Ist Ihnen etwas aufgefallen? Wie die Syntax zeigt, ist ein Manipulator *mit* Argument(en) kein Zeiger auf eine Funktion, sondern ein Funktionsaufruf. Die Autoren der *iostream*-Bibliothek konnten den Operator `<<` nicht für alle nur denkbaren Fälle von Funktionen mit Parametern überladen. Daher wurde folgender Ausweg gewählt: Die aufgerufene Funktion muss ein Objekt einer (compilerspezifischen) Klasse, zum Beispiel `omanip` genannt, zurückgeben, das vom Ausgabeoperator verarbeitet werden kann. Im Header `<iomanip>` ist die Klasse `omanip` und ein `friend`-Operator `<<` etwa der folgenden oder einer ähnlichen Art zu finden:

```
template<typename T>
class omanip {
    ostream& (*funktPtr)(ostream&, T);
    T arg;
public:
    omanip(ostream& (*f)(ostream&, T), T obj)
        : funktPtr(f),
          arg(obj) {
    }
    friend ostream& operator<<(ostream&, omanip<T>&);
};

template<typename T>
ostream& operator<<(ostream& s, const omanip<T>& fobj) {
    return(*fobj.funktPtr)(s, fobj.arg);
}
```

Ein `omanip`-Objekt hat zwei private Variable: `funktPtr` ist ein Zeiger auf eine Funktion, die eine Referenz auf einen `ostream` zurückgibt und einen Parameter des Typs `ostream&` sowie ein Objekt des Typs `T` erwartet. In der Regel wird das Objekt vom Typ `int` oder `char` sein. Die zweite Variable `arg` enthält das Objekt vom Typ `T`.

Der Konstruktor initialisiert beide Variablen, die ihm als Parameter übergeben werden. Ferner finden Sie in `<iomanip>` eine Funktion (zum Beispiel) `setprecision()`. Die Funktion `setprecision()` gibt ein Objekt vom Typ `omanip` zurück:

```
omanip<int> setprecision(int p) {
    return omanip<int>(precision, p);
}
```

<int> rührt daher, dass die Klasse `omanip` als Template deklariert ist, um Manipulatorfunktionen mit verschiedenen Parametertypen zu erlauben. Dem Konstruktor eines `omanip`-Objekts wird die Adresse einer Funktion (`precision`) und der Wert `p` übergeben. Bei der Konstruktion dieses Objekts anlässlich der `return`-Anweisung werden diese Daten als Elementdaten abgelegt. Der Operator `<<` ruft die im `omanip`-Objekt referenzierte Funktion auf, die wiederum die `ostream`-Funktion `precision()` aufruft, die uns ja schon bekannt ist (Seite 380). Solcherart Objekte werden auch *Funktionsobjekte* genannt, eine Variante der in Abschnitt 9.6 beschriebenen Funktoren. Grund: Über den Weg der Erzeugung eines Objekts wird eine Funktion aufgerufen, wenn auch nicht mit `operator()()`, sondern innerhalb `operator<<()` über einen Funktionszeiger.

Entsprechend zu der Klasse `omanip` für die Ausgabe gibt es die Klasse `imani`p für die Eingabe und die Klasse `smani`p für `ios`-Funktionen. Diese Namen werden jedoch nicht vom C++-Standard vorgeschrieben.

10.3.1 Eigene Manipulatoren

Eigene Manipulatoren ohne Parameter

Dies ist der einfachste Fall. Es muss nur eine Funktion mit der passenden Schnittstelle geschrieben werden. Der Manipulator `endl` von Seite 383 ist ein gutes Beispiel dafür.

Eigene Manipulatoren mit Parametern

Es gibt zwei Wege, eigene Manipulatoren zu schreiben. Der eine Weg führt über den beschriebenen Ansatz: Funktionen, die ein `omanip`-Objekt zurückgeben und sich auf den dazugehörigen Ausgabeoperator verlassen. Dieser Weg hat den gravierenden Nachteil, dass man sich auf nicht standardisierte Klassennamen verlassen muss. Deswegen wird hier nur der zweite Weg, die Realisierung mit einem Funktor, vorgeschlagen. Die Header-Datei enthält die Definition des Manipulators:

Listing 10.1: Manipulator-Klasse Leerzeilen

```
// cppbuch/k10/manipula.h
#ifndef MANIPULA_H
#define MANIPULA_H
#include<iostream>

class Leerzeilen {
public:
    Leerzeilen(int i = 1) : anzahl(i) {}
    std::ostream& operator()(std::ostream& os) const {
        for(int i = 0; i < anzahl; ++i) {
            os << '\n';
        }
        os.flush();
        return os;
    }

private:
    int anzahl;
};
```

```
inline std::ostream& operator<<(std::ostream& os,
                               const Leerzeilen& leerz) {
    return leerz(os);    // Funktoraufruf
}
#endif // MANIPULA_H
```

Ein Funktor ist ein Objekt, das wie eine Funktion behandelt werden kann, wie auf Seite 344 gezeigt. Das Objekt kann beliebige Daten mit sich tragen. Dazu benötigt man nur noch einen mit der Klasse des Funktors überladenen Ausgabeoperator. Dies soll an dem einfachen Beispiel eines Manipulators `Leerzeilen(int z)`, der `z` Leerzeilen ausgibt, gezeigt werden. Eine mögliche Anwendung:

Listing 10.2: Anwendung des Manipulators

```
// cppbuch/k10/manipula.cpp
#include "manipula.h"
int main() {
    std::cout << Leerzeilen(25)    // Konstruktoraufruf!
               << "Ende" << std::endl;
}
```

würde den Bildschirm durch Ausgabe von 25 Leerzeilen löschen und dann den Text *Ende* ausgeben. Der Ablauf im `main()`-Program umfasst mehrere Schritte:

1. Zunächst wird ein Objekt vom Typ `Leerzeilen` konstruiert, das mit 25 als Parameter initialisiert wird.
2. Der Ausdruck `cout << Leerzeilen(25)` wird vom Compiler in die Langform `operator<<(cout, Leerzeilen(25))` umgewandelt. Daran ist zu sehen, dass das erzeugte Objekt als Parameter an den überladenen Operator weitergereicht wird.
3. Innerhalb des Ausgabeoperators wird der Funktionsoperator der Klasse `Leerzeilen` aufgerufen. Die Schreibweise `leerz(os)` wird vom Compiler zu `leerz.operator()(os)` umgewandelt. Damit ist der Ausgabestrom (hier `cout`) innerhalb der Operatorfunktion bekannt, und es kann die gewünschte Zahl von Leerzeilen ausgegeben werden.
4. Durch das per Referenz zurückgegebene `ostream`-Objekt ist eine Verkettung mit weiteren Operatoren denkbar.

Der Vorteil des Einsatzes von Funktoren liegt in der Einfachheit und darin, dass bei entsprechender Gestaltung eine beliebige Zahl von Parametern möglich ist.

10.4 Fehlerbehandlung

Bei der Ein- und Ausgabe können natürlich Fehler auftreten. Zur Erkennung und Behandlung von Fehlern stehen verschiedene Funktionen zur Verfügung. Ein Fehler wird durch das Setzen eines Status-Bits markiert, das durch den Aufzählungstyp `iosstate` der Klasse `ios_base` (und damit auch `ios`) definiert wird. Die tatsächlichen Bitwerte sind implementationsabhängig.


```
enum iostate {
    goodbit  = 0x00,    // alles ok
    eofbit   = 0x01,    // Ende des Streams
    failbit  = 0x02,    // letzte Ein-/Ausgabe war fehlerhaft
    badbit   = 0x04     // ungültige Operation, grober Fehler
};
```

Der Stream ist nicht mehr benutzbar, falls `badbit` gesetzt ist. Tabelle 10.6 zeigt die Elementfunktionen, die auf die Statusbits zugreifen.

Tabelle 10.6: Abfrage des Ein-/Ausgabestatus

Funktion	Ergebnis
<code>ios::iostate rdstate()</code>	aktueller Status
<code>bool good()</code>	wahr, falls »gut«, d.h. <code>rdstate() == 0</code>
<code>bool eof()</code>	wahr, falls Dateiende
<code>bool fail()</code>	wahr, falls <code>failbit</code> oder <code>badbit</code> gesetzt
<code>bool bad()</code>	wahr, falls <code>badbit</code> gesetzt
<code>void clear()</code>	Status auf <code>goodbit</code> setzen
<code>void clear(iostate s)</code>	Status auf <code>s</code> setzen
<code>void setstate(iostate)</code>	einzelne Statusbits setzen

Das folgende Demonstrationsprogramm zeigt die Anwendung der Funktionen zur Diagnose und Behebung von Syntaxfehlern beim Einlesen von `int`-Zahlen. Es wird dabei angenommen, dass beliebig oft `int`-Zahlen `i` eingelesen und angezeigt werden sollen, wobei vorher auftretende falsche Zeichen zu ignorieren sind.

Listing 10.3: `istream`-Status abfragen

```
// cppbuch/k10/iosstate.cpp
#include<iostream>

using namespace std;

int main() {
    int i;
    ios::iostate status;

    while(true) {           // Schleifenabbruch mit break
        cout << "Zahl (Strg+D oder Strg+Z = Ende):";
        cin >> i;
        status = cin.rdstate();
        // Ausgabe der Statusbits
        cout << "status = " << status << endl;
        cout << "good() = " << cin.good() << endl;
        cout << "eof() = " << cin.eof() << endl;
        cout << "fail() = " << cin.fail() << endl;
        cout << "bad() = " << cin.bad() << endl;
        if(cin.eof())
            break;           // Abbruch
        // Fehlerbehandlung bzw. Ausgabe
        if(status) {
            cin.clear();      // Fehlerbits zurücksetzen
            cin.get();        // ggf. fehlerhaftes Zeichen entfernen
        }
    }
}
```

```

    }
    else cout << "*** " << i << endl;
}
}

```

Die Funktion `void clear(iostate statuswort = goodbit)` erlaubt es, den Status zu setzen. Beispielsweise kann das `badbit` bei Erhaltung der anderen Bits mit `cin.clear(ios::badbit | cin.rdstate())` gesetzt werden. Durch Eingabe von Buchstaben ist die Syntax von `int-Zahlen` nicht erfüllt, wie mit `fail()` angezeigt wird. Die Tastenkombination `[Strg]+Z` oder `[Strg]+D` liefert einen Wert ungleich 0 für `eof()` und führt zum Verlassen der Schleife.

Exception `ios::failure`

Wenn seitens des Ein-/Ausgabesystems ein Fehler gefunden und deswegen intern die Funktion `setstate(failbit)` aufgerufen wird, *kann* es sein, dass sie eine `ios_base::failure`-Exception wirft. Der Standard legt sich nicht fest, weil diese Exception neu aufgenommen wurde und das Verhalten bisher gültiger Programme sich nicht ändern soll – die Entscheidung liegt beim Hersteller des Compilers. Da die Klasse `ios` von `ios_base` erbt, kann sie kürzer `ios::failure` genannt werden. `ios::failure` erbt von `system_error`. Man kann sie bei eigenen Programmen selbst werfen und auswerten. Das Programm auf Seite 636 zeigt eine mögliche Anwendung.

10.5 Typumwandlung von Dateiobjekten nach bool

Die Abfrage, ob eine Datei geöffnet werden kann, wird über eine `if`-Abfrage etwa der folgenden Art gelöst:

```

ifstream quellfile;
quellfile.open("text.dat");
if(!quellfile) {
    cerr << "Datei kann nicht geöffnet werden!";
    exit(-1);
}

```

Ob das Ende einer Datei erreicht worden ist, kann in einer Bedingung wie folgt festgestellt werden:

```

while(quellfile.get(c)) {
    zielfile.put(c);
}

```

Wie funktioniert dieser geheimnisvolle Mechanismus? Erinnern wir uns daran, dass der Compiler automatisch versucht, einen nicht ganz passenden Datentyp in einen passenden umzuwandeln, wenn es nötig sein sollte; der Versuch gelingt nicht immer. Dazu kann

er Typumwandlungskonstruktoren (siehe Seite 160) und eingebaute oder selbst definierte Typumwandlungsoperatoren (siehe Seite 337) benutzen. Zur Klasse `ios` gibt es einen vordefinierten Typumwandlungsoperator, der ein Dateiobjekt oder eine Referenz darauf in einen Wert vom Typ `void*` umwandelt. Zusätzlich wird der Negationsoperator überladen. Diese Operatoren werden vom Compiler benutzt, um die Bedingung auswerten zu können. Sie benutzen die Funktion `fail()` zum Lesen des Status und sind etwa wie folgt implementiert:

```
// Konversion eines Objekts in einen void-Zeiger
ios::operator void*() const {
    if(fail()) return static_cast<void*>(0);
    else      return static_cast<void*>(this);
}

bool ios::operator!() const { // überladener Negationsoperator
    return fail();
}
```

Die Anweisung `while(cin) cin.get(c);` wird interpretiert als

```
while(cin.operator void*())
    cin.get(c);
```

und die Anweisung `if(!cin.get(c)) {...}` entspricht

```
if((cin.get(c)).operator!()) {
    //...
}
```

10.6 Arbeit mit Dateien

Die Arbeit mit Dateien ist teilweise aus Kapitel 2 bekannt. In diesem Abschnitt finden sich einige Ergänzungen. Zum Öffnen einer Datei wird ein bestimmter Modus angegeben (englisch *openmode*) wie zum Beispiel `ios::binary`. Die Tabelle 10.7 zeigt die möglichen Werte.

Tabelle 10.7: `ios_base`-Öffnungsarten für Ströme

Modus	Bedeutung
app	beim Schreiben Daten an die Datei anhängen
ate	nach dem Öffnen an das Dateiende springen
binary	keine Umwandlung verschiedener Zeilenendekennungen
in	zur Eingabe öffnen
out	zur Ausgabe öffnen
trunc	vorherigen Inhalt der Datei löschen

Bestimmte Werte sind Stream-abhängig voreingestellt. Beispiel: Bei einem `ifstream`-Objekt ist `ios::in` voreingestellt. Die Angabe eines Modus überschreibt den vorgegebenen

Modus! Deswegen sollte `ios::binary` bei `ifstream`-Objekten mit `ios::in` gekoppelt werden, d.h. `Modus = ios::binary | ios::in` (`ofstream` entsprechend mit `ios::out`). Die verschiedenen Öffnungsarten können durch Verknüpfung mit dem Oder-Operator kombiniert werden. Tabelle 10.8 zeigt die sinnvollen möglichen Kombinationen. Ein Beispiel:

```
// nur am Ende schreiben
ofstream Ausgabestrom("Ausgabe.dat", ios::out | ios::app);
// Lesen und Schreiben (Beispiel folgt auf Seite 392):
fstream EinAusgabestrom("Datei.txt", ios::in | ios::out);
```

Tabelle 10.8: Kombinationen der Dateiöffnungsarten

binary	in	out	trunc	app / ate
	•			
	•	•		
	•	•	•	
		•	•	
		•		•
•	•	•		
•	•	•	•	
•		•	•	
•		•		•

10.6.1 Positionierung in Dateien

Manchmal ist es wünschenswert, eine Datei nicht nur sequenziell zu lesen oder zu schreiben, sondern die Position frei zu bestimmen. Dazu gehört auch, die aktuelle Position zu ermitteln. Die Tabelle 10.9 zeigt die vorhandenen Funktionen. Die Endung »g« steht für »get« (= zu lesende Datei) und »p« steht für »put« (= zu schreibende Datei). Die Bezugsposition in Tabelle 10.9 kann einen von drei möglichen Werten annehmen:

`ios::beg` relativ zum Dateianfang
`ios::cur` relativ zur aktuellen (englisch *current*) Position
`ios::end` relativ zum Dateiende

Das folgende Programmfragment zeigt eine Anwendung:

Tabelle 10.9: Ermitteln und Suchen von Dateipositionen

Rückgabetyt	Funktion	Bedeutung
<code>ios::pos_type</code>	<code>tellg()</code>	aktuelle Leseposition
<code>ios::pos_type</code>	<code>tellp()</code>	aktuelle Schreibposition
<code>istream&</code>	<code>seekg(p)</code>	absolute Position p aufsuchen
<code>istream&</code>	<code>seekg(r, Bezug)</code>	relative Position r aufsuchen (zur Bezugsposition siehe Text)
<code>ostream&</code>	<code>seekp(p)</code>	absolute Position p aufsuchen
<code>ostream&</code>	<code>seekp(r, Bezug)</code>	relative Position r aufsuchen

```
// Auszug aus cppbuch/k10/seektell.cpp
ifstream einIfstream;
```

```

einIfstream.open("seek.dat", ios::binary|ios::in);
einIfstream.seekg(9); // absolute Leseposition 9 suchen (Zählung ab 0)
char c;
einIfstream.get(c); // an Pos. 9 lesen, get schaltet Position um 1 weiter
einIfstream.seekg(2, ios::cur); // 2 Positionen weitergehen
ios::pos_type position = einIfstream.tellg(); // akt. Position merken
// ...
einIfstream.seekg(-4, ios::end); // 4 Positionen vor dem Ende
// ...
einIfstream.seekg(position, ios::beg); // zur gemerkten Position gehen

```

Daraus ergibt sich, dass die relative Positionierung zum Dateianfang redundant ist, d. h. `seekg(x, ios::beg)` ist dasselbe wie `seekg(x)`;

10.6.2 Lesen und Schreiben in derselben Datei

Wenn eine Datei als Datenbasis genutzt wird, ist es interessant, Daten zu lesen und geänderte Daten in *derselben* Datei zu aktualisieren. Für diesen Zweck gibt es die Klasse `fstream`, die von der Klasse `iostream` erbt, die wiederum von den Klassen `istream` und `ostream` abgeleitet ist. Damit hat `fstream` die Eigenschaft, sowohl für die Ein- als auch für die Ausgabe geeignet zu sein. Alle Klassen, die mit Dateien arbeiten, benutzen Puffer. In `fstream` wird derselbe Pufferspeicher zum Lesen und zum Schreiben benutzt. Ein einfaches Programm zeigt die Nutzung eines `fstreams`, bei dem die Funktionen zum Aufsuchen von Positionen naturgemäß eine starke Rolle spielen.

Listing 10.4: Lesen/Schreiben derselben Datei

```

// cppbuch/k10/fstream2.cpp
#include<fstream>
#include<iostream>
using namespace std;

int main() { // Lesen und Schreiben derselben Datei
    // Datei anlegen
    fstream filestream("fstream2.dat", ios::out | ios::trunc);
    filestream.close(); // leere Datei existiert jetzt
    int i; // Hilfsvariable
    // Datei zum Lesen und Schreiben öffnen:
    filestream.open("fstream2.dat", ios::in | ios::out);

    // schreiben:
    for(i = 0; i < 20; ++i) {
        filestream << i << ' '; // kein EOF direkt nach der letzten Zahl
    }
    filestream << endl;

    // lesen:
    filestream.seekg(0); // Anfang suchen
    while(filestream.good()) {
        filestream >> i; // lesen
        if(filestream.good()) {
            cout << i << ' '; // Kontrollausgabe
        }
    }
}

```

```

    else {
        cout << endl << "Dateiende erreicht (oder Lesefehler)";
    }
}
cout << endl; // Ergebnis: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
filestream.clear(); // EOF-Status löschen

// Inhalt teilweise überschreiben
filestream.seekp(5); // Position 5 zum Schreiben suchen
filestream << "neuer Text "; // ab Pos. 5 überschreiben
filestream.seekg(0); // Anfang zum Lesen suchen
char buf[100];
filestream.getLine(buf, 100); // Zeile lesen
cout << buf << endl; // Kontrollausgabe. Ergebnis:
// 0 1 2 neuer Text 9 10 11 12 13 14 15 16 17 18 19
}

```

10.7 Umleitung auf Strings

Die Ausgabe kann mithilfe von `ostream`-Objekten (*output string stream*) auf Strings umgeleitet werden. Dies ist dann sinnvoll, wenn die Ausgabe nicht unmittelbar erfolgen soll. Ähnliches gilt auch für das Lesen aus Strings anstelle einer Datei (`istream`). `stringstream`-Klassen sind im Header `<sstream>` deklariert. Hier sei beispielhaft nur die Ausgabe behandelt. Es ist eine Funktion `zahlToString()` angegeben, die eine Zahl in einen formatierten String umwandelt. Der String kann anschließend anderweitig ausgewertet oder ausgegeben werden.

Die Funktion ist für verschiedene Typen von Zahlen geeignet. Die Parameter zur Formatierung können weggelassen werden, wenn das vorgegebenen Format ausreicht. Ein Beispiel zeigt Anwendung und Funktionsweise.

Listing 10.5: Stream-Ausgabe in String umleiten

```

// cppbuch/k10/strstrea.cpp
#include<iostream>
#include"zahlToString.h"
using namespace std;

int main() { // Anwendung
    double xd = 73.1635435363; // Ausgabe:
    cout << zahlToString(xd) << endl; //73.1635
    cout << zahlToString(xd, 12) << endl; // 73.1635
    cout << zahlToString(xd, 12, 1) << endl; // 7.3164e+01
    cout << zahlToString(xd, 12, 1, 3) << endl; // 7.316e+01
    cout << zahlToString(xd, 12, 0, 3) << endl; // 73.164
    int xi = 1234567;
    cout << zahlToString(xi) << endl; //1234567
    cout << zahlToString(xi, 14) << endl; // 1234567
}

```

```

float xf = 1234.567;
cout << zahlToString(xf) << endl;    //1234.57
unsigned long xl = 123456789L;
cout << zahlToString(xl) << endl;    //123456789
}

```

Listing 10.6: Prototypen

```

// cppbuch/k10/zahlToString.h
#ifndef ZAHLTOSTRING_H
#define ZAHLTOSTRING_H
#include<string>

std::string zahlToString(double d, unsigned int weite = 0,
    unsigned int format = 2, // 0: fix, 1: scientific, sonst: automatisch
    unsigned int anzahlNachkommastellen = 4); // nur format 0/1
// ganze Zahlen:
std::string zahlToString(long i,    unsigned int weite = 0);
std::string zahlToString(int i,    unsigned int weite = 0);
std::string zahlToString(unsigned long i, unsigned int weite = 0);
std::string zahlToString(unsigned int i, unsigned int weite = 0);
#endif

```

Listing 10.7: Implementierung

```

// cppbuch/k10/zahlToString.cpp
#include<sstream>
#include<iostream>
#include"zahlToString.h"

// Das ostream-Objekt wandler in der folgenden Funktion
// zahlToString(double, unsigned int, unsigned int, unsigned int)
// stellt dynamisch Platz bereit, ohne dass man sich besonders darum kümmern muss. Der
// Aufruf wandler.str(); gibt alle in den stringstream geschriebenen Ausgaben als String zurück.
std::string zahlToString(double d, unsigned int weite,
    unsigned int format, // 0: fix, 1: scientific,
    // sonst: automatisch
    unsigned int anzahlNachkommastellen ) { // nur Format 0/1
    std::ostringstream wandler;
    if(format == 0) {
        wandler.setf(std::ios::fixed, std::ios::floatfield);
        if(anzahlNachkommastellen > 0) {
            wandler.setf(std::ios::showpoint);
        }
    }
    else {
        if(format == 1) {
            wandler.setf(std::ios::scientific, std::ios::floatfield);
        }
    }
    if(format == 0 || format == 1) {
        wandler.precision(anzahlNachkommastellen);
    }
}

```

```

    if(weite > 0) {
        wandler.width(weite);
    }
    // Zahl und abschließendes Nullzeichen in den Strom einfügen
    wandler << d << std::ends;
    return wandler.str();
}

std::string zahlToString(long i, unsigned int weite) {
    return zahlToString(static_cast<double>(i), weite, 0, 0);
}
std::string zahlToString(int i, unsigned int weite) {
    return zahlToString(static_cast<double>(i), weite, 0, 0);
}
std::string zahlToString(unsigned long i, unsigned int weite) {
    return zahlToString(static_cast<double>(i), weite, 0, 0);
}
std::string zahlToString(unsigned int i, unsigned int weite) {
    return zahlToString(static_cast<double>(i), weite, 0, 0);
}

```

Mit einem `istringstream`-Objekt kann aus einem String gelesen werden, ein `stringstream`-Objekt erlaubt Lesen und Schreiben.

10.8 Ergänzungen

locale-Objekt ermitteln

Das für einen Stream gültige `locale`-Objekt (`locale` siehe Seite 821 ff.) für die Sprachumgebung kann mit `getloc()` ermittelt werden, z. B. `cout.getloc();`.

Streams verbinden mit `tie()`

Die Funktion `tie()` der Klasse `basic_ios` verbindet und löst Verbindungen zwischen einem `istream` und einem `ostream`. Wenn das Programmfragment

```

cout << "Zahl eingeben: ";
cin >> Zahl;

```

gegeben ist, dann ist nicht sichergestellt, dass die Bildschirmausgabe *vor* der Eingabeoperation erscheint, weil der Ausgabepuffer noch nicht voll ist. Auf Seite 379 wird das Problem mit dem Setzen von `ios::unitbuf` gelöst, eine andere Möglichkeit ist das Verbinden der Streams:

```

cin.tie(&cout); // Verbindung herstellen
cout << "Zahl eingeben: ";
cin >> Zahl;    // Diese Zeile verursacht jetzt cout.flush()
// Abfrage der Verbindung
if(cin.tie() == &cout) {

```



```
    cout << "Streams cin und cout sind verbunden" << endl;
}
cin.tie(0);      // Verbindung lösen
```

sentry

Es gibt sowohl die Klasse `basic_istream::sentry` als auch die Klasse `basic_ostream::sentry`. `sentry`-Objekte sorgen für eine sichere Umgebung bei Ein- bzw. Ausgabeoperationen. Der Konstruktor enthält Code, der vorher ausgeführt werden soll (Präfix-Code), zum Beispiel den Puffer eines mit `tie()` verbundenen Streams schreibend zu leeren (`flush()`). Der `bool`-Operator gibt an, ob der Stream in einem guten Zustand ist. Der Destruktor enthält Code, der nach der Operation ausgeführt werden soll (Suffix-Code), zum Beispiel das Werfen einer Exception, wenn der Stream in einen Fehlerzustand gelangt ist. Anwendungsbeispiele sind auf den Seiten [835](#) und [836](#) zu finden. Ein typisches Anwendungsmuster ist:

```
// Beispiel für istream::sentry
void f(istream& is) {
    istream::sentry s(is); // Präfix-Code
    if(s) {                 // ok? (bool sentry::operator())
        // irgendetwas mit is tun
    }
}                          // Suffix-Code (Destruktor)
```

Mit `sentry`-Objekten wird das C++-Prinzip »Resource Acquisition Is Initialization« (RAII, siehe Glossar) realisiert.

11

Einführung in die Standard Template Library (STL)

Dieses Kapitel behandelt die folgenden Themen:

- Entstehung und Konzept
- Bezug zur C++-Standardbibliothek
- Wie wirken Container, Iteratoren und Algorithmen zusammen?

Einer der Erfolge von C++ beruht darauf, dass zahlreiche Bibliotheken am Markt vorhanden sind, die die Entwicklung von Programmen erheblich erleichtern, weil sie verlässliche und erprobte Komponenten anbieten. Eine besonders sorgfältig konstruierte Bibliothek ist die *Standard Template Library*, die bei Hewlett-Packard von Alexander Stepanov, Meng Lee und ihren Kollegen entwickelt wurde. Sie ist wegen ihrer überzeugenden Konzeption des Zusammenwirkens von Behälterklassen (englisch *Container*) und Algorithmen vom ISO-Komitee als Teil des C++-Standards akzeptiert worden. Durch das Aufgehen in der C++-Standardbibliothek hat die STL die Rolle als eigenständige Bibliothek verloren.

Der große Vorteil von Templates liegt auf der Hand. Die Auswertung der Templates geschieht zur Compilierzeit, es gibt keine Laufzeiteinbußen – etwa durch polymorphe Funktionszugriffe, falls die Generizität mit Vererbung realisiert wird. Der Vorteil der Standardisierung ist noch größer einzuschätzen. Programme, die eine standardisierte Bibliothek benutzen, sind leichter portierbar, weil jeder Compiler-Hersteller sich am Standard orien-

tieren wird. Darüber hinaus sind sie leichter wartbar, weil das entsprechende Know-how sehr viel verbreiteter ist als das Wissen über eine spezielle Bibliothek.

Der Schwerpunkt liegt auf *Algorithmen*, die mit *Containern* und *Iteratoren* zusammenarbeiten. Durch den Template-Mechanismus von C++ sind die Container für Objekte verschiedenster Klassen geeignet. Ein Iterator ist ein Objekt, das wie ein Zeiger auf einem Container bewegt werden kann, um auf das eine oder andere Objekt zu verweisen. Algorithmen arbeiten mit Containern, indem sie auf zugehörige Iteratoren zugreifen. Diese Konzepte werden weiter unten detailliert dargestellt.

Reduktion der Bibliothekskomponenten durch Templates

Die C++-Standardbibliothek rückt nicht Vererbung und Polymorphismus in den Vordergrund, sondern Container und Algorithmen für viele, auch benutzerdefinierte Datentypen, sofern sie einigen wenigen Voraussetzungen genügen. Die C++-Templates bieten die Grundlage dafür. Der Schwerpunkt liegt also nicht auf der Objektorientierung, sondern auf der generischen Programmierung. Damit ist der große Vorteil verbunden, dass die Anzahl der notwendigen verschiedenen Container- und Algorithmentypen drastisch reduziert wird. Ein weiterer Vorteil ist die Typsicherheit, weil Templates bereits zur Compilationszeit aufgelöst werden.

Nehmen wir zunächst an, dass es keine Templates gäbe und dass wir ein Element eines Datentyps `int` in einem Container vom Typ `vector` finden wollen. Dazu brauchen wir einen Algorithmus `find()`, der den Container durchsucht. Falls wir n verschiedene Container (Liste, Menge ...) haben, brauchen wir für jeden Container einen eigenen Algorithmus, also n `find()`-Algorithmen. Nun könnte es ja sein, dass wir nicht nur ein `int`-Objekt, sondern ein Objekt eines beliebigen, von m möglichen Datentypen suchen wollen. Damit würde die Anzahl der `find()`-Algorithmen auf $n \cdot m$ steigen. Diese Betrachtung soll für k verschiedene Algorithmen gelten, sodass insgesamt $k \cdot n \cdot m$ Algorithmen zu schreiben sind. Die Benutzung von Templates erlaubt es, die Anzahl m auf 1 zu reduzieren. Algorithmen der C++-Standardbibliothek arbeiten jedoch nicht direkt mit Containern, sondern nur mit Schnittstellenobjekten, den Iteratoren, die auf Container zugreifen. Iteratoren sind zeigerähnliche Objekte, die unten genau erklärt werden. Dadurch reduziert sich die notwendige Gesamtzahl auf $n + k$ statt $n \cdot k$, eine erhebliche Ersparnis.



11.1 Container, Iteratoren, Algorithmen

Zunächst werden die wichtigsten Elemente skizziert, ehe auf ihr Zusammenwirken eingegangen wird.

Container

Die C++-Standardbibliothek stellt verschiedene Arten von Containern zur Verfügung, die als Template-Klassen formuliert sind. Sie werden detailliert in Kapitel 28 beschrieben. Container sind Objekte, die zur Verwaltung anderer Objekte dienen, wobei es den

Benutzern überlassen bleibt, ob sie die Objekte per Wert oder per Referenz ablegen. Die Ablage per Wert meint, dass jedes Element des Containers ein Objekt eines kopierbaren Typs ist (Wertsemantik). Die Ablage per Referenz heißt, dass die Elemente des Containers Zeiger auf Objekte von möglicherweise heterogenem Typ sind. In C++ müssen die verschiedenen Typen von einer Basisklasse abgeleitet und die Zeiger vom Typ »Zeiger auf Basisklasse« sein.

Ein Mittel, verschiedene Algorithmen mit verschiedenen Containern zusammenarbeiten zu lassen, besteht darin, dass die *Namen*, die zur Compilierzeit ausgewertet werden, für gleichartige Operationen gleich gewählt sind. Zum Beispiel gibt die Methode `size()` die Anzahl der Elemente eines Containers zurück, sei er nun vom Typ `vector`, `list` oder `map`. Ein anderes Beispiel sind die Methoden `begin()` und `end()`, mit denen die Position des ersten und die Position *nach* dem letzten Element ermittelt werden. Die letzte Position ist auch in einem C++-Array stets definiert, wenn auch nicht dereferenzierbar. Ein leerer Container wird durch Gleichheit von `begin()` und `end()` gekennzeichnet.

Iteratoren

Iteratoren arbeiten wie Zeiger. Je nach Anwendungsfall können sie selbst gewöhnliche Zeiger oder Objekte mit zeigerähnlichen Eigenschaften sein. Der Zugriff auf ein Container-Element ist über einen Iterator möglich. Iteratoren können sich von einem Element zum nächsten bewegen, wobei die Art der Bewegung nach außen hin verborgen ist (Kontrollabstraktion). Beispielsweise bedeutet in einem Vektor die Operation `++` das einfache Weiterschalten zur nächsten Position im Speicher, während dieselbe Operation in einem binären Suchbaum mit dem Entlangwandern im Baum verbunden ist.



Hinweis

Die folgende Konvention der C++-Standardbibliothek muss beachtet werden: Wenn zwei Iteratoren einen *Bereich* definieren, zeigt der erste Iterator auf die Position des ersten Elements und der zweite auf die Position *nach* dem letzten Element.

Algorithmen

Die Template-Algorithmen arbeiten mit Iteratoren, die auf Container zugreifen. Da nicht nur benutzerdefinierte Datentypen unterstützt werden, sondern auch die in C++ ohnehin vorhandenen Datentypen wie `int`, `char` usw., wurden die Algorithmen so entworfen, dass sie ebenso gut mit normalen Zeigern arbeiten (siehe Beispiel im folgenden Abschnitt).

Zusammenwirken

Container stellen Iteratoren zur Verfügung, Algorithmen benutzen sie:

Container $\Longleftrightarrow$ Iteratoren $\Longleftrightarrow$ Algorithmen

Dadurch gibt es eine Entkopplung, die ein außergewöhnlich klares Design erlaubt. Im Folgenden soll ein Programm in verschiedenen Varianten zeigen, dass Algorithmen mit C-Arrays genauso gut funktionieren wie mit Template-Klassen der C++-Standardbibliothek.

In diesem Beispiel soll ein per Dialog einzugebender `int`-Wert in einem Array gefunden werden, wozu eine Funktion `find()` benutzt wird, die auch als Algorithmus der C++-

Standardbibliothek vorliegt. Parallel wird `find()` auf verschiedene Arten formuliert, um die Abläufe sichtbar zu machen. Um sich schrittweise der angestrebten Formulierung zu nähern, wird zunächst eine Variante *ohne* Benutzung der C++-Standardbibliothek dargestellt. Der Container ist ein schlichtes C-Array. Um auszudrücken, dass ein Zeiger als Iterator wirkt, wird der Typname `IteratorType` mit `typedef` eingeführt.

Listing 11.1: Variante noch ohne Nutzung der STL

```
// cppbuch/k11/einf/ohneSTL.cpp
#include<iostream>
using namespace std;
// neuer Typname IteratorType für 'Zeiger auf const int' :
typedef const int* IteratorType; // hier noch Zeiger

// Prototyp des Algorithmus
IteratorType find(IteratorType begin, IteratorType end, const int& Value);

int main() {
    const int ANZAHL = 100;
    int einContainer[ANZAHL]; // Container definieren
    IteratorType begin = einContainer; // Zeiger auf den Anfang
    // Position NACH dem letzten Element
    IteratorType end = einContainer + ANZAHL;
    // Container mit beliebigen Werten füllen (hier: gerade Zahlen)
    for(int i = 0; i < ANZAHL; ++i) {
        einContainer[i] = 2*i;
    }
    int zahl = 0;
    while(zahl != -1) {
        cout << "gesuchte Zahl eingeben (-1 = Ende):";
        cin >> zahl;
        if(zahl != -1) { // weitermachen?
            IteratorType position = find(begin, end, zahl);
            if (position != end) {
                cout << "gefunden an Position "
                     << (position - begin) << endl;
            }
            else {
                cout << zahl << " nicht gefunden!" << endl;
            }
        }
    }
}

// Implementation
IteratorType find(IteratorType begin, IteratorType end, const int& Value) {
    while(begin != end // Zeigervergleich
           && *begin != Value) // Dereferenzierung und Objektvergleich
        ++begin; // nächste Stelle
    return begin;
}
```

Im Fall eines leeren Containers haben beide Iteratoren denselben Wert. Man sieht, dass der Algorithmus `find()` selbst nichts über den Container wissen muss. Er benutzt nur Zeiger (Iteratoren), die einige wenige Fähigkeiten benötigen:

- Der Operator `++` dient zum Weiterschalten auf die nächste Position.
- Der Operator `*` dient zur Dereferenzierung. Angewendet auf einen Zeiger (Iterator) gibt er eine Referenz auf das dahinterstehende Objekt zurück.
- Die Zeiger müssen mit dem Operator `!=` vergleichbar sein.

Die Objekte im Container werden hier mit dem Operator `!=` verglichen. Im nächsten Schritt streiche ich die Implementierung der Funktion `find()` und ersetze den Prototyp durch ein Template:

Listing 11.2: Variante 2: Algorithmus als Template

```
// Auszug aus cppbuch/k11/einf/ohneSTLmitTemplate.cpp
template<typename Iterator, typename T>
Iteratortype find(Iterator begin, Iterator end, const T& Value) {
    while(begin != end // Zeigervergleich
           && *begin != Value) // Dereferenzierung und Objektvergleich
        ++begin; // nächste Position
    return begin;
}
```

Der Rest des Programms bleibt *unverändert*. Der Platzhalter `Iteratortype` für den Datentyp des Iterators kann jeden beliebigen Namen haben. Im dritten Schritt benutze ich einen Container der STL. Die Iteratoren `begin` und `end` werden im `main()`-Programm durch die Methoden der Klasse `vector<T>` ersetzt, die einen entsprechenden Iterator zurückgeben.

Listing 11.3: Variante 3: mit Template-Container vector

```
// cppbuch/k11/einf/mitVector.cpp
#include<iostream>
#include<vector>
using namespace std;

// Prototyp des Algorithmus ersetzt durch Template
template<typename Iterator, typename T>
Iterator find(Iterator begin, Iterator end, const T& Value) {
    while(begin != end // Iterator-Vergleich
           && *begin != Value) // Dereferenzierung und Objektvergleich
        ++begin; // nächste Position
    return begin;
}

// neuer Typname zur Verbesserung der Lesbarkeit; vector<int>::const_iterator ist vordefiniert.
typedef vector<int>::const_iterator Iteratortype;

int main() {
    const int ANZAHL = 100;
    vector<int> einContainer(ANZAHL); // Container definieren
    // Container mit beliebigen Werten füllen (hier: gerade Zahlen)
    for(int i = 0; i < ANZAHL; ++i) {
        einContainer[i] = 2*i;
    }
}
```

```

}
int zahl = 0;
while(zahl != -1) {
    cout << " gesuchte Zahl eingeben (-1 = Ende):";
    cin >> zahl;
    if(zahl != -1) {          // weitermachen?
        // globales find() von oben benutzen
        IteratorType position =
            ::find(einContainer.begin(), // Container-Methoden
                  einContainer.end(), zahl);
        if (position != einContainer.end())
            cout << "gefunden an Position "
                  << (position - einContainer.begin()) << endl;
    }
    else {
        cout << zahl << " nicht gefunden!" << endl;
    }
}
}

```

Man sieht, wie der Standard-Container mit unserem Algorithmus zusammenarbeitet und wie Arithmetik mit Iteratoren möglich ist (Differenzbildung). Im letzten Schritt verwende ich den in der C++-Standardbibliothek vorhandenen `find()`-Algorithmus und ersetze das gesamte Template durch eine weitere `#include`-Anweisung:

Listing 11.4: Variante 4: Algorithmus `find()` der C++-Standardbibliothek

```

// Auszug aus cppbuch/k11/einf/mitSTL.cpp
#include<algorithm>
// ... Rest wie Variante 3, aber ohne find()-Template. Der Aufruf ::find() wird
// durch find() ersetzt (d.h. Namespace std).

```

Darüber hinaus ist es nicht erforderlich, mit `typedef` einen Iteratortyp zu definieren, weil jeder Container der STL einen entsprechenden Typ liefert. Anstatt `IteratorType position` kann im obigen Programm `vector<int>::iterator position` geschrieben werden – oder noch einfacher: `auto position`.

Interessant ist, dass der Algorithmus mit *jeder* Klasse von Iteratoren zusammenarbeiten kann, die die Operationen `!=` zum Vergleich, `*` zur Dereferenzierung und `++` zur Weiter-schaltung auf das nächste Element zur Verfügung stellt. Dies ist ein Grund für die Mächtigkeit des Konzepts und für die Tatsache, dass jeder Algorithmus nur in *einer* Form vorliegen muss, womit Verwaltungsprobleme minimiert und Inkonsistenzen ausgeschlossen werden. Durch Verwendung der vorhandenen Algorithmen und Container werden Programme nicht nur kürzer, sondern auch zuverlässiger, weil Programmierfehler vermieden werden. Die Produktivität der Softwareentwicklung steigt damit.

11.2 Iteratoren im Detail

Die sequenzielle Bearbeitung einer Datenstruktur, die keine Liste sein muss, durch eine Methode, die nach außen hin *die innere Struktur der Daten verbirgt*, heißt *Kontrollabstraktion*. Das »Durchwandern« der Datenstruktur in einer bestimmten Reihenfolge kann sinnvoll sein, um auf jedes Element eine bestimmte Operation anzuwenden, zum Beispiel Ausdrucken des Inhalts. Der Benutzer soll auf die Elemente der Reihe nach zugreifen können, ohne Kenntnis über die verborgene Implementierung haben zu müssen. Ein Prinzip zur Kontrollabstraktion dieser Art wird *Iterator* genannt.

Ein Iterator ist ein Konzept, das auf eine Menge von Klassen und Typen zutrifft, die bestimmten Anforderungen entsprechen. Ein Iterator kann auf verschiedene Weise mit Grunddatentypen oder Klassenobjekten realisiert werden. Weil ein Iterator dazu dient, auf Elemente eines Containers zuzugreifen, ist er eine Art verallgemeinerter Zeiger. Mehr zu den Iteratoren der C++-Standardbibliothek lesen Sie ab Seite [805](#). Wesentlich für alle Iteratoren sind die bereits genannten Fähigkeiten des Weberschaltens (++), der Dereferenzierung (*) und der Vergleichsmöglichkeit (!= bzw. ==). Falls der Iterator nicht ein gewöhnlicher Zeiger, sondern ein Objekt einer Iterator-Klasse ist, werden diese Eigenschaften durch die entsprechenden Operatorfunktionen realisiert:

Listing 11.5: Schema eines einfachen Iterators

```
template<typename T>
class Iteratortyp {
public:
    // Konstruktoren, Destruktor weggelassen
    bool operator==(const Iteratortyp<T>&) const;
    bool operator!=(const Iteratortyp<T>&) const;
    Iteratortyp<T>& operator++();           // präfix
    Iteratortyp<T> operator++(int);        // postfix
    T& operator*() const;
    T* operator->() const;
private:
    // Verbindung zum Container ...
};
```

Der Operator `->` erlaubt es, einen Iterator wie einen Zeiger zu verwenden. Man kann sich bei einem Vector-Container natürlich vorstellen, dass der Iterator auch eine Methode `operator--()` haben sollte. Die wesentlichen Eigenschaften eines Iterators sind also:

1. Ein Iterator kann wie ein Zeiger benutzt werden.
2. Wie das Vorgehen mit dem `++`- bzw. `--`-Operator intern funktioniert, ist dem Benutzer verborgen. So wird ein Iterator `I` nur mit dem Befehl `++I` um eine Position weiterschaltet. Die möglicherweise damit verbundene komplexe Operation, zum Beispiel den nächsten Eintrag in einem binären Suchbaum zu finden, ist für den Benutzer nicht sichtbar.

Zustände

Iteratoren erlauben es, mit verschiedenen Containern auf gleichartige Weise zu arbeiten. Ein Iterator kann verschiedene Zustände haben.

- Ein Iterator kann erzeugt werden, auch ohne dass er mit einem Container verbunden ist. Die Verbindung zu einem Container wird dann erst nachträglich hergestellt. Ein solcher Iterator ist nicht dereferenzierbar. Ein vergleichbarer C++-Zeiger könnte zum Beispiel den Wert 0 haben.
- Ein Iterator kann während der Erzeugung oder danach mit einem Container verbunden werden. Typischerweise – aber nicht zwingend – zeigt er nach der Initialisierung auf den Anfang des Containers. Die Methode `begin()` eines Containers liefert die Anfangsposition. Wenn der Container nicht leer ist, ist der Iterator in diesem Fall dereferenzierbar. Man kann also über ihn auf ein Element des Containers zugreifen. Der Iterator ist mit Ausnahme der `end()`-Position (siehe nächster Punkt) für alle Werte dereferenzierbar, die mit der Operation `++` erreicht werden können.
- In C++ ist der Wert eines Zeigers, der auf die Position direkt *nach* dem letzten Element eines C-Arrays zeigt, stets definiert. In Analogie dazu gibt die Methode `end()` eines Containers einen Iterator mit eben dieser Bedeutung zurück, auch wenn der Container kein Array, sondern zum Beispiel eine Liste ist. Damit können Iteratorobjekte und Zeiger auf C++-Grunddatentypen gleichartig behandelt werden. Der Vergleich eines laufenden Iterators mit diesem Nach-dem-Ende-Wert signalisiert, ob das Ende eines Containers erreicht wurde. Ein Iterator, der auf die Position nach dem Ende eines Containers verweist, ist natürlich nicht dereferenzierbar.

Auf die verschiedenen sinnvollen, möglichen Fähigkeiten eines Iterators und weitere Eigenschaften wird in Kapitel 29 eingegangen. Im nächsten Abschnitt wird ein Beispiel mit einem Iterator, der mehr können muss als ein Zeiger auf ein `int`-Array, gezeigt

11.3 Beispiel verkettete Liste

Wie funktioniert das Zusammenwirken von Containern, Iteratoren und Algorithmen genau? Um dies im Einzelnen zu zeigen, verwende ich das `main`-Programm aus Abschnitt 11.1 in leicht variiert Form. Damit ein Iterator dieser Klasse nicht einfach einem Zeiger gleichgesetzt werden kann, muss die Komplexität des Beispiels geringfügig erhöht werden: Eine einfach verkettete Liste tritt an die Stelle des Vektors. Die Klasse werde `Liste` genannt. Eine einfach verkettete Liste ist wohl die bekannteste dynamische Datenstruktur. Sie besteht aus Elementen, die miteinander über Zeiger verbunden sind (siehe Abbildung 11.1).

Ein Element einer einfach verketteten Liste besteht aus Daten, zum Beispiel einer Adresse, und einem Verweis auf das nächste Element. Der Nachteil der einfach verketteten Liste besteht darin, dass man sie nur in Vorwärtsrichtung bearbeiten kann, weil die Information über das Vorgängerelement in den Elementen der Liste nicht vorhanden ist. Es gibt verschiedene Möglichkeiten, auf eine Liste zuzugreifen:

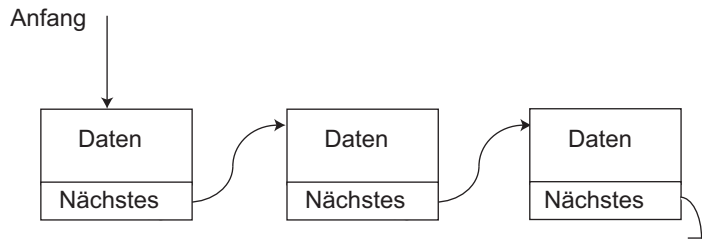


Abbildung 11.1: Einfach verkettete Liste

- a) Einfügen eines Elements am Anfang
- b) Einfügen eines Elements am Ende
- c) Einfügen eines Elements dazwischen
- d) Lesen und Entfernen eines Elements am Anfang
- e) Lesen und Entfernen eines Elements am Ende

und andere mehr. Eine Liste, bei der nur die Operationen a) und e) zugelassen sind, heißt *Warteschlange* (englisch *queue*) und arbeitet nach dem *fifo*-Prinzip (*fifo* = first in, first out). Falls nur die Operationen a) und d) (beziehungsweise b) und e)) erlaubt sind, heißt die Liste *Stapel* oder Kellerspeicher (englisch *stack*). Auf einen Stapel kann man nur von oben etwas legen oder wegnehmen, er arbeitet nach dem *lifo*-Prinzip (*lifo* = last in, first out). In eine (allgemeine) Liste kann auch mittendrin eingefügt oder entnommen werden.

Im folgenden Beispiel beschränke ich mich auf a). Es gibt keinen Indexoperator und somit keinen wahlfreien Zugriff auf die Elemente. Ferner wird keinerlei Rücksicht auf Laufzeitverhalten genommen, um die Klasse so einfach wie möglich zu gestalten. Der vorhandene Algorithmus `std::find()` wird verwendet, um zu zeigen, dass sich die selbstgeschriebene Klasse wirklich genau wie eine Klasse der C++-Standardbibliothek verhält, wenigstens bezüglich des `main`-Programms unten.



Hinweis

Die Klasse für eine einfache Liste ist nicht vollständig und daher nicht sicher benutzbar! Sie stellt nur das zur Verfügung, was im Beispiel benötigt wird. Konsequenz: Nachdem Ihnen die Wirkungsweise klar ist, benutzen Sie für weitere Zwecke bitte die Klasse `list` der C++-Standardbibliothek (siehe Seite 772).

Die Liste besteht aus Elementen, deren Typ innerhalb der Klasse als geschachtelte `public`-Klasse (`struct`) definiert ist. In einem `struct`-Objekt ist der Zugriff auf interne Daten möglich. Dies stellt hier aber kein Problem dar, weil die Klasse innerhalb der `private`-Sektion der Klasse `Liste` definiert ist. Jedes Listenelement trägt die Daten (zum Beispiel eine Zahl) mit sich sowie einen Zeiger auf das nächste Listenelement. Die Klasse `Liste` stellt einen öffentlichen Iteratortyp `iterator` zur Verfügung. Ein Iterator-Objekt verweist auf die aktuelle Position in der Liste (Attribut `aktuellesElement` der Klasse `Liste::iterator`) und ist geeignet, darüber auf die in der Liste abgelegten Daten zuzugreifen, wie das Beispiel unten zeigt. Die Iteratormethoden erfüllen die auf Seite 401 beschriebenen Anforderungen.

Listing 11.6: Klasse für eine Liste (unvollständig)

```

// cppbuch/k11/liste/liste.t
// Unvollständig! Nur für das Beispiel notwendige Funktionen sind implementiert!
#ifndef LISTE_T
#define LISTE_T
#include<cstddef> // ptrdiff_t

template<typename T>
class Liste {
public:
    Liste() : anfang(0), anzahl(0) { }
    // Kopierkonstruktor, Destruktor und Zuweisungsoperator weggelassen! (siehe Aufgaben)
    // push_front() erzeugt ein neues Listenelement und fügt es am Anfang der Liste ein:
    void push_front(const T& wert) {
        anfang = new ListElement(wert, anfang);
        ++anzahl;
    }
private:
    struct ListElement {
        T daten;
        ListElement *naechstes;
        ListElement(const T& wert, ListElement* p)
            : daten(wert), naechstes(p) {}
    };
    ListElement *anfang;
    size_t anzahl;
public:
    class iterator {
    public:
        // von find() geforderte Schnittstelle:
        typedef std::forward_iterator_tag iterator_category;
        typedef T value_type;
        typedef T* pointer;
        typedef T& reference;
        typedef ptrdiff_t difference_type;
        iterator(ListElement* init = 0) // Konstruktor
            : aktuellesElement(init) {}
        T& operator*() { // Dereferenzierung
            return aktuellesElement->daten;
        }
        const T& operator*() const { // Dereferenzierung
            return aktuellesElement->daten;
        }
        iterator& operator++() { // Präfix
            if(aktuellesElement) // noch nicht am Ende angelangt?
                aktuellesElement = aktuellesElement->naechstes;
            return *this;
        }
        iterator operator++(int) { // Postfix
            iterator temp = *this;
            ++*this;
            return temp;
        }
    };
};

```

```

    }
    bool operator==(const iterator& x) const {
        return aktuellesElement == x.aktuellesElement;
    }
    bool operator!=(const iterator& x) const {
        return aktuellesElement != x.aktuellesElement;
    }
private:
    ListElement* aktuellesElement;
}; // iterator
// Einige Methoden der Klasse Liste benutzen die Klasse iterator:
iterator begin() const { return iterator(anfang);}
iterator end() const { return iterator();}
};
#endif

```

Das zugehörige `main`-Programm verwendet die Listenkasse. Eine gefundene Zahl wird nicht mit der Variablen `zahl` angezeigt, was auch möglich wäre, sondern mit dem dereferenzierten Iterator.

Listing 11.7: Anwendung der Liste

```

// cppbuch/k11/liste/main.cpp
#include<iostream>
#include<algorithm>
#include"liste.t"
using namespace std;

int main() {
    const int ANZAHL = 100;
    Liste<int> einContainer; // Container definieren

    // Container mit beliebigen Werten füllen (hier: gerade Zahlen)
    for(int i = 0; i < ANZAHL; ++i) {
        einContainer.push_front(2*i);
    }

    int zahl = 0;
    while(zahl != -1) {
        cout << " gesuchte Zahl eingeben (-1 = Ende):";
        cin >> zahl;
        if(zahl != -1) { // weitermachen?
            // std::find() benutzen
            auto position = // siehe Text unten
                find(einContainer.begin(), // Container-Methoden
                    einContainer.end(), zahl);
            if (position == einContainer.end()) {
                cout << " nicht gefunden!";
            }
            else {
                cout << *position // Dereferenzierung des Iterators
                    << " gefunden!" << endl;
            }
        }
    }
}

```

```

    }
    cout << "Liste ausgeben:" << endl;
    for(auto iter = einContainer.begin();    // siehe Text unten
        iter != einContainer.end(); ++iter) {
        cout << *iter << endl;
    }
}

```

In diesem Programm kommt das Schlüsselwort `auto` von Seite 90 zur Geltung, um nicht `Liste<int>::iterator` schreiben zu müssen. Statt zum Beispiel

```

for(Liste<int>::iterator iter = einContainer.begin();
    iter != einContainer.end(); ++iter) {
    cout << *iter << endl;
}

```

heißt es einfach

```

for(auto iter = einContainer.begin();
    iter != einContainer.end(); ++iter) {
    cout << *iter << endl;
}

```

Was der Klasse `Liste` noch fehlt, um sie zu einer vollwertigen Klasse im Sinne der C++-Standardbibliothek zu machen, können Sie Abschnitt 28.2.3 entnehmen.

Diese Einführung zeigt die nur Funktionsweise der STL. Mehr Informationen zu den Containern, Iteratoren und Algorithmen der C++-Standardbibliothek finden Sie in weiteren Kapiteln:

- Container: Kapitel 28
- Iteratoren: Kapitel 29
- Algorithmen: Kapitel 24 und 30



Übung

11.1 Ergänzen Sie die Klasse `Liste` wie folgt und versuchen Sie, Lösungen der Teilaufgaben wiederzuverwenden.

- Kopierkonstruktor und Zuweisungsoperator. Der letztere kann vorteilhaft den ersten einsetzen, indem erst eine temporäre Kopie der Liste erzeugt wird und dann die Verwaltungsinformationen vertauscht werden.
- Destruktor.
- Methode `iterator erase(iterator p)`, die das Element, auf das der Iterator `p` zeigt, aus der Liste entfernt. Der zurückgegebene Iterator soll auf das nach `p` folgende Element zeigen, sofern es existiert. Andernfalls soll `end()` zurückgegeben werden. Ergänzen Sie vorher die Klasse `iterator` um `friend class Liste`, damit `erase()` auf das zu löschende Element zugreifen darf.
- Methode `void pop_front()`, die das erste Element löscht.
- Methode `void clear()`, die die ganze Liste löscht.
- Methode `bool empty()`, die anzeigt, ob die Liste leer ist.
- Methode `size_t size()`, die die Anzahl der Elemente zurückgibt.

12

Reguläre Ausdrücke

Dieses Kapitel behandelt die folgenden Themen:

- Grundlagen regulärer Ausdrücke
- Interaktiver Test
- Auszug der Programmierschnittstelle
- Praktische Anwendungen in C++

Ein regulärer Ausdruck ist eine Zeichenkette, die eine oder mehr grammatische Regeln enthält und zum Suchen oder zur Textersetzung dienen kann. Aus der Informatik wissen Sie, dass reguläre Ausdrücke mit einem endlichen Zustandsautomat (englisch *finite state machine*) verarbeitet werden können. Die Leistungsfähigkeit der Programmiersprache Perl basiert wesentlich auf regulären Ausdrücken. Die Programmiersprache Java stellt seit langem reguläre Ausdrücke zur Verfügung. So verwundert es nicht, dass das C++-Standardkomitee sich entschlossen hat, sie in den C++-Standard aufzunehmen.

Reguläre Ausdrücke sind ein leistungsfähiges Werkzeug. So lassen sich mit einem Befehl alle versehentlichen (und beabsichtigten) Wortverdopplungen wie »und und« in allen tex-Dateien eines Verzeichnisses auffinden, wobei die Klein- bzw. Großschreibung keine Rolle spielt:

```
egrep -n -i '\b([a-z]+) +\1\b' *.tex
```

Das folgende Kommando ersetzt jedes Vorkommen von »Version 1.2« und »Version 1.3« in der Datei *a.txt* durch »Version 2.0« bzw. »Version 3.0« und schreibt das Ergebnis in die Datei *b.txt*. Überzählige Leerzeichen nach »Version« werden gelöscht:

```
sed "s/ Version \+1\.\([23]\)/ Version \1.0/g" a.txt > b.txt
```

So wird aus »Nach Version 1.1 kamen Version 1.2 und Version 1.3.«:

»Nach Version 1.1 kamen Version 2.0 und Version 3.0.«

Warum das so ist, wird weiter unten erklärt. Dieses Kapitel konzentriert sich nach einer kurzen Einführung, die die wichtigsten Aspekte abdeckt, auf die Anwendung regulärer Ausdrücke mit C++. Zum tieferen Verständnis des Themas empfehle ich das exzellente Buch von Friedl [Fri] und als C++-spezifische Ergänzung [BeckP]. Es gibt mehrere Grammatiken (Dialekte) für reguläre Ausdrücke, die sich in großen Teilen nur wenig unterscheiden. Die gewünschte Grammatik kann im Konstruktor eines regulären Ausdrucks angegeben werden. So wird mit

```
regex rgx(regexAusdruck, regex_constants::egrep);
```

die Grammatik gewählt, die von dem am Anfang des Kapitels erwähnten Programm *egrep* benutzt wird (`regexAusdruck` ist eine Zeichenkette des Typs `const char*` oder `string`). Falls keine Grammatik angegeben wird, ist Ecma-262 [Ecma] mit einigen Modifikationen voreingestellt ([ISOC++], Kap. 28.13). Ecma-262, früher JavaScript, bezieht sich bezüglich der regulären Ausdrücke auf Perl 5. Im Folgenden wird von dieser Voreinstellung ausgegangen. Auch werden reguläre Ausdrücke auf ASCII-Zeichen des Typs `char` beschränkt, obwohl es Erweiterungen für `wchar_t`-Zeichen und Unicode gibt.

12.1 Elemente regulärer Ausdrücke

Wie an den einführenden Beispielen zu sehen, haben manche Zeichen besondere Bedeutung. Sie werden Meta-Zeichen genannt:

- Ein Punkt steht für ein beliebiges Zeichen. Der reguläre Ausdruck `.` trifft damit auf jedes Zeichen der Folge `abc123` zu.
- | gibt eine Alternative an. `a|b` heißt `a` oder `b`.
- () Runde Klammern dienen zur Gruppierung. Wenn `a`, gefolgt von `b` oder `c`, gemeint ist, schreibt man `a(b|c)`. Eine weitere Bedeutung ist die Gruppierung zwecks späterer Identifikation der Gruppe (Beispiel folgt).
- (?:) Mit `(?:)` wird eine Gruppierung markiert, die *nicht* für eine späterer Identifikation vorgesehen ist. Beispiel: `(?:a)`.
- \ Der Backslash (Escape-Zeichen) hat mehrere Bedeutungen:
 - Wenn er einem Meta-Zeichen vorangestellt wird, ist nicht das Meta-Zeichen gemeint, sondern das zugehörige Literal. Der reguläre Ausdruck `\\.` trifft damit nur auf den Punkt in der Folge `ab.c123` zu.

- Ein `\`, gefolgt von einem besonderen Buchstaben, kann zu einem ASCII-Steuerzeichen oder einem Meta-Zeichen werden. So meint `^` das Tabulatorzeichen, während `\d` für eine beliebige Ziffer steht. Falls dem nachfolgenden Buchstaben keine besondere Bedeutung zukommt, ist dieser Buchstabe gemeint. So entspricht `\m` genau dem Buchstaben `m` (siehe Tabelle 12.1).

Tabelle 12.1: Escape-Zeichen

Zeichen	Bedeutung
<code>\\</code>	Steuerzeichen (Auswahl): Backslash
<code>\f</code>	Seitenvorschub
<code>\n</code>	neue Zeile
<code>\r</code>	CR
<code>\t</code>	horizontaler Tabulator
<code>\v</code>	vertikaler Tabulator
<code>\d</code>	Zeichenklassen: Ziffer, dasselbe wie <code>[0-9]</code>
<code>\D</code>	keine Ziffer, dasselbe wie <code>[^0-9]</code> oder <code>[^\d]</code>
<code>\s</code>	Zwischenraumzeichen, dasselbe wie <code>[\f\n\r\t\v]</code>
<code>\S</code>	kein Zwischenraumzeichen, dasselbe wie <code>[^\s]</code>
<code>\w</code>	Wortzeichen, in ASCII <code>[a-zA-Z0-9]</code>
<code>\W</code>	kein Wortzeichen, d.h. <code>[^\w]</code>
<code>\b</code>	Begrenzer: Wortgrenze (siehe unten)
<code>\B</code>	keine Wortgrenze

- Ein `\`, gefolgt von einer Ziffer, ist ein Verweis auf eine Gruppe, die vorher gefunden wurde. `(abc)xy\1` stimmt daher mit `abcxyabc` überein. Nach `xy` steht `\1` für die erste gefundene Gruppe, hier `abc`.
 - Wenn ein Backslash-Literal gemeint ist, ist `\\` zu schreiben. Da in einem C-String `\` als `\\` geschrieben wird, heißt das, dass ein Backslash, der als regulärer Ausdruck `\\` geschrieben wird, in einem C-String vierfach, also als `\\\\`, auftritt.
- [] kennzeichnet eine Zeichenklasse. `[xy]` passt auf alle `x` und alle `y` einer Folge. `[a-z]` passt auf jeden Kleinbuchstaben einer Zeichenfolge, `[0-9]` auf jede Ziffer.
- [^] negiert die Bedeutung. `[^a-z]` passt auf jedes Zeichen, das kein Kleinbuchstabe ist.
- + ist wie die folgenden Meta-Zeichen ein Quantifizierer. + passt nur, wenn das vorstehende Element mindestens einmal vorkommt. So passt `a+` sowohl auf `abc` wie auf `aaabc`, nicht aber auf `bc`.
- ? passt, wenn das vorstehende Element einmal oder gar nicht vorkommt.
- * passt, wenn das vorstehende Element beliebig oft vorkommt. Das schließt die Möglichkeit, dass das Element nicht vorkommt, mit ein.
- {n} passt, wenn das vorstehende Element genau `n`-mal vorkommt. `a{3}` passt auf `aaa`, aber nicht auf `aa`. Die Erweiterung `{n, m}` passt, wenn das vorstehende Element mindestens `n`-mal und maximal `m`-mal vorkommt.

- ˆ Außerhalb einer Zeichenklasse ist ˆ wie das folgenden Meta-Zeichen \$ ein *Anker*. Ein Anker markiert kein Zeichen, sondern eine Position. ˆ passt auf den Anfang einer Zeile.
- \$ passt auf das Ende einer Zeile oder eines Strings. Der reguläre Ausdruck \d\d passt auf die Teilfolgen 12 und 45 im String a123b456. Wird das \$-Zeichen angefügt, d.h. \d\d\$, passt der Ausdruck nur auf 56.
- \b ist eine Wortgrenze. a\b meint also das Zeichen a am Ende eines Worts. a\b passt nirgends in xyz abc, wohl aber auf das erste a von xyza abc.
- \B ist keine Wortgrenze. a\b passt nur auf das zweite a von xyza abc.

Die Zeichenklasse [a-z] bezeichnet die ASCII-Kleinbuchstaben. Die aus dem POSIX-Standard übernommene Zeichenklasse [:lower:] bezeichnet die Kleinbuchstaben in Abhängigkeit von der Locale-Einstellung, sodass bei Einstellung der Locale de_DE auch Umlaute erfasst werden. In Analogie dazu gibt es weitere Zeichenklassen, deren Namen sich aus den Funktionsnamen der Tabelle 35.2 auf Seite 875 ergeben, wenn das Präfix is weggelassen wird. Aus islower() ergibt sich [:lower:].

Damit sind die wichtigsten Elemente genannt. Es gibt weitere, die aus Platzgründen hier nicht aufgeführt werden. Mit der Bitte um Verständnis verweise ich auf die schon erwähnte Literatur [Fri] und [BeckP].

Verknüpfungen

In Tabelle 12.1 ist in der Zeile zu \w zu sehen, dass in Zeichenklassen kombiniert werden kann. [a-zA-Z0-9] enthält die alphanumerischen Zeichen. Mit der &&-Verknüpfung lässt sich eine Subtraktion darstellen: [a-zA-Z&&[~aeiouy]] bedeutet, dass die Menge der Vokale von der Menge der Kleinbuchstaben abgezogen wird. Die Zeichenklasse enthält nur noch kleingeschriebene Konsonanten.

12.1.1 Greedy oder lazy?

Bei der Auswertung regulärer Ausdrücke wird versucht, eine passende Zeichenkette möglichst großer Länge zu finden. Deshalb wird dieses Vorgehen »gierig« (englisch *greedy*) genannt. Die in der obigen Aufstellung genannten Quantifizierer +, *, ? usw. sind alle *greedy*. Manchmal möchte man aber die kürzestmögliche passende Zeichenkette finden. Dafür gibt es die »träge« oder »faule« (englisch *lazy*) Auswertung. *Lazy* Quantifizierer unterscheiden sich syntaktisch von Greedy-Quantifizierern durch ein nachgestelltes ?, zum Beispiel a?? für ein optionales a. Um den Unterschied zu verdeutlichen, wird ein regulärer Ausdruck gesucht, der aus einer Zeile einen Kommentar herausfiltern soll. Als Teststring wird

```
xyz /* hallo */ abc */ 123
```

vorgegeben. Sie sehen, dass es zwei Positionen für das Kommentarende */ gibt. Im C++-Sinn ist nur die erste korrekt.

Greedy ...

Betrachten Sie den regulären Ausdruck /*.**/. Er passt auf alle Zeichenketten, die

- mit einem Schrägstrich / beginnen, gefolgt

- von einem *. Bitte beachten Sie, dass das Meta-Zeichen * mit einem Escape-Zeichen maskiert werden muss, weil das Literal und nicht der Quantifizierer gemeint ist.
- Anschließend folgen beliebige Zeichen, mit dem Meta-Zeichen . symbolisiert. Die Anzahl dieser Zeichen kann null oder beliebig groß sein, wie das nachfolgende Meta-Zeichen * festlegt.
- Abschließend folgt das Literal *, gefolgt von /.

Nun gehören auch die Zeichen * und / zu den beliebigen Zeichen, sodass die Auswertungsmaschinerie, die den ganzen String untersucht, weitermacht, bis das Ende erreicht ist. Die letzte unterwegs gefundene Möglichkeit, eine Übereinstimmung zu erreichen, ist das zweite Vorkommen von */. Der reguläre Ausdruck `/\.*\*/` passt daher auf die Zeichenkette `/* hallo */ abc */` – nicht der gesuchte C++-Kommentar. Die »gierige« Strategie führt manchmal nicht zum Erfolg.

... oder lazy?

Die gierige Auswertung kann mit einem ?, das dem Quantifizierer folgt, verhindert werden. In diesem Fall lautet der modifizierte reguläre Ausdruck `/\.*?\*/`. Angewendet auf den Teststring passt dieser Ausdruck auf die Zeichenkette `/* hallo */`, das heißt, es wird die erste passende Möglichkeit genommen.



Tip

Vermeiden Sie die greedy-Suche mit `.*`, wenn Sie keinen besonderen Grund dafür haben.

Begründungen: `.*` ohne Lazy-Quantifizierer verschwendet Rechenzeit durch überflüssiges Durchlaufen der Zeichenkette und Backtracking. Das Beispiel `/* hallo */ ... 1000 Zeichen ohne Kommentare` macht es deutlich. Außerdem ist manchmal die kürzeste passende Zeichenkette gewünscht, nicht die längste.



Tip

Eine Alternative zu `.*` ist die Greedy-Suche mit einer Zeichenklasse, die die Negation eines zu suchenden Zeichens enthält.

Der Ausdruck `"<.*>"` findet HTML-Tags, sucht aber bis zum Ende. Die Alternative ist hier, den Punkt durch ein Zeichen, das nicht der Enderkennung entspricht, zu ersetzen: `"<[^>]*>"`. Diese greedy-Suche ist effizient, weil sie beim ersten `>` anhält.

12.2 Interaktive Auswertung

Reguläre Ausdrücke sind oft nicht leicht zu lesen. Die Änderung nur eines einzigen Zeichens verleiht dem Ausdruck eine andere Bedeutung, wie am obigen Beispiel zu sehen ist. Deswegen ist es sinnvoll, einen Ausdruck vor dem festen Einbau in ein Programm zu evaluieren. Die Klasse `RegexTester` ist dafür gut geeignet.

**Hinweis**

Der Compiler g++ bis einschließlich Version 4.5 kann noch keine regulären Ausdrücke, weswegen Boost eingesetzt wird. Die regulären Ausdrücke der Boost-Bibliothek sind in den C++-Standard aufgenommen worden, nur sind die Compiler größtenteils noch nicht angepasst.

Listing 12.1: Klasse RegexTester

```
// cppbuch/k12/regextester/RegexTester.h
#ifndef REGEX_TESTER
#define REGEX_TESTER
#include<boost/regex.hpp>    // siehe Hinweis oben
#include<string>

class RegexTester {
public:
    RegexTester(const char* regEx, const char* teststr);
    void run();
private:
    boost::regex rgx;        // siehe Hinweis oben
    std::string teststring;
};
#endif
```

Dem Hauptprogramm werden bei Aufruf ein regulärer Ausdruck und der auszuwertende String übergeben. Ausgegeben werden alle Zeichen und Positionen, auf die der reguläre Ausdruck zutrifft.

Listing 12.2: Hauptprogramm zum interaktiven Testen

```
// Auszug aus cppbuch/k12/regextester/testRegex.cpp
#include <iostream>
#include "RegexTester.h"
using namespace std;

int main(int argc, char* argv[]) {
    if(3 != argc) {
        cout << "Gebrauch: testRegex.exe \"regex\" \"teststring\"" << endl;
    }
    else {
        try {
            RegexTester rt(argv[1], argv[2]);
            rt.run();
        } catch(boost::regex_error& re) {
            std::cerr << "Fehler: " << re.what() << std::endl;
        }
    }
}
```

Die Implementation der Klasse RegexTester demonstriert bereits einige Möglichkeiten der Verarbeitung regulärer Ausdrücke mit C++. Das Listing wird unten im Detail diskutiert.

Listing 12.3: RegexTester-Implementierung

```
// cppbuch/k12/regextester/RegexTester.cpp
#include <iostream>
#include "RegexTester.h"

RegexTester::RegexTester(const char* regEx, const char* teststr)
    : rgx(regEx), teststring(teststr) {
}

void RegexTester::run() {
    boost::sregex_iterator erster(teststring.begin(), teststring.end(), rgx),
        letzter;
    std::cout << "Regex: " << rgx
        << " Teststring: " << teststring << std::endl;
    if (erster == letzter) {
        std::cout << "nichts gefunden" << std::endl;
    }
    else {
        while(erster != letzter) {
            boost::match_results<std::string::const_iterator>
                ergebnis = *erster++;
            for(size_t i = 0; i < ergebnis.size(); ++i) {
                if(i > 0) {
                    std::cout << "Capturing Group " << i << ": ";
                }
                std::cout << "\"" << ergebnis.str(i) << "\" gefunden. Position "
                    << ergebnis.position(i);
                if(ergebnis.length(i) > 1) {
                    std::cout << " bis "
                        << ergebnis.position(i) + ergebnis.length(i)-1;
                }
                std::cout << std::endl;
            }
        }
    }
}
```

Der Konstruktor initialisiert das Attribut des Typs `regex` mit dem als C-String übergebenen regulären Ausdruck und merkt sich den Teststring. Die Klasse `regex` ist die Klasse für die Verarbeitung regulärer Ausdrücke.

Die Klasse `sregex_iterator` ist die Spezialisierung `regex_iterator<string::const_iterator>` der zugrunde liegenden Template-Klasse. Dem damit erzeugten Iterator `erster` werden der zu untersuchende Bereich des Teststrings und der zu verwendende reguläre Ausdruck übergeben. Der Iterator `letzter` hat die in der STL übliche Funktion, als Endekriterium zu dienen.

Die Klasse `match_results` ist eine Sammlung von Zeichenfolgen, die das Ergebnis der Auswertung repräsentieren. Es kann mehrere Ergebnisse geben, die der Iterator jeweils der Reihe nach dem Objekt `ergebnis` zuweist.

Der Aufruf von `ergebnis.size()` gibt 1 zurück plus die Anzahl gefundener Unterausdrücke, die eine Gruppe darstellen (englisch *capturing group*). In der Schleife werden das

(mit `str()` in einen String umgewandelte) Ergebnis und die gefundene Position ausgegeben. Wenn die Übereinstimmung mehrere Zeichen umfasst, wird auch die Endposition angezeigt. Die Beispiele zeigen den Programmaufruf und direkt danach die Ausgabe des Programms.

- Programmaufruf: `testRegex.exe "[a-c]+" "zcbaxcadey"`

```
Regex: [a-c]+ Teststring: zcbaxcadey
"cbaa" gefunden. Position 1 bis 4
"ca" gefunden. Position 6 bis 7
```

- Programmaufruf: `testRegex.exe "[a-c]*" "z"`

```
Regex: [a-c]* Teststring: z
"" gefunden. Position 0
"" gefunden. Position 1
```

Die Erklärung: Der Quantifizierer `*` erlaubt es, dass ein Element gar nicht auftritt. Das ist hier am Anfang und am Ende gegeben.

- Programmaufruf: `testRegex.exe "^\\D+([0-9]+) \\1" "Sommer 2012 2012"`

```
Regex: ^\D+([0-9]+) \1 Teststring: Sommer 2012 2012
"Sommer 2012 2012" gefunden. Position 0 bis 15
Capturing Group 1: "2012" gefunden. Position 7 bis 10
```

Erklärung: Wie oben schon erläutert, wird ein Backslash in einem C-String als `\\` dargestellt. Deswegen ist die Dopplung auch bei der Eingabe erforderlich. Dieser reguläre Ausdruck verlangt eine Folge von Nicht-Zifferzeichen (`\\D`), die aus mindestens einem Zeichen besteht (`+`), und die am Anfang des Teststrings beginnt (`^`). Dies wird durch die Zeichenfolge »Sommer« einschließlich des Leerzeichens erfüllt. Es schließt sich eine Ziffernfolge (`[0-9]`) mit mindestens einer Ziffer an (`+`). Diese Ziffernfolge wird wegen der runden Klammern als Gruppe gespeichert. Nach den Ziffern folgt ein Leerzeichen und dann die erste gefundene Gruppe (`\\1`), hier 2012.

12.3 Auszug des regex-APIs

Aus Platzgründen kann hier nur der wichtigste Teil der Programmierschnittstelle für reguläre Ausdrücke dargestellt werden. Weitere Informationen bitte ich, [\[ISOC++\]](#) zu entnehmen. Der Header ist `<regex>`. Die Klasse `regex` ist ein anderer Name für die Spezialisierung `basic_regex<char>`. Die entsprechende Klasse für `wchar_t` heißt `wregex`. Im folgenden Text beschränke ich mich auf `regex` und lasse auch viele vordefinierte Parameter weg, weil die Standardeinstellung meistens genügt.

- `regex(const char*)` und `regex(const string&)`
sind Konstruktoren, die ein `regex`-Objekt nach dem vorgegebenen Standard Ecma-262 [\[Ecma\]](#) erzeugen. Falls ein anderer Standard gewünscht ist, kann er als Parameter übergeben werden, zum Beispiel

```
regex rgx(regexAusdruck, regex_constants::egrep);
```

Außer ECMAScript und egrep sind auch awk und grep möglich. Die Konstante `icase` sorgt dafür, dass Groß- und Kleinschreibung nicht unterschieden werden.

- `sregex_iterator` ist ein Iterator, mit dem über die Ergebnisse der Auswertung eines regulären Ausdrucks iteriert werden kann. Dabei ist die auszuwertende Zeichenkette vom Typ `string`. Falls der Typ `const char*` ist, muss ein `cregex_iterator` genommen werden. Beide Iterortypen sind Spezialisierungen des Klassen-Templates `regex_iterator`. Ein Beispiel finden Sie auf Seite [414](#).
- `match_results` ist das Klassen-Template zur Ergebnisablage. Ein `match_results`-Objekt ergibt sich aus der Dereferenzierung des `regex_iterator`s. `match_results` hat unter anderem die Methoden `size()`, `position()` und `length()`. Das genannte Beispiel von Seite [414](#) zeigt die Verwendung von `match_results` und seiner Methoden.

regex_match

- `bool regex_match(BidirectionalIterator first, BidirectionalIterator last, match_results& match, const regex& rgx)` stellt fest, ob der gesamte Bereich zwischen `first` und `last` dem regulären Ausdruck `rgx` entspricht. Falls ja, wird `true` zurückgegeben und `match` entsprechend geändert.
- `bool regex_match(const char* str, match_results& match, const regex& rgx)` gibt `regex_match(str, str+strlen(str), match, rgx)` zurück.
- `bool regex_match(const string& s, match_results& match, const regex& rgx)` gibt `regex_match(s.begin(), s.end(), match, rgx)` zurück.
- `bool regex_match(const char* str, const regex& rgx)` entspricht der zweitgenannten `regex_match()`-Funktion für den Fall, dass kein `match_results`-Objekt gebraucht wird.
- `bool regex_match(const string& s, const regex& rgx)` entspricht der an dritter Stelle genannten `regex_match()`-Funktion für den Fall, dass kein `match_results`-Objekt gebraucht wird.

regex_search

- `bool regex_search(BidirectionalIterator first, BidirectionalIterator last, match_results& match, const regex& rgx)` durchsucht den Bereich zwischen `first` und `last` entsprechend dem regulären Ausdruck `rgx`. Falls er für eine Teilfolge dazwischen zutrifft, wird `true` zurückgegeben und `match` entsprechend geändert.
- `bool regex_search(const char* str, match_results& match, const regex& rgx)` gibt `regex_search(str, str+strlen(str), match, rgx)` zurück.
- `bool regex_search(const string& s, match_results& match, const regex& rgx)` gibt `regex_search(s.begin(), s.end(), match, rgx)` zurück.
- `bool regex_search(BidirectionalIterator first, BidirectionalIterator last, const regex& rgx)` entspricht der erstgenannten `regex_search()`-Funktion für den Fall, dass kein `match_results`-Objekt gebraucht wird.
- `bool regex_search(const char* str, const regex& rgx)` gibt `regex_search(str, str+strlen(str), rgx)` zurück.
- `bool regex_search(const string& s, const regex& rgx)` gibt `regex_search(s.begin(), s.end(), rgx)` zurück. Ein Beispiel finden Sie auf Seite [636](#).

regex_replace

- `OutputIterator regex_replace(OutputIterator out, BidirectionalIterator first, BidirectionalIterator last, const regex& gesucht, string ersatz)`
kopiert den Bereich zwischen `first` und `last` über den Iterator `out` in die Ausgabe, wobei mit `gesucht` gefundene Übereinstimmungen durch `ersatz` ersetzt werden.
- `string regex_replace(const string& alles, const regex& gesucht, const string& ersatz)` gibt einen String zurück, der `alles` entspricht, nur dass dabei mit `gesucht` gefundene Übereinstimmungen durch `ersatz` ersetzt werden. Ein Beispiel finden Sie auf Seite 638.

12.4 Anwendungen

Reguläre Ausdrücke können vielfältig eingesetzt werden. Um Dopplungen zu vermeiden, wird an dieser Stelle auf das Kapitel 24 »Algorithmen für verschiedene Aufgaben« verwiesen. Zu den Programmen dort gibt es weitergehende Erläuterungen. In Kapitel 24 werden Lösungen auf Basis regulärer Ausdrücke in den folgenden Abschnitten angeboten:

- Datei durchsuchen: Abschnitt 24.2.1, Seite 636.
Verwendet werden `regex`, `regex::egrep`, `regex::icase`, `regex_search()`, `regex_error`.
- Ersetzungen in einer Datei vornehmen: Abschnitt 24.2.2, Seite 638.
Verwendet werden `regex`, `regex::egrep`, `regex::icase`, `regex_replace()`, `regex_error`.
- Lines of Code ermitteln: Abschnitt 24.2.4, Seite 641.
Verwendet werden `regex`, `regex_replace()`, `regex_error`.
- Erkennung eines Datums: Abschnitt 24.12.1, Seite 710.
Verwendet werden `regex`, `regex_match()`.
- Erkennung einer IP-Adresse: Abschnitt 24.12.2, Seite 712.
Verwendet werden `regex`, `regex_match()`.

13

Threads

Dieses Kapitel behandelt die folgenden Themen:

- Programmierung paralleler Abläufe mit Threads
- Synchronisation
- Thread-Steuerung: pausieren, fortsetzen, beenden
- Interrupt
- Warten auf Ereignisse/Producer-Consumer-Problem
- Monitor-Konzept
- Gleichzeitige lesende Zugriffe zulassen/Reader-Writer-Problem
- Thread-Sicherheit

Im Betriebssystem laufen verschiedene parallele Programme, auch Prozesse genannt, die jeweils einen eigenen Adressbereich haben. Zum Beispiel können ein Office-Programm und der Internet-Browser nebeneinander laufen. Die zeitliche Ausführung eines Programms kann mit einem Faden (englisch *thread*) symbolisiert werden. Dieser Faden beginnt mit dem Aufruf von `main()` und endet mit dem Ablauf von `main()`. Innerhalb eines solchen Programms, das selbst einen Thread darstellt, kann ein weiterer Thread gestartet werden. Das bedeutet, dass ab diesem Zeitpunkt ein weiterer Faden der Programmausführung neben dem erzeugenden Faden herläuft. Im Unterschied zu Prozessen teilen sich Threads einen Adressraum und heißen deswegen auch leichtgewichtige Prozesse. Zu einem Prozess gehörende Threads können also auf dieselben Variablen zugreifen.

Wenn es gleichviele oder mehr CPUs bzw. CPU-Kerne als parallele Prozesse/Threads gibt, können die Prozesse/Threads echt parallel abgearbeitet werden. Andernfall spricht man von *quasi-parallel*. Diese nur scheinbare Parallelität wird bei wenigen oder nur einer CPU erzeugt, indem in kurzen Zeitabständen jeder Thread mal zur Ausführung kommt.

Das Betriebssystem übernimmt die Aufteilung der Prozesse auf die CPUs. Dieser Vorgang heißt Einplanung (englisch *scheduling*).

Der Laufzeitaufwand zur Verwaltung von Threads ist geringer als der für Prozesse. Wenn ein Prozess der CPU zur Ausführung zugeteilt wird, muss vom Adressraum des vorhergehenden auf den des nun auszuführenden Prozesses umgeschaltet werden (Context-Switch). Weil Threads denselben Adressraum nutzen, ist dieser Vorgang viel weniger aufwendig. Threads sind immer dann sinnvoll, wenn verschiedene Dinge gleichzeitig getan werden sollen, zum Beispiel Bearbeiten von Dialogeingaben und gleichzeitig Laden eines Bildes. Damit wird verhindert, dass der längliche Vorgang des Bildladens die grafische Benutzungsschnittstelle blockiert. Threads werden von den Programmiersprachen auf verschiedene Art realisiert. Das Zustandsdiagramm 13.1 zeigt die wesentlichen Thread-Zustände. Die Zustände sind im Einzelnen:

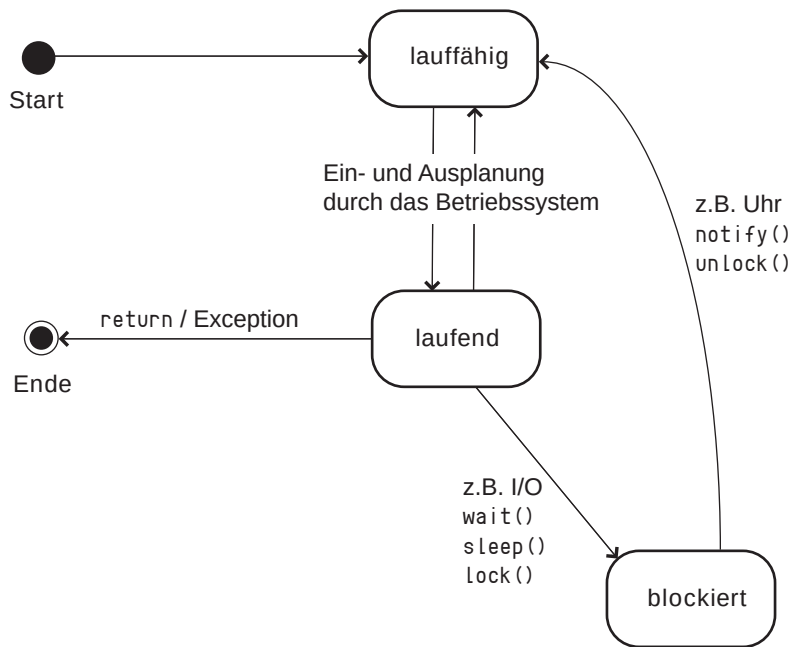


Abbildung 13.1: Thread-Zustände

- **Start:** Dies ist der Zustand nach Ablauf des Konstruktors des zugeordneten Thread-Objekts. In Java muss ein Thread erst mit dem Befehl `start()` gestartet werden, in C++ wird dies schon durch den genannten Konstruktor erledigt. Sofort danach beginnt der Thread anscheinend zu laufen. Tatsächlich wird er als *lauffähig* betrachtet und konkurriert mit anderen Threads und Prozessen um die Ressource CPU. Welcher Thread wann wie viel CPU-Zeit zugeteilt bekommt, bestimmt das Betriebssystem.
- **lauffähig:** In diesem Zustand kann dem Thread vom Betriebssystem CPU-Zeit zugeteilt werden.

- *laufend*: In diesem Zustand erfüllt der Thread seine Aufgabe. Da es noch viele andere Threads geben kann, wird er vom Betriebssystem immer mal wieder in den Zustand *lauffähig* versetzt, damit ein anderer Thread die CPU nutzen kann.
- *blockiert*: Ein Thread geht in diesen Zustand, wenn er warten muss. Gründe dafür können sein: Warten auf Abschluss des Datentransfers von oder zu einer Festplatte, mit `wait()` erzwungenes Warten auf eine Ressource, mit `sleep()` erzwungenes Warten für eine bestimmte Zeitdauer oder bis zu einem bestimmten Zeitpunkt.
- *Ende*: Der vom Thread zu erledigende Funktionsaufruf ist beendet.

Das folgende Beispiel zeigt drei Objekte der Klasse `thread`, denen dieselbe Funktion `f(int t)` zur Ausführung übergeben wird. Die Funktion tut nichts außer `t` Sekunden zu warten und dann die Beendigung anzuzeigen.



Hinweis

Die in [ISOC++] definierte Klasse `thread` ist zurzeit der Drucklegung dieses Buchs noch nicht hinreichend in manchen C++-Compilern verfügbar bzw. lauffähig. Aus diesem Grund wird nachfolgend die Boost Thread-Library verwendet. Die Boost-Libraries sind in vielen Fällen Vorbild für Entwicklungen des C++-Standards und kommen deshalb [ISOC++] recht nahe. Auch Qt, ein Produkt zur GUI-Entwicklung, bietet Threads (siehe Seite 469).

Listing 13.1: Thread-Beispiel

```
// cppbuch/k13/erstesBeispiel.cpp
#include <boost/thread.hpp>
#include <iostream>
using namespace std;

void f(int t) {
    // sleep: t Sekunden warten
    boost::this_thread::sleep(boost::posix_time::seconds(t));
    cout << "Thread " << boost::this_thread::get_id()
         << " : Funktion beendet! Laufzeit = " << t << " s" << endl;
}

int main() {
    boost::thread t1(f, 4);
    boost::thread t2(f, 6);
    boost::thread t3(f, 2);
    cout << "t1.get_id(): " << t1.get_id() << endl;
    cout << "t2.get_id(): " << t2.get_id() << endl;
    cout << "t3.get_id(): " << t3.get_id() << endl;
    t1.join(); // warten auf Beendigung
    t2.join();
    t3.join();
    cout << "t1.get_id(): " << t1.get_id() << endl;
    cout << "main() ist beendet" << endl;
}
```

Die Ausgabe des Programms ist

```
t1.get_id(): 0x8051008
t2.get_id(): 0x8051180
t3.get_id(): 0x80512f8
Thread 0x80512f8 : Funktion beendet! Laufzeit = 2 s
Thread 0x8051008 : Funktion beendet! Laufzeit = 4 s
Thread 0x8051180 : Funktion beendet! Laufzeit = 6 s
t1.get_id(): {Not-any-thread}
main() ist beendet
```

Was geschieht im Einzelnen?

- Zuerst werden drei Thread-Objekte erzeugt. Dem Konstruktor werden als Erstes die auszuführende Funktion (oder ein Funktionsobjekt) übergeben und als Nächstes alle Parameter, die diese Funktion benötigt. Daran kann man erkennen, dass der Thread-Konstruktor ein Template mit variabler Parameterzahl ist, wie in Abschnitt 6.5 beschrieben.
- Für jedes thread-Objekt wird sofort ein Thread gestartet, indem die übergebene Funktion ausgeführt wird. Im Fall eines übergebenen Funktionsobjekts wird dessen `operator()()` ausgeführt. Es ist zwischen den Threads selbst und den thread-Objekten, durch deren Konstruktor die Threads gestartet werden, zu unterscheiden. Der Haupt-Thread (das `main()`-Programm) läuft nach dem Start der Threads weiter, sodass es vier parallele Ausführungsfäden gibt. Jedem thread-Objekt ist eine Kennung zugeordnet, die von `main()` ausgegeben wird, während die anderen Threads beschäftigt sind.
- Anschließend wartet `main()` mit `t1.join()` darauf, dass sich der `t1` zugeordnete Thread beendet.
- Währenddessen meldet sich der Thread mit der kürzesten Laufzeit, dass er sich beendet hat. Innerhalb der Funktion ist das Thread-Objekt, das diese Funktion ausführt, nicht bekannt. Mit `this_thread::get_id()` kann aber die Kennung des thread-Objekts, das dem *aktuell* laufenden Thread zugeordnet ist, abgefragt werden – und das ist eben der, der gerade dabei ist, diese Funktion auszuführen. Das thread-Objekt existiert weiter und ist noch dem Thread zugeordnet, auch wenn der sich beendet hat.
- Der Thread mit der Laufzeit 4 s meldet seine Beendigung. Das ist aber gerade der Thread, auf den `main()` mit `t1.join()` wartet. Das Wort »join« bedeutet zusammenführen. Die Threads laufen ab diesem Punkt nicht mehr nebeneinander her, das heißt, der `t1` zugeordnete Thread hat aufgehört, als eigener Thread zu existieren. Das Objekt `t1` geht in den Zustand *repräsentiert keinen Thread* (mehr dazu auf Seite 424).
- Anschließend wartet `main()` mit `t2.join()` auf den zugeordneten Thread. Da dieser bereits fertig ist, gibt es keine Wartezeit.
- Anschließend wartet `main()` mit `t3.join()` auf den zugeordneten Thread. Dieser ist nach weiteren 2 s beendet, wie die Bildschirmausgabe zeigt, und `t3.join()` wird wirksam. Ab diesem Zeitpunkt gibt es nur noch den `main()`-Thread.
- Anschließend wird nur für `t1`, weil es für `t2` und `t3` dasselbe ergäbe, die Kennung ausgegeben. »Not-any-thread« besagt, dass `t1` zwar noch als Objekt existiert, aber keinen Thread mehr repräsentiert.

**Tipp**

Der Haupt-Thread `main()` sollte stets mit `join()` auf die von ihm angelegten Threads warten. Andernfalls werden diese Threads sofort bei Erreichen des Endes von `main()` zwangsweise beendet!

Das heißt, dass die Threads ihre Aufgabe nicht mehr zu Ende führen können und auch, dass darauf geachtet werden muss, dass jeder Thread wirklich terminiert, damit `main()` nicht hängen bleibt.

13.1 Die Klasse thread

Nach [ISOC++] ist die Klasse `thread` wie folgt deklariert (Auszug):

Listing 13.2: thread API nach [ISOC++] (kommentierter Auszug)

```
class thread {
public:
    class id; // Vorwärtsdeklaration für die Identifier-Klasse
              // Ausgabe eines id-Objekts ergibt dessen Kennung (s.o.)
    thread(); // Konstruktor für einen Thread, der sich von Anfang an im Zustand
              // repräsentiert keinen Thread befindet.
    template <class F, class ...Args> explicit
    thread(F&& f, Args&&... args); // Konstruktor für eine Funktion oder einen Funktor.
    ~thread(); // Destruktor; darf nicht erreicht werden, wenn der Thread noch
              // joinable() ist.
    void swap(thread&);
    bool joinable() const; // gibt (Zustand ≠ repräsentiert keinen Thread) zurück.
    void join(); // kehrt erst nach Ende der abzuarbeitenden Funktion zurück und
                // setzt den Zustand auf repräsentiert keinen Thread.
    void detach(); // löst die Verbindung zum Thread, setzt den Zustand auf repräsentiert
                  // keinen Thread und kehrt sofort zurück (Beispiel siehe unten).
    id get_id() const; // gibt Identitäts-Objekt des Threads zurück bzw. id(),
                      // falls der Thread im Zustand repräsentiert keinen Thread ist.
    thread(const thread&) = delete; // Kopierkonstruktor verbieten
    thread& operator=(const thread&) = delete; // Zuweisung verbieten
};
```

Die ausführliche Erklärung für `&&` lesen Sie erst in Abschnitt 22.2. Gemeint ist, dass Links-Werte in den Konstruktor hineinkopiert werden (also normale Übergabe per Wert), dass aber temporäre Objekte hinein-*bewegt* werden (Wegoptimieren des Kopiervorgangs). Für den Ablauf im Thread macht es keinen Unterschied. Es wird intern mit einer Kopie der Parameter gearbeitet, egal wie optimiert die Inhalte der Parameter zustande gekommen sind. Die Semantik ist genau so, als ob dort

```
template <class F, class ...Args> thread(F f, Args... args);
```

stünde. Mit `delete` wird dem Compiler mitgeteilt, dass er auf die automatische Erzeugung verzichten soll. Ein weiteres wichtiges API ist im Namespace `this_thread`. Dieser Namespace enthält Funktionen, die sich auf den aktuell laufenden Thread beziehen:

Listing 13.3: `this_thread`

```
namespace this_thread {
    thread::id get_id(); // gibt das Id-Objekt des aktuellen Threads zurück,
                        // Verwendung im Beispiel oben.
    void yield();        // Mitteilung an das Betriebssystem, dass dieser
                        // Thread nicht eilig ist und andere vorgezogen werden können.
    // Thread für eine bestimmte Zeitdauer schlafen legen:
    template <class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);
    // Thread bis zu einem bestimmten Zeitpunkt schlafen legen:
    template <class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
}
```

`<chrono>` ist in Abschnitt 27.7 ab Seite 760 beschrieben. Das in den Beispielen benutzte Boost-API für `sleep()` (statt `sleep_for` und `sleep_until` gemäß [ISOC++]) ist:

- Thread für eine bestimmte Zeitdauer schlafen legen:

```
// Namespace boost::this_thread-Funktion:
template<typename TimeDuration>
void sleep(TimeDuration const& rel_time);
```

Ein direkt umsetzbares Anwendungsbeispiel:

```
boost::this_thread::sleep(boost::posix_time::seconds(anzahl));
```

`seconds` kann je nach Anwendungszweck durch `hours`, `minutes`, `millisec` und `nanosec` ersetzt werden. Auch Kombinationen sind möglich, etwa

```
using namespace boost::posix_time;
time_duration dauer = hours(5) + minutes(3) + seconds(59);
boost::this_thread::sleep(dauer);
```

- Thread bis zu einem bestimmten Zeitpunkt schlafen legen:

```
// static Elementfunktion der Klasse thread:
void sleep(system_time const& abs_time);
```

Ein Beispiel, um fünf Minuten zu warten, bis das Ei weich gekocht ist (Auszug):

```
boost::system_time jetzt = boost::get_system_time();
boost::thread::sleep(jetzt + boost::posix_time::minutes(5));
```

Zustände von thread-Objekten

`thread`-Objekte unterscheiden sich von den zugeordneten Threads. Die Zustände von `thread`-Objekten und Threads sind daher ganz verschieden. `thread`-Objekte können sich in zwei Zuständen befinden:

- *repräsentiert einen Thread*: In diesem Zustand ist das Thread-Objekt einem Thread, der von ihm gestartet wurde, zugeordnet. Die Methode `get_id()` gibt das zugehörige Identitäts-Objekt zurück.

- *repräsentiert keinen Thread*: Dieser Zustand bedeutet, dass das Thread-Objekt keinem Thread (mehr) zugeordnet ist. Die Methode `get_id()` gibt ein vom Standardkonstruktor der Klasse `id` erzeugtes Objekt zurück. Das heißt aber nicht unbedingt, dass der Thread als Ausführungsfaden nicht mehr existiert, sondern nur, dass er durch das Thread-Objekt nicht mehr repräsentiert wird – das ist nicht dasselbe! Der Zustand wird durch den Aufruf einer der Methoden `join()` oder `detach()` erreicht. Das Verb »to detach« bedeutet »(Verbindung) lösen«.
- `t.join()` blockiert den Aufrufer, bis der Thread `t` beendet ist. Nach dem Aufruf ist `t` im Zustand *repräsentiert keinen Thread*, der Aufrufer setzt seinen Ablauf fort.
- `t.detach()` blockiert den Aufrufer *nicht*: Der Aufruf kommt sofort zurück mit dem Ergebnis, dass `t` im Zustand *repräsentiert keinen Thread* ist. Der ehemals zugeordnete Thread läuft währenddessen weiter, bis er beendet ist. Das folgende kleine Programm zeigt den Effekt:

Listing 13.4: `detach()`

```
// Auszug aus cppbuch/k13/detach.cpp
// Rest wie oben, siehe cppbuch/k13/erstesBeispiel.cpp
int main() {
    boost::thread t1(f, 4);
    boost::thread t2(f, 6);
    boost::thread t3(f, 2);
    cout << "t1.get_id(): " << t1.get_id() << endl;
    cout << "t2.get_id(): " << t2.get_id() << endl;
    cout << "t3.get_id(): " << t3.get_id() << endl;

    t1.detach();
    cout << "t1.detach(): " << t1.get_id() << endl;
    t3.detach();
    cout << "t3.detach(): " << t3.get_id() << endl;
    t2.join(); // warten auf Beendigung des längstdauernden Threads
    cout << "main() ist beendet" << endl;
}
```

Die Ausgabe des Programms ist:

```
t1.get_id(): 0x8051008
t2.get_id(): 0x8051180
t3.get_id(): 0x8051318
t1.detach(): {Not-any-thread}
t3.detach(): {Not-any-thread}
Thread 0x8051318 : Funktion beendet! Laufzeit = 2 s
Thread 0x8051008 : Funktion beendet! Laufzeit = 4 s
Thread 0x8051180 : Funktion beendet! Laufzeit = 6 s
main() ist beendet
```

13.2 Synchronisation

Programme, die »normalerweise« korrekt funktionieren, tun dies nicht unbedingt, wenn Threads im Spiel sind. Dazu ein kleines Beispiel: Mit der Klasse `Gerade` kann eine Folge gerader Zahlen erzeugt werden. Der Aufruf von `next()` liefert die jeweils nächste Zahl:

Listing 13.5: Klasse für gerade Zahlen – nicht threadsafe

```

1 class Gerade {
2 public:
3     Gerade()
4         : n(0) {
5     }
6     int next() {
7         ++n;
8         ++n;
9         return n;
10    }
11 private:
12     int n;
13 };

```

Eine Testfunktion könnte wie folgt aussehen:

```

Gerade g;

void testeGerade() {
    for(int i=0; i < 10000; ++i) {
        int wert = g.next();
        if(wert % 2 != 0) {
            cout << wert << " ist ungerade!"<< endl;
            break;
        }
    }
}

```

Im sequentiellen Fall ist garantiert, dass `next()` eine gerade Zahl liefert. Wenn jedoch zwei oder mehrere Threads auf dasselbe Objekt zugreifen, kann es Fehler geben. Wenn nur zwei Threads A und B existieren und `n` den Anfangswert 0 hat, kann es zu folgender Abfolge kommen:

1. Thread A führt `next()` bis Zeile 7 einschließlich aus und wird dann vom Betriebssystem in die Warteschlange wartender Threads gepackt, damit Thread B drankommt. Das Attribut `n` hat den Wert 1.
2. Thread B führt `next()` aus und erhöht `n` zweimal. Es wird 3 zurückgegeben – Fehler!
3. Thread A kommt wieder dran und führt `next()` zu Ende aus; es wird 4 zurückgegeben. Das Problem lässt sich leicht visualisieren, wenn man mehrere Threads auf das obige Objekt `g` loslässt. Am einfachsten lässt sich das durch eine Gruppe von Threads realisieren:

```

// Auszug aus cppbuch/k13/gerade.cpp
int main() {
    boost::thread_group threads; // mehr zu thread_group auf Seite 428

```

```

for (int i = 0; i < 20; ++i) {
    threads.create_thread(testeGerade);
}
threads.join_all();
}

```

Das Programm wird mal ungerade Zahlen ausgeben, mal auch nicht – abhängig von der Einplanung der Threads. Der Effekt, dass zufällig mal der eine und mal der andere Thread zuerst die Daten ändert, heißt *data race*. Eine Bedingung, die dazu führt, nennt man *race condition*. Die Ursache des Problems liegt darin, dass eine korrekte Abarbeitung der Funktion `next()` *atomar* (unteilbar) sein muss. Ein Thread sollte nur dann `next()` betreten dürfen, wenn jeder andere Thread die Funktion verlassen hat. Bereiche, in denen Daten verändert werden und die dem Zugriff mehrerer Threads ausgesetzt sind, wie hier die Zeilen 7 und 8 der Klasse, heißen *kritischer Bereich* (englisch *critical section*). Der Zugriff auf einen kritischen Bereich muss *synchronisiert* werden, indem sich die Zugriffe gegenseitig ausschließen (englisch *mutual exclusion*). Die Klasse `mutex` steuert den gegenseitigen Ausschluss. Der Ablauf wäre etwa wie folgt:

```

mutex mtx;

int next() {    // erster Ansatz
    mtx.lock(); // Zugriff für alle anderen verbieten
    // Anfang des kritischen Bereichs
    ++n;
    ++n;
    // Ende des kritischen Bereichs?
    mtx.unlock(); // Zugriff wieder freigeben. Immer noch problematisch, siehe Text
    return n;
}

```

Diese Idee hat zwei Schwächen:

- `mtx.unlock()` ist nicht die letzte Anweisung. Daher kann ein anderer Thread nach `mtx.unlock()`, aber vor `return`, `next()` ausführen und das Ergebnis verfälschen. Eine Lösung wäre natürlich, in der Funktion auf `mtx` zu verzichten und jeden Aufruf von `next()` einzuschachteln, etwa

```

mtx.lock();
int wert = g.next();
mtx.unlock();
// wert weiterverarbeiten

```

Diese richtige Lösung ist fehleranfällig, weil das Paar der `lock()/unlock()`-Aufrufe doch mal vergessen wird, und sie ist auch sehr umständlich.

- Es kann vorkommen, dass in einem kritischen Bereich eine Exception geworfen wird. Die Folge ist, dass `mtx.unlock()` nicht aufgerufen und nichts freigegeben wird. Jeder nachfolgende Thread bleibt danach bei `mtx.lock()` stehen – das Programm hängt.

Deshalb wählt man einen anderen, komfortableren und sichereren Weg, indem ein Lock-Objekt angelegt wird:

```

mutex mtx;
// ...

```



```
int next() {           // richtiger Ansatz:
    Lock lock(mtx);    // Lock-Objekt anlegen
    // Anfang des kritischen Bereichs
    ++n;
    ++n;
    return n;         // Ende des kritischen Bereichs
}
```

Der lock-Konstruktor ruft `mtx.lock()` und realisiert damit die Sperre. Der Destruktor ruft `mtx.unlock()`, aber erst beim Verlassen des Funktionsbereichs, also bei der schließenden Klammer des Funktionsblocks. Dieser Ansatz nennt sich englisch *scoped locking* oder *guard* ([SSRB]). Dahinter verbirgt sich nichts anderes als das Prinzip »Resource Acquisition Is Initialization« (siehe RAII im Glossar). C++ verwendet für den Zweck des *scoped locking* die Klassen `lock_guard` oder `unique_lock`. Letztere hat ein paar mehr Möglichkeiten wie etwa `lock()` und `unlock()`; Funktionen, die `lock_guard` nicht hat und die in diesem Kontext nicht gebraucht werden. Die obige Klasse `Gerade` wird wie folgt ergänzt, damit jeder Zugriff auf `next()` garantiert eine gerade Zahl liefert:

Listing 13.6: Klasse für gerade Zahlen – threadsafe

```
// Auszug aus cppbuch/k13/gerade.cpp
namespace {
    boost::mutex mtx;
}

class Gerade {
public:
    Gerade() : n(0) {
    }
    int next() {
        boost::lock_guard<boost::mutex> lock(mtx); // neu!
        ++n;
        ++n;
        return n;
    }
private:
    int n;
};
```

13.2.1 Thread-Group

Eine Thread-Group fasst mehrere Threads zusammen, um Funktionen für alle Threads der Gruppe mit nur einer Anweisung realisieren zu können. In [ISOC++] ist eine Thread-Group nicht vorgesehen, es gibt sie aber in der Boost Library. Die Schnittstelle:

Listing 13.7: Boost `thread_group`

```
class thread_group {
public:
    thread_group();           // Konstruktor
    ~thread_group();         // Destruktor
    template<typename F>
    thread* create_thread(F threadfunc); // Thread erzeugen und starten
```

```

void add_thread(thread* thrd);    // Thread hinzufügen
void remove_thread(thread* thrd); // Thread entfernen
void join_all();                 // join() für alle ausführen
void interrupt_all();            // interrupt() für alle ausführen
int size() const;                // gibt Anzahl der Threads zurück
};

```

Der Destruktor führt `delete` auf alle erzeugten und hinzugefügten Threads aus, woraus sich ergibt, dass die Threads von `create_thread()` mit `new` angelegt werden. Auch darf `add_thread()` nur ein Zeiger auf ein mit `new` erzeugtes Objekt übergeben werden, damit der Destruktor nicht ein Objekt zu löschen versucht, das auf dem Heap nicht existiert. Wie funktioniert eine Thread-Group? Am einfachsten ist es, sich einen Vektor vorzustellen, in dem Zeiger auf Threads gespeichert werden, etwa wie hier gezeigt:

Listing 13.8: Mögliche ThreadGroup-Implementierung

```

class ThreadGroup { // unvollständig
public:
    ~ThreadGroup() {
        for(size_t i = 0; i < pthreads.size(); ++i)
            delete pthreads[i];
    }
    template<typename T>
    boost::thread* create_thread(T t) {
        pthreads.push_back(new boost::thread(t)); // Thread speichern und starten
        return pthreads[pthreads.size()-1];
    }
    void join_all() {
        for(size_t i = 0; i < pthreads.size(); ++i)
            pthreads[i]->join();
    }
    // weitere Methoden
private:
    std::vector<boost::thread*> pthreads;
};

```

13.3 Thread-Steuerung: pausieren, fortsetzen, beenden

In diesem Abschnitt geht es darum, wie ein Thread kontrolliert angehalten, fortgesetzt und schließlich beendet werden kann. Dazu muss erst ein kleines Problem gelöst werden, weil die Klasse `thread` aus gutem Grund dafür keine Methoden hat. `thread` kann zum Beispiel nicht wissen, ob vor dem Pausieren bestimmte Arbeiten noch erledigt werden müssen. Ein anderes Problem besteht in der Art, wie die Übergabe des Funktionsobjekts in der Klasse `thread` vorgesehen ist. Sehen Sie sich das folgende Beispiel an:

```

class Funktor {
public:
    void operator()() {
        // eigentliche Arbeit erledigen
    }
    void pause() { ... }
    void fortsetzen() { ... }
};

int main() {
    Funktor f;
    thread t(f); // Ab hier wird operator()() ausgeführt.
    // ... warten
    f.pause();   // wirkt nicht!
}

```

Dass der Aufruf `f.pause()` nicht wirkt, liegt daran, dass `f` per Wert übergeben wird. Innerhalb des Threads wird `operator()()` für eine lokale Kopie von `f` ausgeführt. Diese lokale Kopie ist von `main()` aus nicht erreichbar. Deshalb gibt es eine Hüllklasse (englisch *wrapper*) `CallWrapper`, die eine Referenz auf den Funktor enthält. Darüber sind alle Methoden des Funktors erreichbar. Die Klasse, die die eigentliche Arbeit erledigt, heißt `Worker`. Zunächst stelle ich die Klasse `CallWrapper` und die Anwendung vor, damit Sie sich ein Bild vom Ablauf bilden können, der einen Arbeitstag simuliert. Eine Sekunde entspricht einer simulierten Stunde.

```

// Auszug aus cppbuch/k13/flags.cpp
template<class Callee> // Callee = Klasse des aufzurufenden Funktors
class CallWrapper {
public:
    CallWrapper(Callee& c) : callee(c) {}
    void operator()() {
        callee();           // = callee.operator()()
    }
private:
    Callee& callee;
};

```



Hinweis

C++ stellt die Klasse `reference_wrapper` zur Verfügung, die dasselbe wie die Klasse `CallWrapper` leistet (und noch ein wenig mehr), siehe Seite 761. Aus diesem Grund wird nur im direkt folgenden Beispiel zur Demonstration `CallWrapper` verwendet, danach ausschließlich die Funktion `ref()`, die ein `reference_wrapper`-Objekt zurückgibt.

Weil das `Worker`-Objekt bei Instanziierung des Templates mit der Klasse `Worker` per Referenz übergeben wird, bewirkt der Aufruf von `operator()()` stets, dass `operator()()` für das übergebene Originalobjekt `c` aufgerufen wird, auch wenn das `CallWrapper`-Objekt selbst eine Kopie ist – der Wert des Attributs (Referenz `callee`) bleibt bei einer Kopie erhalten. Weiter geht es mit `main()` und der Simulation eines Arbeitstages:

Listing 13.9: Simulation eines Arbeitstages

```
// Auszug aus cppbuch/k13/workerthread/main.cpp
int main() {
    Worker worker;
    CallWrapper aufrufer(worker); // Referenz auf Worker übergeben
    boost::thread t(aufrufer); // Thread anlegen und starten
    int stunde = 8;
    while(!worker.istBeendet()) {
        boost::this_thread::sleep(boost::posix_time::seconds(1));
        cout << ++stunde << " Uhr: ";
        switch(stunde) {
            case 10 : worker.warten(); // Pause
                       break;
            case 13 : worker.weiter(); // Fortsetzung
                       break;
            case 16 : worker.beenden(); // Ende
                       t.join(); // Ende abwarten
                       break;
            default: if(stunde > 16) {
                       throw "Fehler!";
                   }
        }
    }
}
```

Die Ausgabe des Programms ist:

```
Worker bei der Arbeit!
9 Uhr: Worker bei der Arbeit!
10 Uhr: Worker bei der Arbeit!
Worker pausiert!
11 Uhr: 12 Uhr: 13 Uhr: Worker macht weiter!
14 Uhr: Worker bei der Arbeit!
15 Uhr: Worker bei der Arbeit!
16 Uhr: Worker macht Feierabend!
```

Die Steuerung läuft über die Flags `pause` und `beendet` der Klasse `Worker`:

Listing 13.10: Klasse `Worker`

```
// Auszug aus cppbuch/k13/flags.cpp
class Worker {
public:
    Worker()
        : pause(false), beendet(false) {
    }
    void operator()() {
        while(!beendet) {
            cout << "Worker bei der Arbeit!" << endl;
            pauseOderNicht();
            boost::this_thread::sleep(boost::posix_time::seconds(1));
        }
        cout << "Worker macht Feierabend!" << endl;
    }
}
```

```

void warten() {
    boost::lock_guard<boost::mutex> lock(mtxPause);
    pause = true;
}
void weiter() {
    boost::lock_guard<boost::mutex> lock(mtxPause);
    pause = false;
    cond.notify_one();
}
void beenden() {
    beendet = true;
    cond.notify_one(); // ... falls im Zustand pausierend
}
bool istBeendet() const {
    return beendet;
}
private:
    bool pause;
    bool beendet;
    boost::mutex mtxPause;
    boost::condition_variable cond;

    void pauseOderNicht() {
        boost::mutex mtx;
        boost::unique_lock<boost::mutex> lock(mtx);
        while(pause && !beendet) {
            cout << "Worker pausiert!" << endl;
            cond.wait(lock);
            cout << "Worker macht weiter!" << endl;
        }
    }
};

```



Hinweis

Der lesende und schreibende Zugriff auf die Variable `beendet` ist nicht mit einem Lock synchronisiert, weil es nicht darauf ankommt: Die Änderung auf `true` ist irreversibel.

Im `main()`-Programm wird der Thread gestartet und jede Sekunde die Variable `stunde` um eins erhöht. Um 10 »Uhr« wird pausiert, veranlasst durch den Aufruf von `warten()`. Mit `weiter()` wird der Thread um 13 Uhr fortgesetzt und um 16 Uhr der simulierten Zeit beendet. Die Arbeit des Threads wird in der Methode `operator()()` geleistet, solange das Flag `beendet` nicht gesetzt ist. Entscheidend in der Methode `operator()()` ist der Aufruf der privaten Methode `pauseOderNicht()`. In dieser Methode wird gewartet, bis das Flag `pause`, gesetzt durch die Methode `warten()`, zurückgesetzt worden ist. Auf den ersten Blick sind zwei einfache Lösungen denkbar, die jedoch beide ihre Nachteile haben:

```

// ständige Abfrage: Tun Sie das nicht!
while(pause) {
}

```

Bei dieser Methode, englisch *busy waiting* genannt, wird sinnlos CPU-Zeit verbraten.

```
// periodische Abfrage: Tun Sie das nicht, wenn es eine Alternative mit wait gibt!  
while(pause) {  
    sleep(seconds(1));  
}
```

Die periodische Abfrage wird englisch »Polling« genannt. Der Nachteil besteht darin, dass die eingestellte Periode zwangsweise abläuft, wenn `pause` inmitten der Sekunde zurückgesetzt wird. Alternativ `seconds(1)` durch `millisec(10)` zu ersetzen, würde den genannten Nachteil reduzieren, wäre aber durch einen erheblichen Aufwand, der durch die vielen nutzlosen `sleep()`-Aufrufe entsteht, zu bezahlen. Die richtige Lösung ist etwas komplizierter, aber dafür kostet das Warten fast nichts, weil der `wait()`-Aufruf vom Betriebssystem unterstützt wird. Der Ablauf in `pause0derNicht()` ist im Einzelnen (`pause` ist gesetzt):

- `cond.wait(lock)` sorgt dafür, dass der Thread in eine Warteschlange des Betriebssystems versetzt wird und somit nicht mehr aktiv ist. `cond` ist eine Bedingungsvariable, die Attribut der Klasse ist. Bedingungsvariablen stehen die Funktionen `wait()` und `notify()` (in Varianten) zur Verfügung. Beim Aufruf von `wait()` wird das Lock-Objekt übergeben. Das Lock-Objekt ist zwar funktionslokal, jedoch nicht die Mutex-Variable, die übergeben wird.
- Um 13 Uhr simulierter Zeit wird in `main()`, also von einem anderen, aktiven Thread `worker.weiter()` aufgerufen. Diese Funktion setzt das Flag `pause` zurück und ruft `cond.notify_one()` auf. Der Aufruf von `notify_one()` bewirkt, dass genau ein Thread aus der Warteschlange befreit wird. In diesem Beispiel kann es auch nur einer sein. Die Funktion `wait()` kehrt zurück.
- Das Flag wird in der `while()`-Bedingung noch einmal überprüft, weil es inzwischen wieder gesetzt worden sein könnte. Wenn nicht, wird die Schleife kein zweites Mal ausgeführt, die Funktion `pause0derNicht()` kehrt zurück, und die Schleife in `operator()()` kann weiterlaufen.
- Um 16 Uhr simulierter Zeit wird in `main()` `worker.beenden()` aufgerufen. Diese Funktion setzt das Flag `beendet`, das in der Schleifenbedingung in `operator()()` abgefragt wird. Es ist jedoch möglich, dass `beenden()` gerade dann aufgerufen wird, wenn der Thread in `pause0derNicht()` wartet. Um den Thread wirklich zu Ende zu bringen, muss er in diesem Fall befreit werden; dies ist der Grund für die Anweisung `cond.notify_one()` in der Funktion `beenden()`. Falls ein Thread nicht in `wait()` hängt, ist der Aufruf von `notify_one()` ohne Wirkung.
- Das abschließende `t.join()` in `main()` sorgt dafür, dass `main()` wartet, bis sich der Thread wirklich beendet hat, weil der Vorgang des Beendens einige Zeit in Anspruch nehmen kann.

Statt `notify_one()` könnte auch `notify_all()` verwendet werden. `notify_all()` befreit *alle* der betreffenden Bedingung zugeordneten wartenden Threads. Von denen wählt das Betriebssystem einen aus, der zum Zuge kommen kann. Die anderen werden wieder in die Warteposition geschickt.



Mehr zum Unterschied der beiden Funktionen finden Sie auf Seite 576.

Einen Thread pro Objekt garantieren

In der obigen `main()`-Funktion wäre es möglich, dass zwei `thread`-Objekte dasselbe `Worker`-Objekt verwenden, was zu Fehlern führte. Wenn Sie garantieren wollen, dass pro `Worker`-Objekt genau ein Thread läuft, konstruieren Sie am besten eine Klasse ähnlich wie `Worker`, zum Beispiel mit Namen `WorkerThread`, die aber ein `Thread`-Objekt als Attribut hat. Nach Erzeugung eines `WorkerThread`-Objekts kann der Thread mit einer Methode `start()`, die ein `thread`-Objekt anlegt, gestartet werden:

```
// Auszug aus cppbuch/k13/workerthread/WorkerThread.h
void start() {
    derThread = boost::thread(std::ref(*this));
}
```

`derThread` ist ein Attribut, das anfangs mit dem `thread`-Standardkonstruktor initialisiert wird, also keinen Thread repräsentiert. In `start()` wird ihm der tatsächlich zu startende Thread zugewiesen. Die Methode `beenden()` sorgt für die `join()`-Operation, sodass sie nicht von außen aufgerufen werden muss:

```
void beenden() {
    beendet = true;
    cond.notify_one(); // ... falls im Zustand pausierend
    derThread.join();
}
```

Das vollständige Beispiel finden Sie im Verzeichnis `cppbuch/k13/workerthread`.

13.4 Interrupt



Hinweis

Einen Thread mit der Funktion `interrupt()` zu unterbrechen, ist in [ISOC++] nicht vorgesehen, im Gegensatz zu den Boost-C++-Threads. Um langfristig portabel zu bleiben, empfehle ich die daher die oben skizzierte Steuerung mit Flags. Dieser Abschnitt beschreibt das Interrupt-Konzept für Boost-C++-Threads (in Java wird es auf ähnliche Art realisiert). Abschnitt 13.5 zeigt, wie ein `wait()` ohne Interrupt unterbrochen werden kann.

Interrupt heißt auf Deutsch Unterbrechung, und der Name der Funktion `interrupt()` suggeriert, als würde von ihr ein Thread unterbrochen. Tatsächlich ist es jedoch nur eine Interrupt-Anforderung: Es wird ein internes Flag gesetzt, das von allen Varianten der `wait()`-, `join()`- und `sleep()`-Methoden ausgewertet wird. Der Interrupt-Mechanismus kann ganz abgeschaltet werden. Falls das nicht geschehen ist, werfen die genannten Methoden bei gesetztem Flag eine `boost::thread_interrupted`-Exception. Das heißt auch: Falls die Exception folgenlos abgefangen wird, läuft der Thread einfach weiter – ohne Unterbrechung. Zur Demonstration wird die obige Klasse `Worker` angepasst, indem un-

ter anderem das Attribut beendet gestrichen wird. Zur Abwechslung wird die Abfrage in `main()`, ob der Thread noch läuft, durch die Zustandsabfrage des `thread`-Objekts gelöst.

```
// Auszug aus cppbuch/k13/interr.cpp
int main() {
    Worker worker;
    boost::thread t(std::ref(worker)); // reference_wrapper
    int stunde = 8;
    boost::thread::id keinThread;      // neu
    while(t.get_id() != keinThread) {  // neu
        boost::this_thread::sleep(boost::posix_time::seconds(1));
        cout << ++stunde << " Uhr: ";
        switch(stunde) {
            case 10 : worker.warten();
                    break;
            case 13 : worker.weiter();
                    break;
            case 16: t.interrupt();      // neu
                    t.join();          // wartet und ändert Zustand
                    break;
            default: if(stunde > 16) {
                    throw "Fehler!";
                }
        }
    }
}
```

In der Klasse `Worker` entfallen die Methoden `beenden()` und `istBeendet()`. Die Methoden `operator()()` und `pauseOderNicht()` werden leicht modifiziert:

Listing 13.11: Interrupt-Auswertung

```
// Auszug aus cppbuch/k13/interr.cpp
void operator()() {
    try {
        while(true) { // Abbruch mit Interrupt
            cout << "Worker bei der Arbeit!" << endl;
            pauseOderNicht();
            boost::this_thread::sleep(boost::posix_time::seconds(1));
        }
    } catch(const boost::thread_interrupted& e) {
        ; // nichts machen, while-Schleife ist beendet
    }
    cout << "Worker macht Feierabend!" << endl;
}
```

Sofort nach Aufruf von `t.interrupt()` in `main()` wird eine `boost::thread_interrupted`-Exception geworfen. Damit wird die Schleife verlassen, und der Thread ist beendet. Für die Exception kann es zwei Quellen geben:

- Die Methode `sleep()` in der Funktion `operator()()` und
- Die Methode `wait()` in der Funktion `pauseOderNicht()`.

Im letzten Fall wird die während `wait()` geworfene Exception automatisch an den Aufrufer `operator()()` weitergegeben.


```
// Auszug aus cppbuch/k13/interr.cpp
void pause0derNicht() {
    boost::mutex mtx;
    boost::unique_lock<boost::mutex> lock(mtx);
    while(pause) {
        cout << "Worker pausiert!" << endl;
        cond.wait(lock);           // kann Exception werfen
        cout << "Worker macht weiter!" << endl;
    }
}
```

13.5 Warten auf Ereignisse

Das Producer-Consumer-Problem ist eins der sehr gut bekannten Probleme der Thread-Programmierung. Es geht um Folgendes: Ein Produzent produziert etwas, legt es ab, und ein Konsument holt sich das abgelegte Objekt zur Weiterverarbeitung. Die Kapazität der Ablage ist begrenzt. Die skizzierte Aufgabenstellung findet sich in jeder Fabrik oder Fabriksimulation. Eine Maschine bearbeitet ein Werkstück und legt es auf einen Stapel oder ein Fließband, von wo es einer anderen Maschine zur Weiterverarbeitung zugeführt wird. Diese andere Maschine wartet auf das Ereignis, dass das nächste Werkstück zur Verfügung steht.

Es geht dabei im Wesentlichen um die Abstimmung von Producer und Consumer: Der Erste kann nichts ablegen, wenn die Ablage voll ist, und der Letzte kann nichts entnehmen, wenn sie leer ist. Gekoppelte Systeme dieser Art zur Fabriksimulation sind interessant, weil mit ihnen die Durchlaufzeit vom Anfang bis zum fertigen Produkt berechnet werden kann. Störungen im Ablauf oder Verbesserungen durch Einführung paralleler Strecken lassen sich gut simulieren. Um das Problem zu vereinfachen, nehme ich im Folgenden an, dass die Ablage nur Platz für ein einziges Element hat. Ferner sollen die »Elemente« durch einfache `int`-Zahlen dargestellt werden, wie in der Abbildung 13.2 gezeigt.



Abbildung 13.2: Producer-Consumer-Problem

Im Programmbeispiel bekommt der Producer eine Nummer `id` zur Identifizierung. Nach dem Start »produziert« er 5 `int`-Zahlen in einem zeitlich zufälligen Abstand und legt sie im Ablage-Objekt ab, dessen Referenz dem Konstruktor übergeben wird. Der Producer kann im Fehlerfall bei `wait()` hängenbleiben, weil der Consumer ausgefallen ist. Deshalb kann das Warten mit einer Exception, von `put()` geworfen, beendet werden.

Listing 13.12: Producer

```
// cppbuch/k13/prodcons/Producer.h
#ifndef PRODUCER_H
#define PRODUCER_H
#include <iostream>
#include <Random.h>
#include "Ablage.h"

class Producer {
public:
    Producer(Ablage& a, int i)
        : ablage(a), id(i), zufall(500) {
    }
    void operator()() {
        for(int i = 0; i < 5; ++i) {
            int wert = id*10 + i;
            boost::this_thread::sleep(
                boost::posix_time::millisec(200 + zufall()));
            try {
                ablage.put(wert);           // Exception-Quelle
                boost::lock_guard<boost::mutex> lock(ausgabeMutex);
                std::cout << "Producer Nr. " << id
                    << " legt ab: " << wert << std::endl;
            } catch(...) {
                break; // Producer ist beendet
            }
        }
        boost::lock_guard<boost::mutex> lock(ausgabeMutex);
        std::cout << "Producer " << id << " beendet sich." << std::endl;
    }
private:
    Ablage& ablage;
    int id;
    Random zufall;
};
#endif
```

Falls die Ablage belegt ist, muss der Producer bei `put()` warten, bis sie frei ist. Die Ausgabeanweisungen sind nicht threadsicher; es kann zu einem *data race* kommen. `ausgabeMutex` sorgt deshalb dafür, dass sich die Nachrichten von Producer und Consumer auf dem Bildschirm nicht vermischen. `ausgabeMutex` ist in der unten gezeigten inkludierten Datei *Ablage.h* enthalten. Der Destruktor des Locks gibt die Mutex-Variable automatisch frei. Falls nach der Ausgabe noch langandauernde Tätigkeiten stattfinden (hier nicht der Fall), kann man die Freigabe durch Einschließen des Locks und der Ausgabe in einen Block beschleunigen:

```
{ // Block-Beginn
    boost::lock_guard<boost::mutex> lock(ausgabeMutex);
    std::cout << "Ausgabe" << std::endl;
} // Block-Ende: Freigabe von ausgabeMutex durch Destruktor
// ... weitere Anweisungen
```

Der Consumer holt sich einen Wert aus der Ablage. Ist sie leer, muss der Consumer bei `get()` warten. Der Producer beendet sich nach Erledigung seiner Aufgabe automatisch; der Consumer hingegen kann nicht wissen, wann er den letzten Wert verarbeitet hat, und wartet vielleicht vergeblich. Aus diesem Grund ist vorgesehen, dass das Warten mit einer Exception, von `get()` geworfen, beendet wird.

Listing 13.13: Consumer

```
// cppbuch/k13/prodcons/Consumer.h
#ifndef CONSUMER_H
#define CONSUMER_H
#include <iostream>
#include "Ablage.h"

class Consumer {
public:
    Consumer(Ablage& a)
        : ablage(a) {

    void operator()() {
        try {
            while(true) {
                // Abbruch mit Exception
                int wert = ablage.get(); // Exception-Quelle
                boost::lock_guard<boost::mutex> lock(ausgabeMutex);
                std::cout << "Consumer hat "
                    << wert << " geholt." << std::endl;
            }
        } catch(...) {
            boost::lock_guard<boost::mutex> lock(ausgabeMutex);
            std::cout << "Consumer beendet sich." << std::endl;
        }
    }
private:
    Ablage& ablage;
};
#endif
```

Die Synchronisation wird durch das Ablage-Objekt, das eine gemeinsam genutzte Ressource darstellt, vorgenommen. Im zugehörigen `main()`-Programm gibt es zwei Producer und einen Consumer:

Listing 13.14: Producer-Consumer-Beispiel

```
// cppbuch/k13/prodcons/main.cpp
#include <boost/thread.hpp>
#include "Ablage.h"
#include "Producer.h"
#include "Consumer.h"
using namespace std;

int main() {
    Ablage ablage;
    Producer p1(ablage, 1);
```

```

Producer p2(ablage, 2);
Consumer c(ablage);
// Threads starten
boost::thread tp1(p1);
boost::thread tp2(p2);
boost::thread tc(c);
// Ende der Producer abwarten
tp1.join();
tp2.join();
// warten, bis alles abgeholt ist
boost::this_thread::sleep(boost::posix_time::seconds(1));
ablage.beenden(); // wartende Prozesse beenden
tc.join();        // Ende des Consumers abwarten
}

```

Monitor-Konzept

Diese Klasse von Problemen zeichnet sich dadurch aus, dass viele Threads gleichzeitig auf Methoden eines Objekts, die kritische Bereiche lesen oder ändern, zugreifen können. Die Lösung ist eine Synchronisation, die es nur einem einzigen Thread zurzeit erlaubt, eine Methode aufzurufen. Man sagt, dass die Methoden *synchronisiert* sind. Ein Objekt, das diese Voraussetzung erfüllt, heißt *Monitor*-Objekt. »Monitor« ist ein aus der Betriebssystemprogrammierung stammender Begriff. Diese Voraussetzung kann leicht erfüllt werden, wenn es ein spezielles Mutex-Attribut gibt, für das am Eingang jeder kritischen Methode ein Lock-Objekt angelegt wird, wie die Klasse *Ablage* zeigt:

Listing 13.15: Gemeinsame Ressource: *Ablage*

```

// cppbuch/k13/prodcons/Ablage.h
#ifndef ABLAGE_H
#define ABLAGE_H
#include <boost/thread.hpp>
#include <iostream>

namespace {
    boost::mutex ausgabeMutex;
}

class Ablage {
public:
    Ablage()
        : belegt(false), beendet(false) {
    }
    int get() {
        boost::unique_lock<boost::mutex> lock(monиторMutex);
        while(!belegt) {
            cond.wait(lock);
            if(beendet) {
                throw "get()-wait beendet";
            }
        }
    }
    belegt = false;

```

```

        cond.notify_one();
        return inhalt;
    }
    void put(int wert) {
        boost::unique_lock<boost::mutex> lock(monitorMutex);
        while(belegt) {
            cond.wait(lock);
            if(beendet) {
                throw "put()-wait beendet";
            }
        }
        belegt = true;
        inhalt = wert;
        cond.notify_one();
    }
    void beenden() { // Alle wait()-Aufrufe zwangsläufig beenden
        beendet = true;
        cond.notify_all();
    }
private:
    bool belegt;
    bool beendet;
    int inhalt;
    boost::mutex monitorMutex;
    boost::condition_variable cond;
};
#endif

```

Auf den ersten Blick scheint es, dass ein mit `get()` wartender Consumer das Lock blockiert, sodass auch kein Producer Zugriff hat – schließlich beziehen sie sich beide auf dieselbe Mutex-Variable. Tatsächlich ist es nicht so, weil `wait()` eine wichtige Eigenschaft hat:

- Beim Aufruf von `wait()` wird implizit `monitorMutex.unlock()` aufgerufen, also die Sperre *gelöst*! Dann geht der Thread in den Zustand »wartend«.
- Nach einem `notify()` wird implizit `monitorMutex.lock()` aufgerufen, also das Lock angefordert. Erst wenn dies gelingt, also kein anderer Thread das Lock besitzt, kehrt `wait()` zurück.

Beide Vorgänge laufen atomar ab. Damit wird erreicht, dass ein wartender Thread das Objekt nicht gänzlich blockiert, und dass *nach* `wait()` kein anderer Thread im Objekt aktiv ist.

wait() unterbrechen

Wie gesagt, kann der Consumer nicht wissen, wann seine Aufgabe beendet ist, weswegen eine Exception zum Anhalten vorgesehen ist. Oder der Producer bleibt aufgrund eines Fehlers im Consumer hängen. Deswegen ist eine Möglichkeit, `wait()` unterbrechen zu können, sehr sinnvoll. In Abschnitt 13.4 wird dies mit dem Aufruf von `interrupt()` für einen Thread demonstriert. Ohne Interrupt ist dies genauso möglich, wenn es ein Flag `beendet` gibt, das von der Methode `beenden()` gesetzt wird. Anschließend muss sie die wartenden Threads freigeben, wie oben in der Klasse `Ablage` zu sehen.

Nach der Rückkehr aus `wait()` wird das Flag abgefragt, und es wird, falls gesetzt, eine Exception geworfen, die der Aufrufer der Funktion auffangen kann. Verlassen der Funktion mit einer Exception bewirkt den Aufruf der Destruktoren aller lokal angelegten Objekte, in diesem Fall nur der Variablen `lock`. Der Destruktor sorgt für die Freigabe der Mutex-Variablen, sodass andere wartende Threads befreit werden, ihrerseits `wait()` verlassen und sich auf dieselbe Art beenden.



Übung

13.1 Erweitern Sie das Producer-Consumer-Beispiel:

1. Die Ablage soll nicht nur einen, sondern mehrere Plätze haben. Diese sollen intern durch ein `int`-Array implementiert werden. Die Anzahl der Plätze wird dem Konstruktor mitgegeben.
2. Zuerst abgelegte Objekte sollen auch zuerst wieder entnommen werden (FIFO).
3. Es sollen mehrere Producer und mehrere Consumer gleichzeitig arbeiten können. Hinweis: Statt das Attribut `belegt` auszuwerten, ist es sinnvoll, den Zustand `voll` bzw. `leer` abzufragen.

13.6 Reader/Writer-Problem

Im vorhergehenden Abschnitt haben Sie gesehen, wie der gleichzeitige Zugriff auf eine Ressource gegenseitig verriegelt wird. Das ist manchmal nicht erwünscht. Eine Datei oder eine Datenbank (die Ressource) soll von vielen gleichzeitig gelesen werden können. Nur wenn eine Änderung notwendig wird, müssen zur Sicherung der Integrität der gelesenen Informationen alle Lesezugriffe gesperrt werden, bis die Änderung vollzogen ist. Der gegenseitige Ausschluss (englisch *mutual exclusion*) ist dafür zu einschränkend und kann die Performanz stark beeinträchtigen.

- Beliebig viele *Reader*, aber kein einziger *Writer* dürfen gleichzeitig aktiv sein. Ein geteilter Zugriff (englisch *shared access*) muss für Reader erlaubt sein.
- Wenn ein *Writer* aktiv ist, darf zur selben Zeit kein weiterer Writer und auch kein Reader aktiv sein. Ein Writer benötigt exklusiven Zugriff.

Solange die Integrität der Daten jedem lesenden Thread garantiert wird, also während des Lesens kein anderer Thread die Daten modifiziert, ist alles in Ordnung. Beliebig viele Reader oder ein einziger Writer können auf die Daten zugreifen, aber nie zur selben Zeit. Das wird durch ein Read-Write-Lock bewerkstelligt, in der Boost Library durch ein `shared_mutex` gesteuert. Zusammengefasst:

```
shared_mutex rwMutex;
shared_lock<shared_mutex> lock(rwMutex); // Lock für geteilten Zugriff
unique_lock<shared_mutex> lock(rwMutex); // Lock für exklusiven Zugriff
```

In der folgenden Klasse `Ressource` wird die Read-Write-Steuerung auf diese Weise realisiert. Das Attribut `inhalt` ist ein schlichter String, der gelesen bzw. geschrieben wird. Der Schwerpunkt des Beispiels liegt auf der Dokumentation der Zugriffe, damit erkenn-

bar wird, wie viele Reader bzw. Writer gerade aktiv sind. Um wirklich zu erreichen, dass mehrere Reader gleichzeitig aktiv sind, wird der lesende Zugriff zeitlich verzögert. Auch der Writer wird gebremst.

Listing 13.16: Ressource

```
// cppbuch/k13/rw/Ressource.h
#ifndef RESSOURCE_H
#define RESSOURCE_H
#include <string>
#include <boost/thread.hpp>
#include <boost/lexical_cast.hpp>
#include <iostream>
#include <Random.h>

namespace {
    boost::mutex ausgabeMutex;
    Random derZufall(500);
}

class Ressource {
public:
    Ressource()
        : inhalt("nichts"), nreader(0), nwriter(0) {

    std::string read(const std::string& id) {
        // Lock für geteilten Zugriff:
        boost::shared_lock<boost::shared_mutex> lock(rwMutex);
        ++nreader;
        println(id + " liest " + inhalt
                + rwprotokoll()); // siehe private Methode unten
        // lesen dauert ... (Simulation)
        boost::this_thread::sleep(
            boost::posix_time::millisec(2000+ derZufall()));
        --nreader;
        return inhalt;
    }

    void write(const std::string& neu, const std::string& id) {
        // Lock für exklusiven Zugriff:
        boost::unique_lock<boost::shared_mutex> lock(rwMutex);
        ++nwriter; // mögliche Werte 0 oder 1
        println(id + " schreibt " + neu + rwprotokoll());
        // schreiben dauert auch ... (Simulation)
        boost::this_thread::sleep(
            boost::posix_time::millisec(1000+ derZufall()));
        --nwriter;
        inhalt = neu;
    }

    static void println(const std::string& was) {
        boost::lock_guard<boost::mutex> lock(ausgabeMutex);
```

```

        std::cout << was << std::endl;
    }
private:
    std::string rwprotokoll() {
        std::string msg(" Anzahl Aktiver: R=");
        // lexical_cast wandelt die Zahl in einen string um, siehe Seite 629
        msg += boost::lexical_cast<std::string>(nreader) + " W="
            + boost::lexical_cast<std::string>(nwriter);
        return msg;
    }
    std::string inhalt;
    boost::shared_mutex rwMutex;
    int nreader;
    int nwriter;
};

#endif

```

Im `main()`-Programm werden mehrere Reader und zwei Writer angelegt, die alle um die Ressource konkurrieren:

Listing 13.17: Reader-Writer-Beispiel

```

// cppbuch/k13/rw/main.cpp
#include<boost/thread.hpp>
#include<functional> // ref()
#include"Ressource.h"
#include"Writer.h"
#include"Reader.h"

using namespace std;

int main() {
    Ressource ressource;
    Writer w1(ressource, "w1");
    Writer w2(ressource, "w2");
    Reader r1(ressource, "r1");
    Reader r2(ressource, "r2");
    Reader r3(ressource, "r3");
    Reader r4(ressource, "r4");

    boost::thread_group threads;
    threads.create_thread(std::ref(w1));
    threads.create_thread(std::ref(r1));
    threads.create_thread(std::ref(r2));
    threads.create_thread(std::ref(r3));
    threads.create_thread(std::ref(r4));
    boost::this_thread::sleep(boost::posix_time::seconds(1));
    threads.create_thread(std::ref(w2));

    // Threads eine Zeit lang laufen lassen
    boost::this_thread::sleep(boost::posix_time::seconds(30));
    w1.beenden();
    w2.beenden();
}

```



```

    r1.beenden();
    r2.beenden();
    r3.beenden();
    r4.beenden();
    threads.join_all(); // warten, bis alles beendet ist
}

```

Jetzt fehlen nur noch die Reader und Writer. Weil sie einiges gemeinsam haben, werden die gemeinsamen Methoden und Attribute in eine abstrakte Oberklasse `ReaderWriter` verlagert:

Listing 13.18: Oberklasse `ReaderWriter`

```

// cppbuch/k13/rw/ReaderWriter.h
#ifndef READERWRITER_H
#define READERWRITER_H
#include "Ressource.h"

class ReaderWriter {
public:
    virtual void operator()() = 0; // vom Thread aufgerufene Methode
    void beenden() {
        ende = true;
    }
protected:
    ReaderWriter(Ressource& r, const std::string& i)
        : ende(false), ressource(r), id(i) {
    }
    virtual ~ReaderWriter() {}
    bool ende;
    Ressource& ressource;
    std::string id;
};
#endif

```

Weil die Klassen `Reader` und `Writer` von `ReaderWriter` erben, gestalten sich die zugehörigen Dateien recht kurz. Außer dem Konstruktor muss nur die Methode `operator()()` definiert werden.

Listing 13.19: Klasse `Reader`

```

// cppbuch/k13/rw/Reader.h
#ifndef READER_H
#define READER_H
#include "ReaderWriter.h"

class Reader : public ReaderWriter {
public:
    Reader(Ressource& r, const std::string& i)
        : ReaderWriter(r, std::string("Reader ") + i) {
    }
    void operator()() {
        Random zufall(1000);
        while(!ende) {

```

```

        std::string inhalt = ressource.read(id); // hier nicht weiter verwendet,
                                                // es geht nur um den reinen Lesevorgang.
        // Den nächsten Lesevorgang zufällig verzögern:
        boost::this_thread::sleep(boost::posix_time::millisec(zufall()));
    }
    Ressource::println(id + " beendet sich.");
}
};
#endif

```

Listing 13.20: Klasse Writer

```

// cppbuch/k13/rw/Writer.h
#ifndef WRITER_H
#define WRITER_H
#include "ReaderWriter.h"

class Writer : public ReaderWriter {
public:
    Writer(Ressource& r, const std::string& i)
        : ReaderWriter(r, std::string("Writer ") + i) {
    }
    void operator()() {
        int nr = 0;
        Random zufall(200);
        while(!ende) {
            std::string nachricht("Nachricht Nr.");
            // lexical_cast wandelt die Zahl in einen string um, siehe Seite 629
            nachricht += boost::lexical_cast<std::string>(++nr);
            ressource.write(nachricht, id);
            // Den nächsten Schreibvorgang zufällig verzögern:
            boost::this_thread::sleep(boost::posix_time::millisec(zufall()));
        }
        Ressource::println(id + " beendet sich.");
    }
};
#endif

```

Am besten lassen Sie das Programm mehrmals laufen. Durch das eingebaute Zufallselement, aber auch, weil dem Laufzeitsystem keine Vorgabe gegeben werden kann, wann welche Threads laufen, ist das Ergebnis variabel. Ein möglicher Auszug:

```

Reader r1 liest nichts Anzahl Aktiver: R=2 W=0
Writer w1 schreibt Nachricht Nr.1 Anzahl Aktiver: R=0 W=1
Reader r4 liest Nachricht Nr.1 Anzahl Aktiver: R=1 W=0
Reader r3 liest Nachricht Nr.1 Anzahl Aktiver: R=2 W=0
Reader r2 liest Nachricht Nr.1 Anzahl Aktiver: R=4 W=0
Reader r1 liest Nachricht Nr.1 Anzahl Aktiver: R=4 W=0
Writer w2 schreibt Nachricht Nr.1 Anzahl Aktiver: R=0 W=1

```

13.6.1 Wenn Threads verhungern

Da keine Annahmen getroffen werden können, welcher Thread wann läuft, können die folgenden Situationen entstehen:

- Es kommen ständig neue Reader hinzu, auch wenn andere den Lesevorgang beendet haben, sodass die Anzahl stets größer als 0 ist. In diesem Fall hat kein Writer eine Chance, die Daten zu aktualisieren – man sagt, der Writer-Thread verhungert.
- Ein oder mehrere Writer haben jeweils nacheinander exklusiven Zugriff, sodass kein Reader eine Chance zum Lesen der Daten bekommt. Die Reader »verhungern«.

Es gibt also kein Fair Play in dem obigen Algorithmus. Reader und Writer konkurrieren gnadenlos. Daraus ergibt sich die Frage: Wenn sowohl Reader als auch ein Writer gleichzeitigen Zugriff begehren, wer sollte bevorzugt werden?

- Falls Reader bevorzugt werden, können gleichzeitig viele lesende Threads laufen – die Parallelität und damit die Ausnutzung der Ressource steigt. Die Gefahr: Ein Writer könnte verhungern.
- Falls ein Writer bevorzugt wird, wird die Parallelität wegen des exklusiven Zugriffs reduziert. Reader könnten verhungern, obwohl die Wahrscheinlichkeit dafür in der Praxis klein ist. Meistens gibt es viele Reader und nur wenige oder einen Writer. Ein Vorteil der Bevorzugung eines Writers ist, dass die Daten schneller aktualisiert werden.
- Wenn die Aktualität der Daten hohe Priorität hat, kann man den Zutritt neuer Reader blockieren, sobald ein Writer Zugriff auf die Ressource verlangt, sodass der Writer sofort nach Ende des letzten Lesevorgangs die Daten aktualisieren kann.

Allerdings können den Threads keine Prioritäten zugeordnet werden, man muss also bei Bedarf eine andere Lösung wählen. Um den dritten Fall zu realisieren, kann man den Writern ermöglichen, sich anzumelden. Sobald mindestens ein Writer angemeldet ist, müssen neue Reader warten. Dazu braucht es neue Attribute in der Klasse `Ressource`:

```
int angemeldeteWriter;
boost::mutex writerAnmeldungsMutex;
```

Der Mutex garantiert, dass der gleichzeitige Zugriff von mehr als einem Writer auf die Variable `angemeldeteWriter` keine Inkonsistenz erzeugt. Die damit modifizierte Methode `write()` kann jetzt von vielen Writern betreten werden. Bis auf einen bleiben sie am `unique_lock` hängen, bis der erste den Schreibvorgang beendet hat:

```
// Auszug aus cppbuch/k13/rwp/Ressource.h
void write(const std::string& neu, const std::string& id) {
    {
        boost::lock_guard<boost::mutex> anmeldungsLock(writerAnmeldungsMutex);
        ++angemeldeteWriter;
    } // Lock wird freigegeben, sodass sich weitere Writer anmelden können.
    // Ab hier muss ggf. gewartet werden:
    boost::unique_lock<boost::shared_mutex> lock(rwMutex);
    ++nwriter;
    println(id + " schreibt " + neu + rwprotokoll());
    boost::this_thread::sleep(boost::posix_time::millisec(1000 + derZufall()));
    --nwriter;
    inhalt = neu;
    boost::lock_guard<boost::mutex> anmeldungsLock(writerAnmeldungsMutex);
```

```
--angemeldeteWriter;    // abmelden
}
```

Auch das Herunterzählen muss verriegelt werden. Für `anmeldungsLock` ist im Gegensatz zum Eingang der Methode kein eigener Block erforderlich, weil die Methode zwei Zeilen später beendet und damit das Lock freigegeben wird. In der Methode `read()` wird die Information über die Anzahl der Anmeldungen ausgewertet:

```
std::string read(const std::string& id) {
    while(angemeldeteWriter > 0) {
        boost::this_thread::sleep(boost::posix_time::millisec(500)); // siehe Text
    }
    boost::shared_lock<boost::shared_mutex> lock(rwMutex);
    ++nreader;
    println(id + " liest " + inhalt + rwprotokoll());
    boost::this_thread::sleep(boost::posix_time::millisec(2000 + derZufall()));
    --nreader;
    return inhalt;
}
```

Die `while`-Schleife am Anfang sorgt dafür, dass kein Reader bei angemeldeten Writern versucht, das `shared_lock` in Anspruch zu nehmen. Ob die Bedingung noch gilt, wird periodisch überprüft. Auf Seite 433 wird empfohlen, Polling nicht einzusetzen, wenn es eine Alternative mit `wait` gibt. Diese Alternative ist in diesem Fall nicht oder nur schwierig zu realisieren. Nehmen Sie an, in der Schleife gäbe es ein `wait()`, etwa

```
while(angemeldeteWriter > 0) {
    condition.wait(lock);
}
```

Dann müsste, sobald das Ende der Methode `write()` erreicht wird, ein `notify_all()` erfolgen, um die Reader zu befreien. Das Problem: Es kann sein, dass dies *nach* Zeile 1, aber *vor* Zeile 2 geschieht. Ein `notify_all()` vor dem `wait()` ist jedoch wirkungslos! Da ab diesem Zeitpunkt möglicherweise kein anderer Writer mehr aktiv wird, würden die Reader nie befreit. Die Alternative, die Schleife atomar auszuführen, indem sie mit einem Lock versehen wird, ist keine, weil erstens viele Reader in der Schleife warten sollen und zweitens mit demselben Lock das Herunterzählen von `angemeldeteWriter` samt `condition.notify_all()` versehen werden müsste. Damit wäre ein Deadlock vorprogrammiert.

13.6.2 Reader/Writer-Varianten

Eine ganze Datenbank wegen eines einzelnen Writer-Zugriffs für alle Reader zu sperren, ist bei stark durch lesende Anfragen belastete Datenbanken nicht akzeptabel. Deswegen ist es üblich, verschiedene Grade von Sperrungen zu ermöglichen, wobei zwischen Aufwand und Wirkung abgewogen werden muss. So kann bei einem Update nur eine Tabelle gesperrt werden oder sogar nur ein Record. Dasselbe Problem tritt nicht nur bei Datenbanken, sondern bei allen komplexen Datenstrukturen auf, wenn sie intensiv genutzt werden. Je feiner granular die Sperrung ist, desto höher ist der Verwaltungsaufwand. Die dadurch sinkende Performanz zahlt sich aus, wenn die Parallelität der Zugriffe nur wenig eingeschränkt wird. Ein allgemeingültiges Rezept gibt es nicht; die tatsächliche Nutzung im Einzelfall muss betrachtet werden.

13.7 Thread-Sicherheit

Threads bringen durch die Parallelität große Vorteile, genügend Prozessoren vorausgesetzt. Wenn es um gemeinsame Daten geht, ist der Synchronisationsaufwand teilweise erheblich, aber unerlässlich. Der Begriff *Thread-Sicherheit* einer Klasse bedeutet, dass alle Methoden korrekt ausgeführt werden, unabhängig von der Anzahl der ausführenden Threads. Das bedeutet auch, dass der Zustand eines Objekts dieser Klasse, gegeben durch die Werte seiner Attribute, stets konsistent (widerspruchsfrei) bleibt, und zwar ohne dass Benutzer der Klasse selbst dafür sorgen müssen. Die Klasse stellt intern alle notwendigen Synchronisationsmechanismen zur Verfügung. Bei dem Entwurf so einer Klasse ist zu berücksichtigen:

- Es dürfen keinerlei Annahmen getroffen werden, in welcher Reihenfolge Threads vom Betriebssystem zur Ausführung gebracht werden. Keine der möglichen Reihenfolgen darf das Objekt in einem ungültigen Zustand hinterlassen. Anders gesagt: Es darf keine Race Conditions geben, also Bedingungen, unter denen das Programmergebnis vom Timing oder der Abfolge von Threads abhängt.
- Zusammengehörende Operationen wie das Lesen und Verändern einer gemeinsam genutzten Variablen müssen *atomic* ausgeführt werden (siehe Beispiel auf Seite 426).
- Thread-sichere Software muss extrem sorgfältig entworfen werden, weil vollständige Tests auf Thread-Sicherheit sehr schwierig bis unmöglich sind. In der Regel können nicht alle möglichen Abfolgen von Threads überprüft werden. So kann es sein, dass ein Programm zwar alle Tests besteht, jedoch in einigen Jahren ein Fehler auftritt, weil zufällig zwei Threads unsynchronisiert dieselbe Variable ändern.
- Ebenso sorgfältig muss die Reihenfolge der Akquirierung und Freigabe von Ressourcen geplant werden, damit kein Deadlock (Verklemmung) entsteht. Ein Deadlock bewirkt, dass das Programm stehen bleibt. Eine typische Ursache: Thread A akquiriert Ressource X, Thread B akquiriert Ressource Y. Nun möchte A ebenfalls Y akquirieren, und Thread B die Ressource X. Weil die Ressourcen vorher nicht freigegeben wurden, warten beide bis zum Jüngsten Tag aufeinander oder bis jemand das Programm mit Strg+C abbricht.

Im Zusammenhang mit der Thread-Sicherheit wird gelegentlich der Begriff *reentrant* (dt. etwa wiedereintrittsfähig) genannt. Darunter ist zu verstehen, dass eine Funktion gleichzeitig ohne Probleme von verschiedenen Threads aufgerufen werden kann, wenn keine gemeinsamen Daten benutzt werden. Die Voraussetzungen sind bei vielen modernen Programmiersprachen gegeben: Der ausführbare Programmcode liegt in einem schreibgeschützten Bereich (kann also nicht geändert werden), und jedem Aufruf wird ein eigener lokaler Speicherbereich für die Daten zugewiesen.

In diesem Kapitel werden die Grundzüge der Arbeit mit C++-Threads dargestellt. Mehr über Boost-Threads erfahren Sie in [\[thread\]](#). Wenn Sie mehr über die Anwendung von Software-Mustern in der Programmierung paralleler Anwendungen wissen möchten, kann ich Ihnen das Buch [\[SSRB\]](#) empfehlen. Grundlagen über die Parallelität von Prozessen, Synchronisation, Deadlock-Entdeckung und -Vermeidung finden Sie in Büchern, in deren Titel »Parallele Programmierung«, »concurrent programming« oder »operating system principles« vorkommen.



Teil II: Bausteine komplexer Anwendungen

14

Grafische Benutzungsschnittstellen

Dieses Kapitel behandelt die folgenden Themen:

- Ereignisgesteuerte Programmierung
- Meta-Objektsystem
- Signale, Slots und Widgets
- Dialoge

Standard C++ kennt keine Elemente für grafische Benutzungsoberflächen (englisch *graphical user interfaces*) (GUI). Nichtsdestoweniger sind GUIs nicht mehr wegzudenken, weswegen hier eine Einführung in die GUI-Programmierung gegeben wird. Die entsprechenden Komponenten finden sich in verschiedenen Bibliotheken. Sehr bekannt sind die Microsoft Foundation Classes (MFC) bzw. ihre Nachfolger in .NET für Windows-Betriebssysteme. Die in Abschnitt 1.5 empfohlene Entwicklungsumgebung Code::Blocks kann unter anderem auf Basis der Windows-API Programme mit einer grafischen Benutzungsschnittstelle erzeugen. Der Nachteil der genannten Möglichkeiten ist die mangelnde Portabilität.

GTK+ (= the GIMP Toolkit) ist eine weitere Bibliothek zur Erstellung grafischer Benutzungsoberflächen. GTK+ wurde ursprünglich für GIMP, ein mächtiges Bildbearbeitungsprogramm, entwickelt. GTK+ wird zur Entwicklung des GNOME-Desktops benutzt. GNOME ist eine Benutzungsoberfläche, die auf vielen Linux-Systemen zu finden ist. GTK+ gibt es für verschiedene Betriebssysteme und ist nicht lizenzpflichtig, auch dann

nicht, wenn kommerzielle Software damit entwickelt wird. Die Homepage von GTK+ ist <http://www.gtk.org>.

Ebenfalls sehr bekannt ist die portable Qt-Bibliothek für Windows-, Mac- und Unix-Betriebssysteme. Die in manchen Linux-Systemen vorhandene Benutzungsoberfläche KDE wird mit Qt entwickelt. Im Gegensatz zu GTK+ gibt es eine sehr ausführliche Dokumentation. Qt zeichnet sich durch weitere hervorstechende Merkmale aus:

- Das Programm Qt Assistant (Aufruf `assistant`) ist ein umfangreiches Hilfesystem für alle Qt-Programme.
- Das Programm Qt Designer (Aufruf `designer`) ist ein Werkzeug, das es ermöglicht, mithilfe von Mausoperationen eine grafische Benutzungsoberfläche am Bildschirm zu erzeugen und zu konfigurieren.
- Das Programm Qt Linguist unterstützt die Anpassung von Anwendungen, die international nutzbar sein sollen, an verschiedene Sprachen.
- Das Programm Qt Creator ist eine IDE speziell für Qt-Anwendungen.
- Ab Version 4.5 ist Qt nicht nur mit einer kommerziellen Lizenz, sondern auch als Open Source Software erhältlich – der Grund, warum Sie sie mitsamt den Lizenzen auf der DVD finden. Eine vergleichende Übersicht der Lizenzen finden Sie unter <http://www.qt.nokia.com/products/licensing>.
- Es gibt Unterstützung für die Arbeit mit Verzeichnissen, ein Hilfesystem, die Verarbeitung von XML und die Netzwerkprogrammierung. Eine integrierte Datenbank ist ebenfalls vorhanden.

Zusammengefasst: Qt ist die ausgereifteste und umfangreichste Open Source Software für die portable Entwicklung grafischer Benutzungsoberflächen mit C++. Java-Kenner finden in Qt all das, was sie an Standard C++ vermissen. Aus diesem Grund wird in diesem Kapitel ein kleiner Einblick in Qt gegeben.



14.1 Ereignisgesteuerte Programmierung

Die bislang in diesem Buch dargestellte Programmierung beruht darauf, dass die Abarbeitung der Programmschritte wie vorgesehen der Reihe nach abläuft. Das gilt auch innerhalb der Threads des Kapitels 13 – nicht aber für die Programmierung grafischer Benutzungsoberflächen. Der Grund ist einfach: Die Reihenfolge der Interaktionsschritte eines Benutzers ist nicht vorhersehbar und damit auch nicht die Reihenfolge der auszuführenden Programmschritte. Die Aktion eines Benutzers, wie zum Beispiel Anklicken eines Menüpunktes oder Bewegung mit der Maus, wird als *Ereignis* aufgefasst, auf das das Programm reagieren soll. Das Ereignis wird vom Betriebssystem erkannt und an das Programm, wenn es sich dafür angemeldet (registriert) hat, weitergeleitet.

Es geht also darum, Funktionen zu schreiben, die nicht von anderer Stelle desselben Programms, sondern bei Eintreffen eines bestimmten Ereignisses aufgerufen werden und

die gewünschte Reaktion liefern. Insofern besteht eine Ähnlichkeit mit den auf Seite 225 erwähnten Callback-Funktionen. Diese Art der Programmierung ist typisch für alle *Frameworks*. Ein Framework (dt. etwa Rahmenwerk) ist eine Software mit bestimmten Funktionen, die die Struktur einer Anwendung vorgibt. Die Anwendung kann diese Funktionen nutzen, indem sie konkrete Implementierungen bereitstellt, die beim Framework registriert und von ihm aufgerufen werden. Der Kontrollfluss wird also wesentlich durch das Framework bestimmt.

Etwas konkreter: Wenn bekannt ist, was bei Anklicken eines Menüpunktes zu tun ist, kann eine entsprechende Funktion geschrieben werden. Dies reicht jedoch nicht aus: Dem Framework muss mitgeteilt werden, dass diese Funktion mit dem Ereignis »Anklicken des Menüpunktes« verbunden werden soll. Wenn das geschehen ist, wird das Framework die Funktion aufrufen, sobald der Menüpunkt angeklickt wird. Wie das im Einzelnen aussieht, sehen Sie weiter unten am Beispiel. Die skizzierte Verfahrensweise gilt für alle Systeme zur GUI-Programmierung, nicht nur für Qt.

14.2 GUI-Programmierung mit Qt

Hinweis: Dieses Kapitel gibt eine erste kurze Einführung in die GUI-Programmierung mit Qt und ist eher für Fortgeschrittene gedacht. Der Schwerpunkt liegt nicht auf den vielfältigen Möglichkeiten der umfangreichen Bibliothek, sondern auf den grundlegenden Konzepten: dem Programmiermodell (Meta-Objekt-Compiler, Signale und Slots) und der Speicherverwaltung. Wer sich näher mit Qt beschäftigen möchte, dem sei das Buch [BISu] empfohlen. Dieses Kapitel bietet eine gute Grundlage für das Verständnis.



Installationshinweise finden Sie im Anhang ab Seite 938 (Windows) bzw. 946 (Linux).

Hilfe zu Qt

Es gibt viele ergiebige Informationsquellen. An erster Stelle ist das laut Nokia »einzige offizielle« Buch [BISu] zu nennen. Als Hilfe während der Programmierung ist das Programm *assistant* unerlässlich, das unter anderem die ausführliche Dokumentation aller Klassen bietet. Bei weitergehenden Fragen können unter anderem die folgenden Webseiten herangezogen werden:

<http://www.qt.nokia.com/developer>,
<http://www.qtforum.de/> und
<http://www.qtcentre.org/>

14.2.1 Meta-Objektsystem

Qt erweitert C++ nicht nur um GUI-Elemente, sondern auch um ein sprachliches Konzept, das Meta-Objektsystem genannt wird. Seine wichtigsten Bestandteile:

- **Introspektion:** Dieser Mechanismus erlaubt es, *zur Laufzeit* Informationen über andere Objekte zu bekommen, zum Beispiel den Namen der Klasse zu erfragen und die zur

Verfügung gestellten Methoden zu ermitteln. Standard-C++ bietet diese Möglichkeit nicht. Deswegen wurde das Qt-Werkzeug *moc* (Meta-Objekt-Compiler) entwickelt. Es analysiert Qt-Quellprogramme und erzeugt C++-Quellcode, der die benötigten Funktionen bereitstellt und bei der Erzeugung eines ausführbaren Qt-Programms übersetzt und eingebunden wird. Mit Hilfe der Introspektion können die Signale und Slots (siehe unten) ermittelt werden.

- **Signale:** Ein Signal entspricht einem Ereignis im obigen Sinn und ist nicht mit dem Begriff »signal« der Unix-Programmierung zu verwechseln. Ein Signal ist einem Sender zugeordnet, zum Beispiel einem Button oder einem anderen GUI-Element.
- **Slots:** Ein Slot ist eine Funktion, die dem Empfänger eines Signals zugeordnet ist und auf das Signal reagieren soll.
- **Verbindung von Signal und Slot:** Die `connect()`-Anweisung verbindet Signale mit Slots:

```
connect(sender, // Zeiger auf Sender-Objekt
        SIGNAL( // Makro
            signal), // einem Signal zugeordneter Funktionsname
        empfaenger, // Zeiger auf Empfänger-Objekt
        SLOT( // Makro
            slot)); // Name der Funktion, die reagieren soll
```

Die Funktionsnamen sind mit runden Klammern (), aber ohne Parameter anzugeben. Mehrere `connect()`-Anweisungen können ein Signal mit mehreren Funktionen verbinden, es ist aber auch möglich, mehrere Signale mit derselben Funktion zu verbinden oder auch Signale an andere Signale zu koppeln.

Konkrete Anwendungen finden Sie weiter unten. `SIGNAL` und `SLOT` sind nur zwei der verwendeten Makros. Makros sind das Mittel, auf den ersten Blick merkwürdige anmutende syntaktische Qt-Konstruktionen in C++ zu verwandeln. Die Makros werden durch den Präprozessor abgearbeitet, sodass der Compiler sie nicht sieht.

14.2.2 Der Programmablauf

Für jede Qt-Anwendung mit einem GUI existiert ein `QApplication`-Objekt. Es dient zur Initialisierung und sorgt für die Ereignisverarbeitung. Letztere wird durch Aufruf der Methode `exec()` angestoßen, die den sogenannten Event Loop enthält. Darunter ist zu verstehen, dass in einer Endlosschleife auf ein Ereignis gewartet wird. Wenn eins eintrifft, wird es verarbeitet und auf das nächste Ereignis gewartet. Die Schleife bricht erst ab, wenn das letzte grafische Element geschlossen ist. Sehen Sie sich das folgende kleine Beispiel an:

Listing 14.1: Das erste Qt-Programm

```
// cppbuch/k14/label/main.cpp
#include <QApplication>
#include <QLabel>
#include <iostream>

class MeinLabel : public QLabel{
public:
    MeinLabel(const char* text)
```

```

        : QLabel(text) {
    }
    ~MeinLabel() {
        std::cout << "Destruktor ~MeinLabel gerufen!" << std::endl;
    }
};

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    MeinLabel* mlabel = new MeinLabel("Erstes Qt-Programm");
    mlabel->show();
    int ergebnis = app.exec();
    delete mlabel;
    return ergebnis;
}

```

Sinn der Klasse `MeinLabel` ist, über die Funktionalität der Klasse `QLabel` hinaus den Destruktoraufwurf zu protokollieren. Der Start des Programms erzeugt ein Label als Mini-Fenster auf dem Bildschirm (Abbildung 14.1).

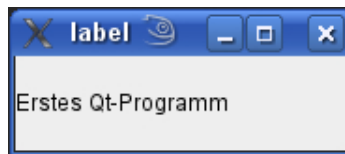


Abbildung 14.1: Label

Zuerst wird das `QApplication`-Objekt angelegt. Sie sehen, dass es einige Parameter von `main()` übernimmt. Dies ist für Qt-interne Zwecke notwendig, sodass `main()` diese Parameter stets aufführen muss. Nach der Konstruktion des `MeinLabel`-Objekts auf dem Heap und seiner Anzeige wird `exec()` ausgeführt. `exec()` kehrt erst dann zurück, wenn das Label durch Anklicken des Kreuzes in der rechten oberen Ecke geschlossen wird. Wenn alles gutgegangen ist, wird 0 zurückgegeben.



Hinweis für Windows-Nutzer

Um Ausgaben für Testzwecke mit `cout` oder `cerr` auf der Konsole zu sehen, müssen Sie in die `.pro`-Datei, die mit `qmake -project` erzeugt wird, die Zeile `CONFIG += console` einfügen. Danach wieder `qmake` und `make` aufrufen.

14.2.3 Speicher sparen und lokal Daten sichern

Qt führt mithilfe einer Baumstruktur Buch über die auf dem Heap angelegten Objekte. Wenn einem Konstruktor ein Zeiger auf ein Eltern-Objekt mitgegeben wird, wird das Objekt in die Liste der Kind-Objekte, die das Eltern-Objekt führt, eingetragen. Der Destruktor des Eltern-Objekts sorgt dafür, dass alle Kind-Objekte gelöscht werden. Damit muss `delete` nur noch für Heap-Objekte aufgerufen werden, die kein Elternobjekt haben.

**Tipp**

Qt-Objekte nicht auf dem Stack, sondern mit `new` anlegen!

Außer der beschriebenen automatischen Löschung gibt es weitere Gründe dafür, die in der Qt-Dokumentation unter der Überschrift »Qt Object Model«, Abschnitt »Qt Objects: Identity vs Value« beschrieben werden.

Ein `QMainWindow` ist das Hauptfenster einer Anwendung. Um es zu verwenden, wird eine eigene Klasse davon abgeleitet. Das folgende Beispiel zeigt, dass `delete` nicht aufgerufen werden muss, wenn das Attribut `WA_DeleteOnClose` gesetzt ist. Es können mehrere Fenster geöffnet sein. Um Speicher zu sparen und möglicherweise noch Daten zu sichern, bewirkt das Attribut den Aufruf des Destruktors und die Löschung des Objekts, wenn es geschlossen wird. Eine zweite Möglichkeit ist das Überschreiben der geerbten `protected`-Methode `closeEvent()`, die automatisch aufgerufen wird, wenn das Fenster geschlossen wird. Beide Möglichkeiten sind dargestellt, obwohl eine davon in der Praxis ausreicht. `#include<QtGui>` schließt *alle* GUI-Elemente ein. Bei großen Projekten sollten nur die tatsächlich benötigten Dateien inkludiert werden, um Compilationszeit zu sparen.

Listing 14.2: Einfaches Window

```
// cppbuch/k14/window/Window.h
#ifndef WINDOW_H
#define WINDOW_H
#include<QtGui>
#include<fstream>

class Window : public QMainWindow {
public:
    Window() {
        setAttribute(Qt::WA_DeleteOnClose);
    }
    ~Window() {
        // save(); Alternative zu closeEvent(), ohne Benutzerinteraktion
    }
protected:
    void closeEvent(QCloseEvent* ce) {
        int antwort = QMessageBox::warning(this, "Titel",
                                           "closeEvent gerufen. Sichern?",
                                           QMessageBox::Yes|QMessageBox::No);
        if(antwort == QMessageBox::Yes) {
            save();
        }
    }
private:
    void save() {
        std::ofstream log("daten.txt");
        log << "gesicherte Daten" << std::endl;
        log.close();
    }
};
```

Listing 14.3: delete ist nicht erforderlich.

```
// cppbuch/k14/window/main.cpp
#include <QApplication>
#include "Window.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    Window* win1 = new Window();
    win1->show();
    Window* win2 = new Window();
    win2->show();
    return app.exec();
}
```

Im Gegensatz zu Aufräumarbeiten auf der Ebene des main-Programms können so pro Fenster abschließende Aktionen wie zum Beispiel eine Datensicherung durchgeführt werden. Eine Benutzerinteraktion wie in `closeEvent()` ist im Destruktor nicht möglich.

14.3 Signale, Slots und Widgets

Die oben kurz beschriebenen Signale und Slots werden in diesem Abschnitt in einem konkreten Beispiel eingesetzt, um das Zusammenwirken mit verschiedenen GUI-Elementen zu demonstrieren. Die GUI-Elemente werden *Widgets* genannt. Die Abbildung 14.2 zeigt ein Fenster des Typs `QMainWindow` mit verschiedenen Widgets. Im Wesentlichen geht es um das Zeichnen eines Kreises, dessen Radius mit der Maus (Widget `QDial`) oder durch Wahl einer Zahl (Widget `QSpinBox`) verändert werden kann.

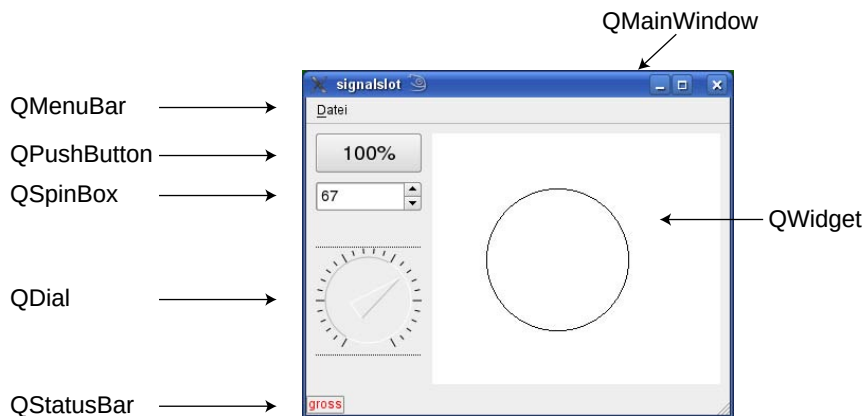


Abbildung 14.2: Qt-Beispiel

Die Elemente sind:

- Ein Menübalken des Typs `QMenuBar`, dem ein Datei-Menü »Datei« zugeordnet ist. Mit der Tastenkombination `Alt+D` (Hotkey) oder durch Mausklick kommt man zu den Einträgen des Menüs. In diesem Fall ist es nur der im Bild nicht sichtbare Eintrag »Ende«, der zum Beenden des Programms führt.
- Eine Statuszeile, die anzeigt, ob der Kreis groß oder klein ist. Falls der genannte Menüeintrag »Ende« aktiviert wird, zeigt die Statuszeile den erläuternden Text »Programm beenden«.
- Mehrere Widgets mit verschiedenen Aufgaben. `QWidget` ist die Basisklasse aller GUI-Objekte. Außer dem Hauptfenster gehören alle Widgets zu einem Eltern-Widget. Sie sind Elemente, die selbst angezeigt werden, oder aber Container für weitere Widgets. Das in der Abbildung 14.2 eingezeichnete `QWidget` bezieht sich nur auf die Zeichenfläche, das umhüllende Widget `QtBeispiel` ist nicht direkt sichtbar. Die Struktur ist am besten im UML-Diagramm unten (Abbildung 14.3) zu erkennen. Die Klasse `QtBeispiel` dient als Container für die nachfolgend aufgeführten Widgets. Das UML-Diagramm zeigt vereinfachend nicht, dass alle Widgets von `QWidget` erben.

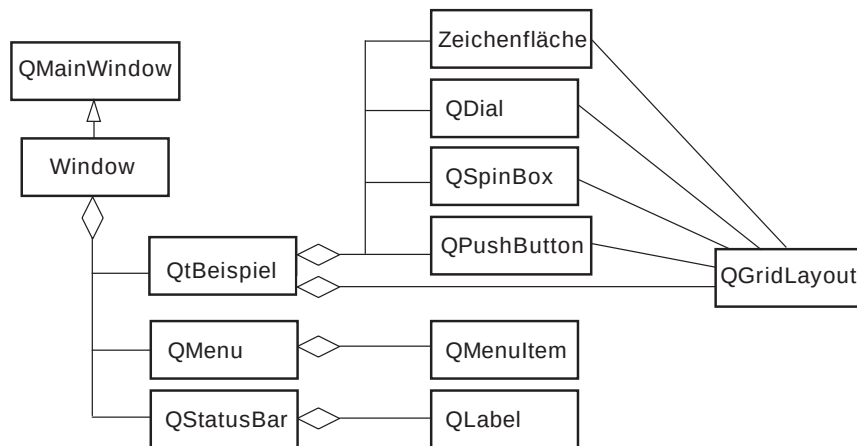


Abbildung 14.3: Vereinfachtes UML-Diagramm zum Beispiel

- Ein »100%«-Button (`QPushButton`), dem das Signal `clicked()` zugeordnet ist. Anklicken bewirkt, dass der Kreis maximal groß gezeichnet wird.
- Ein Eingabefeld des Typs `QSpinBox`, in das eine Zahl eingetippt werden kann. Alternativ kann sie durch Anklicken der Elemente rechts im Eingabefeld hoch- bzw. heruntergezählt werden. So ein Eingabefeld wird »SpinBox« genannt, vom englischen *to spin* (sich drehen), auch wenn die Bewegung innerhalb einer Reihe aufeinanderfolgender Elemente nicht kreisförmig ist. Eine Änderung der Zahl bewirkt eine Radiusänderung des Kreises.
- Eine Einstellscheibe (englisch *dial*) des Typs `QDial`, die mit der Maus bedient werden kann. Eine Änderung der Einstellung bewirkt eine Radiusänderung des Kreises.
- Eine Zeichenfläche, auf der der Kreis dargestellt wird.

Das main-Programm ist gewohnt kurz. Das Löschen des Fensters wird mit dem Schließen erledigt (Attribut `WA_DeleteOnClose` im Window-Konstruktor).

Listing 14.4: Beispielprogramm mit Qt: main()

```
// cppbuch/k14/signalslot/main.cpp
#include <QApplication>
#include "window.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    Window* win = new Window;
    win->show();
    return app.exec();
}
```

Der Konstruktor der Klasse `Window` setzt zunächst das Attribut zum Löschen und dann die äußeren Koordinaten und Abmessungen mit `setGeometry(int x, int y, int breite, int hoehe)`. Die `int`-Werte werden in Pixeln angegeben. Ein `QMainWindow` hat verschiedene Bereiche:

- Am oberen Rand ist der Menübalken (englisch *menu bar*).
- Am unteren Rand befindet sich die Statuszeile.
- In der Mitte ist der Bereich für das zentrale Widget. Dieser Bereich kann von sogenannten »Dock Widgets« oder »Toolbars« umgeben sein. Erstere sind Widgets, die frei innerhalb des erlaubten Bereichs verschoben werden können, z.B. Widgets in Zeichenprogrammen zur Pinsel- oder Farbwahl. Letztere sind verschiebbare Leisten am Rand, die Widgets für verschiedene Aktionen enthalten. Dock Widgets und Toolbars werden im Beispiel nicht verwendet, aber das zentrale Widget sowie Menübalken und Statuszeile. Das zentrale Widget `widget` vom Typ `QtBeispiel` ist ein Container für die sichtbaren Widgets. Es wird in Zeile 14 unten festgelegt.

Listing 14.5: Klasse `Window`

```
1 // cppbuch/k14/signalslot/window.h
2 #ifndef WINDOW_H
3 #define WINDOW_H
4 #include "QtBeispiel.h"
5
6 class Window : public QMainWindow {
7     // Das Makro Q_OBJECT sorgt u.a. für die richtige Interpretation der Signale und Slots.
8     Q_OBJECT
9 public:
10     Window() {
11         setAttribute(Qt::WA_DeleteOnClose);
12         setGeometry(100, 100, 400, 300);
13         widget = new QtBeispiel(this);
14         setCentralWidget(widget);
15         menuAnlegen(); // siehe unten
16         statuszeileAnlegen(); // siehe unten
17         connect(widget->getZeichenflaeche(), SIGNAL(radiusChanged(int)),
18                 this, SLOT(statuszeileAktualisieren(int)));
19     }
```


Die connect-Anweisung (Zeilen 17 und 18) ist wie folgt zu verstehen: Die Zeichenfläche des Widgets ist der Sender des Signals `radiusChanged(int)`. Die Zeichenfläche ruft bei Änderung die Funktion auf und übergibt ihr den aktuellen Radius. Der Empfänger des Signals ist `*this`, also das Window-Objekt, und für dieses Objekt soll die Funktion `statuszeileAktualisieren(int)` mit dem übergebenen Wert aufgerufen werden. `connect` verbindet das Signal `radiusChanged(int)` mit dem Slot (= der aufzurufenden Funktion des Empfängers) `statuszeileAktualisieren(int)`. Damit der Slot vom Meta-Objekt-Compiler erkannt wird, ist `private slots:` vorzuschalten bzw. `public slots:`, wenn der Slot von einem anderen Objekt aus erreichbar sein soll. In der Funktion ist zu sehen, dass auch HTML-formatierter Text möglich ist.

```

20 private slots:
21     void statuszeileAktualisieren(int radius) {
22         if(radius < 51) {
23             statusLabel->setText("<span style=\"color:green\">klein</span>");
24         }
25         else{
26             statusLabel->setText("<span style=\"color:red\">gross</span>");
27         }
28     }
29 private:
30     void menuAnlegen() {
31         fileMenu = menuBar()->addMenu(tr("&Datei"));
32         QAction* quitAction = new QAction(tr("&Ende"), this);
33         quitAction->setShortcut(tr("Ctrl+Q"));
34         quitAction->setStatusTip(tr("Programm beenden"));
35         fileMenu->addAction(quitAction);
36         // qApp ist globale Variable der Applikation
37         connect(quitAction, SIGNAL(triggered()), qApp, SLOT(quit()));
38     }
39
40     void statuszeileAnlegen() {
41         statusLabel = new QLabel("nicht bewegt");
42         statusBar()->addWidget(statusLabel);
43     }
44     QtBeispiel* widget;
45     QMenu* fileMenu;
46     QLabel* statusLabel;
47 };
48 #endif

```

In der Funktion `menuAnlegen()` wird ein Menüeintrag mit einer Aktion verbunden. Die Schritte im Einzelnen (Zeilennummern am Anfang):

31 Das Menü `tr("&Datei")` wird dem Menübalken hinzugefügt. Ein einem Buchstaben vorangestelltes `&`-Zeichen definiert diesen Buchstaben als Hotkey. `tr()` steht für *translate* und ist eine Funktion, die bei der tatsächlichen Ausgabe für die richtige Sprache sorgt, sofern dieses Feature eingestellt ist.

32 Es wird eine Aktion des Namens `tr("&Ende")` erzeugt.

- 33 Die Tastenkombination Ctrl+Q (= Strg-Q) wird als Shortcut definiert. Mit dieser Tastenkombination wird das Programm auch ohne Anwahl des Menüeintrags beendet.
- 34 Bei Aktivieren des Eintrags wird der erläuternde Text in der Statuszeile angezeigt.
- 35 Die Aktion wird dem Menü hinzugefügt.
- 37 Damit die Anwahl dieses Menü-Items tatsächlich zum Beenden des Programms führt, muss das von ihm ausgehende Signal mit der `quit()`-Methode der Applikation verbunden werden. `qApp` ist eine in jeder Qt-Anwendung definierte globale Variable, die auf das `QApplication`-Objekt zeigt. Sender des Signals ist das Objekt `*quitAction`. Das Signal ist für Aktionen vordefiniert und heißt `triggered()`. Empfänger des Signals ist das `QApplication`-Objekt `qApp`, dessen Slot-Methode `quit()` bei Auftreten des Signals gerufen wird.

Das allgemeine Aussehen wird vom aktuellen System übernommen, das heißt, unter Windows würde die Abbildung 14.2 wie ein typisches Windows-Fenster aussehen. Im Folgenden werden die noch fehlenden Dateien *QtBeispiel.h* und *Zeichenflaeche.h* dargestellt und kommentiert.

Listing 14.6: Klasse `QtBeispiel`

```

1 // cppbuch/k14/signalslot/QtBeispiel.h
2 #ifndef QTBEISPIEL_H
3 #define QTBEISPIEL_H
4 #include "Zeichenflaeche.h"
5
6 class QtBeispiel : public QWidget {
7     Q_OBJECT
8 public:
9     QtBeispiel(QMainWindow* parent = 0)
10         : QWidget(parent),
11           zeichenflaeche(new Zeichenflaeche),
12           qdial(new QDial),
13           qspinbox(new QSpinBox) {
14         qdial->setNotchesVisible(true); // Marken sichtbar machen
15         qspinbox->setFont(QFont("Helvetica", 12, QFont::StyleNormal));
16         QPushButton *maxWertButton = new QPushButton("100%");
17         maxWertButton->setFont(QFont("Helvetica", 16, QFont::Normal));
18
19         // Connect-Aufrufe
20         connect(qdial, SIGNAL(valueChanged(int)), zeichenflaeche,
21                SLOT(setRadius(int)));
22         connect(qdial, SIGNAL(valueChanged(int)), qspinbox, SLOT(setValue(int)));
23         connect(qspinbox, SIGNAL(valueChanged(int)), qdial, SLOT(setValue(int)));
24         connect(maxWertButton, SIGNAL(clicked()), this, SLOT(setMaxwert()));
25
26         QGridLayout *gridLayout = new QGridLayout(this);
27         gridLayout->addWidget(maxWertButton, 0, 0);
28         gridLayout->addWidget(qspinbox, 1, 0);
29         gridLayout->addWidget(qdial, 2, 0);
30         gridLayout->addWidget(zeichenflaeche, 0, 1, 3, 1);

```

```

31 // Zeile, Spalte, erstreckt sich über 3 Zeilen und eine Spalte
32 gridLayout->setColumnStretch(1, 10); // Spalte 1 breiteren Raum einräumen
33 setLayout(gridLayout);
34 }
35 qdial->setValue(1); // Startwert
36 const Zeichenflaeche* getZeichenflaeche() const {
37     return zeichenflaeche;
38 }
39 private slots:
40     void setMaxwert() {
41         qdial->setValue(100);
42     }
43 private:
44     Zeichenflaeche* zeichenflaeche;
45     QDial* qdial;
46     QSpinBox* qspinbox;
47 };
48 #endif

```

11-13 In diesen Zeilen werden drei der Widgets initialisiert, für die QtBeispiel als Container dient.

16 Konstruktion des 100%-Buttons, des vierten angezeigten Widgets.

20-21 Entsprechend der oben dargestellten Logik werden Signale und Slots verbunden. Es wird dafür gesorgt, dass Änderungen der Drehscheibe *qdial zum Aufruf der Funktion setRadius(int) des Objekts *zeichenflaeche führen. Der Aufruf von setRadius(int) bewirkt das Zeichnen des Kreises mit einem neuen Radius (siehe Listing der Klasse Zeichenflaeche unten).

22 Hier können Sie sehen, dass dasselbe Signal an mehrere Empfänger geleitet werden kann, in diesem Fall an die SpinBox, damit gleichzeitig die Zahlen aktualisiert werden.

23 Umgekehrt sollen sich Änderungen der SpinBox in der Einstellung der Drehscheibe bemerkbar machen. Dabei wird ein Problem deutlich: Wenn die Spinbox die Drehscheibe benachrichtigt und diese daraufhin nicht nur den Zustand ändert, sondern wiederum die Spinbox informiert, könnte es eine unendliche Folge gegenseitiger Aufrufe geben! In diesen von Qt vorgegebenen Klassen wird dafür gesorgt, dass so etwas nicht geschehen kann (siehe Tipp unten).

24 Das Signal clicked() ist für QPushButton vordefiniert. Anklicken des Buttons führt zum Aufruf der Methode setMaxwert(), die ihrerseits dafür sorgt, dass die Drehscheibe qdial entsprechend gesetzt wird (Zeile 41). Dieses wiederum führt durch die connect-Anweisungen der Zeilen 20-23 zum Neuzeichnen des Kreises mit dem Radius 100 und zur Aktualisierung der SpinBox. Der voreingestellte mögliche Bereich der SpinBox liegt zwischen 0 und 99, wenn er nicht zum Beispiel mit setRange(int min, int max) anders definiert wird. Die Klasse sorgt dafür, dass der angezeigte Wert nicht überschritten wird. Wenn der Wert auf 100 gesetzt wird, erscheint also tatsächlich nur 99.

26–30 Das `QGridLayout`-Objekt ist ein Layout-Manager, der die Widgets in einer Tabellenform anordnet. Der Funktion `addWidget(QWidget w, int zeile, int spalte)` wird dabei die gewünschte Position mitgegeben. Die Zählung beginnt ab 0. Nun kann es sein, dass sich ein Objekt über mehrere Zeilen und Spalten erstreckt, wie bei der Zeichenfläche in unserem Fall. Dafür gibt es eine überladene Variante: `addWidget(QWidget w, int zeile, int spalte, int anzahlDerZeilen, int anzahlDerSpalten)`.

33 Das Layout wird dem Container zugeordnet.

35 Der Startwert wird festgelegt. Dank des Signal/Slot-Mechanismus pflanzt sich der eingestellte Wert auf die `SpinBox` und die Zeichenfläche fort.



Tip

Bei gegenseitigen Informationen eigener Klassen muss darauf geachtet werden, Rekursion zu verhindern.

Die einfachste Möglichkeit, Rekursion zu verhindern, ist, nur dann etwas zu tun, wenn sich der Zustand tatsächlich ändert, etwa

```
void setZustand(int neuerZustand) {
    if(zustand != neuerZustand) { // nur dann etwas tun!
        zustand = neuerZustand;
        // weitere damit verbundene Aktionen ausführen
        // beteiligte Objekte informieren
    }
}
```

Die Zeichenfläche dient ausschließlich der Anzeige und der Aktualisierung des Kreises. Der Konstruktor legt den Anfangswert des Radius und die Hintergrundfarbe fest. Erläuterungen zu den anderen Funktionen folgen nach dem Listing.

Listing 14.7: Klasse `Zeichenflaeche`

```
1 // cppbuch/k14/signalslot/Zeichenflaeche.h
2 #ifndef ZEICHENFLAECH_H
3 #define ZEICHENFLAECH_H
4 #include <QtGui>
5
6 class Zeichenflaeche : public QWidget {
7     Q_OBJECT
8 public:
9     Zeichenflaeche() {
10         aktuellerRadius = 1;
11         setPalette(QPalette(QColor(255, 255, 255))); // weiß
12         setAutoFillBackground(true);
13     }
14
15 public slots:
16     void setRadius(int radius) {
17         if(radius != aktuellerRadius) { // nur dann etwas tun!
18             if (radius < 1) {
```

```

19         radius = 1;
20     }
21     aktuellerRadius = radius;
22     update();
23     emit radiusChanged(aktuellerRadius); // Änderung signalisieren
24 }
25 }
26
27 signals:
28     void radiusChanged(int neuerRadius);
29
30 protected:
31     void paintEvent(QPaintEvent*) { // Parameter wird nicht benutzt
32         QPainter painter(this);
33         // Transformation der Koordinaten auf die Zeichenfläche
34         painter.translate(0, rect().height());
35         int offset = rect().height()/2;
36         int r = aktuellerRadius;
37         // einen durch ein Rechteck definierten Kreis zeichnen:
38         painter.drawEllipse(QRectF(offset-r, -offset-r, 2*r, 2*r));
39     }
40 private:
41     int aktuellerRadius;
42 };
43 #endif

```

- 15-25 Der Slot `setRadius(int radius)` wird über den Signal-Mechanismus von den anderen Widgets aufgerufen. Es wird nur dann etwas getan, wenn sich der Radius geändert hat. Damit ist eine Zusammenarbeit mit möglichen weiteren, noch zu schreibenden Klassen ohne Rekursion gewährleistet. Allein für dieses Beispiel könnte die Abfrage auf eine Änderung des Radius entfallen, weil schon die Qt-Widgets Rekursion verhindern; es wäre jedoch schlechter Stil und riskant.
- 22 Die Funktion `update()` plant einen `paintEvent()`-Aufruf (Zeile 31) ein. Der eigentliche Zeichenvorgang kostet Zeit. Deshalb sammelt Qt aus Performance-Gründen intern alle Aufrufe zum Zeichnen und führt sie erst bei passender Gelegenheit zusammen aus. Außer der erhöhten Geschwindigkeit ist ein weiterer Vorteil das reduzierte Flackern eines Bildes.
- 23 Hier sehen Sie ein neues Schlüsselwort. `emit` ist tatsächlich ein Makro. Die Bedeutung ist klar: die Erzeugung eines Signals. Das Signal wird von der Klasse `Window` ausgewertet, die gegebenenfalls die Statuszeile aktualisiert (Zeilen 17-18 im Listing der Klasse `Window` oben).
- 27-28 Das Signal wird hier definiert. Sie sehen, dass es die Form einer Funktionsdeklaration hat. Wo bleibt die Funktionsdefinition? Sie wird vom Meta-Objekt-Compiler (moc) erzeugt. Wie das Ergebnis, auf das ich hier *nicht* eingehen möchte, aussieht, können Sie herausfinden, wenn Sie sich die vom moc erzeugten `.cpp`-Dateien ansehen. Der Dateiname beginnt mit `moc_`.

Zum Ablauf: Im Konstruktor wird der Startwert des Radius mit 1 festgelegt. Einmalig, ohne dass seitens des Programms etwas dazu notwendig ist, wird das Zeichnen ausgeführt. Der Aufruf der Methode `setRadius()` im Konstruktor kann deswegen entfallen. Auch würde beim ersten Mal der Signal-Mechanismus nicht funktionieren, weil die `connect`-Anweisungen im Konstruktor von `QtBeispiel` noch nicht abgelaufen sind. Das erste Zeichnen hätte keine Auswirkungen auf die anderen beteiligten Widgets. Damit alle Anzeigen mit dem Kreisradius garantiert übereinstimmen, wird am Ende des Konstruktors von `QtBeispiel`, also nach Ablauf der `connect`-Anweisungen, der Startwert für alle Widgets mit `q dial->setValue(1)`, festgelegt (Zeile 35 im Listing von `QtBeispiel`).

14.4 Dialog

Ein Dialog ist ein GUI-Element zur Abfrage von Informationen, die eine Anwendung benötigt, zum Beispiel gewünschte Farben, Namen, Einstellungen für die Schriftgröße und mehr. Im folgenden Beispiel wird ein einfacher Dialog zur Eingabe eines Namens sowie seine Wechselwirkung mit dem aufrufenden Programm gezeigt. Der Dialog des Typs `NamenDialog` (siehe Listing unten) besitzt außer dem Eingabefeld einen Button zur Bestätigung der Eingabe und einen Button zum Abbruch, wie Abbildung 14.4 zeigt. `NamenDialog` erbt von der Qt-Klasse `QDialog`.

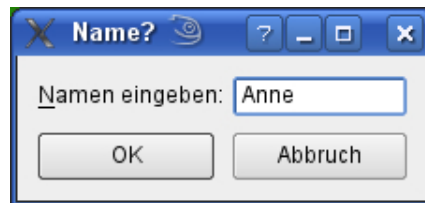


Abbildung 14.4: Einfacher Dialog

Listing 14.8: Dialog zur Eingabe eines Namens

```

1 // cppbuch/k14/dialog/NamenDialog.h
2 #ifndef NAMENDIALOG_H
3 #define NAMENDIALOG_H
4 #include<QtGui>
5
6 class NamenDialog : public QDialog {
7     Q_OBJECT
8     public:
9         NamenDialog(QWidget* parent = 0)
10             : QDialog(parent) {
11             setWindowTitle("Name?");
12             eingabezeile = new QLineEdit;
13             QLabel* text = new QLabel("&Namen eingeben:");
14             text->setBuddy(eingabezeile);

```

```

15     QPushButton* okButton = new QPushButton("OK");
16     QPushButton* abbruchButton = new QPushButton("Abbruch");
17
18     QVBoxLayout* zeile1 = new QVBoxLayout(QBoxLayout::LeftToRight);
19     zeile1->addWidget(text);
20     zeile1->addWidget(eingabezeile);
21     QVBoxLayout* zeile2 = new QVBoxLayout(QBoxLayout::LeftToRight);
22     zeile2->addWidget(okButton);
23     zeile2->addWidget(abbruchButton);
24     QVBoxLayout* alles = new QVBoxLayout(QBoxLayout::TopToBottom);
25     alles->addLayout(zeile1);
26     alles->addLayout(zeile2);
27     setLayout(alles);
28
29     connect(okButton, SIGNAL(clicked()), this, SLOT(accept()));
30     connect(abbruchButton, SIGNAL(clicked()), this, SLOT(reject()));
31 }
32 QString getText() {
33     return eingabezeile->text();
34 }
35 private:
36     QLineEdit* eingabezeile;
37 };
38 #endif

```

Die Erläuterung des Listings, soweit es Qt betrifft:

- 11–16 Diese Zeilen setzen den Titel des Dialogs und definieren alle Widgets. Der Aufruf `text->setBuddy(eingabezeile)` bewirkt, dass die Tastenkombination Alt+N den Focus auf die Eingabezeile bringt. Der Buchstabe N ist in der Variablen `text` als Shortcut festgelegt.
- 18 Ein `QBoxLayout` sorgt dafür, dass Widgets in einer Reihe angeordnet werden. Die Reihe kann horizontal oder vertikal sein. Der dem Konstruktor übergebene Parameter definiert die Anordnung.
- 19–20 Dem Layout-Objekt werden das Label und die Eingabezeile von links nach rechts hinzugefügt.
- 21–23 Das nächste `QBoxLayout`-Objekt `zeile2` nimmt die beiden Buttons auf.
- 24 Das `QBoxLayout`-Objekt `alles` definiert eine vertikale Anordnung von oben nach unten.
- 25–26 Diesem Objekt werden die beiden Zeilen hinzugefügt, sodass sie übereinander liegen, entsprechend der Abbildung 14.4.
- 27 Das Layout für diesen `QDialog` wird festgelegt.
- 29–30 Die Signale der beiden Buttons werden mit Slots verbunden. Die Slots `accept()` und `reject()` sind nicht im Listing zu finden, weil sie von der Oberklasse geerbt werden.

Beide Slots bewirken, dass der Dialog geschlossen (aber nicht gelöscht!) wird. `accept()` setzt den Zustand des Dialogs auf `QDialog::Accepted`, `reject()` auf `QDialog::Rejected`. Diese beiden Konstanten (1 und 0) können mit der geerbten Methode `result()` abgefragt werden.



Tipp

Wenn Sie nach Schließen des Dialogs noch Fragen an das Objekt haben, also die Methode `result()` oder `getText()` aufrufen, muss das Dialog-Objekt, obgleich geschlossen und deswegen nicht mehr sichtbar, noch existieren. Aus diesem Grund darf das Attribut `WA_DeleteOnClose` nicht gesetzt werden!

Natürlich muss das Dialog-Objekt irgendwann »entsorgt« werden. Das kann man dem Eltern-Objekt überlassen, oder, wenn man den Speicher früh freigeben möchte, nach Abfrage der gewünschten Informationen selbst erledigen. Dieser Weg wird in dem folgenden Mini-Programm, das nur die Benutzung des Dialogs zeigen soll, eingeschlagen.

Listing 14.9: Window als Dialog-Benutzer

```
// cppbuch/k14/dialog/DialogUser.h
#ifndef DialogUser_h
#define DialogUser_h
#include "NamenDialog.h"

class DialogUser : public QMainWindow {
    Q_OBJECT
public:
    DialogUser() {
        setAttribute(Qt::WA_DeleteOnClose);
        namendialog = new NamenDialog(this);
        namendialog->setModal(true);
        namendialog->show();
        connect(namendialog, SIGNAL(accepted()), this, SLOT(dialogBeendet()));
        connect(namendialog, SIGNAL(rejected()), this, SLOT(dialogBeendet()));
    }
private slots:
    void dialogBeendet() {
        // Zuerst Ergebnis abfragen (hier mit Anzeige) ...
        QString str("Dialog abgebrochen");
        if(namendialog->result() == QDialog::Accepted) { // doch erfolgreich
            str = namendialog->getText();
        }
        QMessageBox msgBox;
        QString msg("Dialogergebnis: ");
        msg += str;
        msgBox.setText(msg);
        msgBox.exec();
        // und dann Dialog schließen
        delete namendialog;
    }
private:
    NamenDialog* namendialog;
```



```
};
#endif
```

Für das Hauptfenster kann `WA_DeleteOnClose` natürlich gesetzt werden. Der Konstruktor erzeugt den Dialog und setzt die Eigenschaft »modal«. Ein modaler Dialog zeichnet sich dadurch aus, dass er erst bearbeitet werden muss, ehe ein anderes Fenster der Anwendung den Focus bekommen kann. Im Beispiel ist das leicht zu sehen: Das Hauptfenster kann nicht geschlossen werden, solange der Dialog noch geöffnet ist. Wenn der Dialog nicht modal ist, kann jedes Fenster der Anwendung den Focus bekommen (aktiviert werden).

Auf Seite 462 ist zu sehen, dass dasselbe Signal an verschiedene Empfänger gesendet werden kann. Hier hingegen sehen Sie an den connect-Anweisungen, dass verschiedene Signale an denselben Empfänger, den Slot `dialogBeendet()`, gehen. Die Methode `dialogBeendet()` ermittelt den Zustand des Dialogs und, wenn er nicht abgebrochen wurde, den eingegebenen Text. Qt bietet eine Reihe vorgefertigter Dialoge. `QMessageBox` ist einer davon; er wird zur Anzeige des Ergebnisses verwendet. Abschließend wird der Dialog mit `delete` gelöscht. Der Destruktor der Oberklasse sorgt dafür, dass der Dialog aus der Kind-Liste des Eltern-Objekts (hier `DialogUser`, Oberklasse `QMainWindow`) ausgetragen wird, so dass der Destruktor der Klasse `QMainWindow` dieses »Kind« nicht mehr löscht. Das `main`-Programm ist kurz und schlicht:

```
#include "DialogUser.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    DialogUser* dialogUser = new DialogUser;
    dialogUser->show();
    return app.exec();
}
```

14.5 Qt oder Boost?

Qt ist, wie am Anfang des Kapitels schon erwähnt, nicht nur für GUIs geeignet, sondern bietet viele weitere Funktionen wie ein eigenes Thread-Modell, die Arbeit mit Verzeichnissen und mehr. Es zeigt sich, dass die Boost-Library und Qt eine beachtliche Schnittmenge der angebotenen Funktionalitäten aufweisen. Wenn Sie ohnehin wegen einer zu schreibenden Benutzungsoberfläche Qt verwenden, bietet es sich an, auch die anderen Qt-Funktionen zu nutzen – andernfalls genügt die Boost-Library (oder Sie nehmen auch dann Qt). Im Folgenden wird exemplarisch zum Vergleich auf die anders als in [ISOC++] realisierten Threads und das rekursive Durchwandern eines Verzeichnisbaums eingegangen. Dabei werden die Beispiele den entsprechenden Boost-Beispielen in diesem Buch nachempfunden, um einen direkten Vergleich zu bieten. Weitere Möglichkeiten wie Netzwerkprogrammierung und mehr bitte ich dem Buch [BISu] zu entnehmen.

14.5.1 Threads

QThread-Objekte sind anders als Standard-C++-Threads. Um einen Thread zu erzeugen, erzeugt man eine Unterklasse von QThread, in der die Methode `run()` überschrieben wird.

Listing 14.10: Thread-Beispiel

```
// cppbuch/k14/thread/thread.cpp
#include<QThread>
#include <iostream>

class MyThread : public QThread {
public:
    MyThread(int t)
        : dauer(t) {
    }
    void run() {
        sleep(dauer);
        std::cout << "Thread beendet! Laufzeit = " << dauer << " s" << std::endl;
    }
private:
    int dauer;
};

using namespace std;

int main() {
    MyThread t1(4);
    MyThread t2(6);
    MyThread t3(2);
    t1.start();      // ruft implizit run() auf
    t2.start();
    t3.start();
    cout << "t1.isRunning(): " << t1.isRunning() << endl;
    cout << "t2.isRunning(): " << t2.isRunning() << endl;
    cout << "t3.isRunning(): " << t3.isRunning() << endl;
    t1.wait(); // warten auf Beendigung
    t2.wait(); // warten auf Beendigung
    t3.wait(); // warten auf Beendigung
    cout << "t1.isRunning(): " << t1.isRunning() << endl;
    cout << "main() ist beendet" << endl;
}
```

Das Beispiel entspricht demjenigen auf Seite 421, das auf der Boost-Bibliothek beruht. Die Unterschiede in den Beispielen sind:

- Ein QThread wird nicht schon mit dem Aufruf des Konstruktors gestartet, sondern erst mit der Methode `start()`. Diese geerbte Methode ruft dann die Methode `run()` auf. Die Methode `run()` direkt aufzurufen, würde zwar zur Abarbeitung der Methode führen, aber nicht zu einem eigenen Thread.
- Die auf Seite 421 verwendete Methode `get_id()` dient dort zur Anzeige, ob der Thread aktiv ist. Qt hat dafür die Methode `isRunning()`.
- Anstatt `join()` heißt es nun `wait()`.

Das Qt-Thread-Modell bietet sehr viele Möglichkeiten zur Parallelisierung von Prozessen wie Thread-Pools, thread-lokale Daten oder ein API zum Schreiben paralleler Anwendungen ohne low-level-Steuerungsmechanismen wie Mutex-Objekte (Namespace `QtConcurrent`).

14.5.2 Verzeichnisbaum durchwandern

Qt bietet ebenfalls viele Möglichkeiten der Dateiein- und -ausgabe. Hier sei exemplarisch das von Standard-C++ nicht unterstützte Lesen von Verzeichnissen und der Abstieg im Verzeichnisbaum gezeigt. Das Beispiel entspricht demjenigen des Abschnitts 25.1.5 auf Seite 732, das auf der Boost-Bibliothek beruht. Von den Dateien des Abschnitts ändert sich nur die Datei `tree.cpp`:

Listing 14.11: Verzeichnisbaum anzeigen

```
// cppbuch/k14/dirtree/tree.cpp
#include<iostream>
#include<stdexcept>
#include<QFileInfo>
#include<QFileInfoList>
#include<QDir>
#include"tree.h"

namespace {
    std::ostream& operator<<(std::ostream& os, const QString& qstr) {
        const char* str = qstr.toAscii().constData();
        os << str;
        return os;
    }

    void baumAnzeigen(const QDir& d, int level) {
        QStringList eintraege = d.entryList(
            QDir::Dirs|QDir::Files|QDir::NoDotAndDotDot);
        QStringList::const_iterator dIterator = eintraege.begin();
        while(dIterator != eintraege.end()) {
            QString fn(*dIterator);
            QString absPfad(d.absolutePath() + "/" + fn);
            for(int i = 0; i < level; ++i) {
                std::cout << " | ";
            }
            std::cout << " |-- " << fn << std::endl;
            QFileInfo qi(absPfad);
            if(qi.isDir()) {
                QDir dir(absPfad);
                baumAnzeigen(dir, level+1);
            }
            ++dIterator;
        }
    }
} // anonymer namespace
```

```
void baumAnzeigen(const std::string& verz) {
    QFileInfo pfad(verz.c_str());
    if(pfad.isDir()) {
        std::cout << verz << std::endl;
        QDir dir(verz.c_str());
        baumAnzeigen(dir, 0);
    }
    else {
        throw std::runtime_error(" ist kein Verzeichnis!");
    }
}
```

Die Ablauflogik ist dieselbe wie die des Programms auf Seite 733. Die Qt-spezifischen Unterschiede sind:

- Qt verwendet eine eigene String-Klasse `QString`, für die kein Ausgabeoperator `<<` definiert ist. Deswegen wird dieser oben im Listing definiert.
- Die Einträge eines Verzeichnisses sind unter anderem auch als `QString`-Liste erhältlich. Der Funktion `entryList()` können Filter übergeben werden. Im Beispiel sind Datei- und Verzeichnisnamen erlaubt, aber nicht, wenn sie nur aus einem oder zwei Punkten bestehen (aktuelles bzw. Oberverzeichnis). Symbolische Links oder Geräte werden oben damit ausgeschlossen.

Fazit

Dieses Kapitel bietet einen Einstieg in die GUI-Programmierung mit Qt und seine anderen Möglichkeiten. Besonders die Umwandlung in ein Modell mit gestaffelten Lizenzen, darunter auch die GNU Public License und die LGPL, lassen vermuten, dass die Verbreitung von Qt zunehmen wird. Da auch die technische Qualität und die Qualität der Dokumentation recht gut sind, ist Qt meiner Meinung nach die beste Möglichkeit zur Realisierung portabler Benutzungsoberflächen mit C++.

15

Internet-Anbindung

Dieses Kapitel behandelt die folgenden Themen:

- Protokolle und Adressen
- Netzwerkprogrammierung mit Sockets
- Internet-Anbindung mit HTTP
- Mini-Webserver

Netzwerkprogrammierung sowohl für interne als auch externe Netze wie das Internet, ist kein Bestandteil der Programmiersprache C++. Die notwendigen Funktionen sind in entsprechenden Bibliotheken »versteckt«, von denen es mehrere gibt. Aus Gründen der Portabilität habe ich mich für die Boost.Asio-Library, die betriebssystemabhängige Aspekte kapselt, entschieden.



Hinweis

Dieses Kapitel stellt eine kurze Einführung in die Netzwerkprogrammierung dar, um die Möglichkeiten der Realisierung mit C++ zu zeigen. Tatsächlich ist die Netzwerkprogrammierung ein komplexes Thema, sodass die hier gebotenen Informationen für professionelle Anwendungen nur ein Einstieg sein können. Am Ende des Kapitels wird auf weiterführende Dokumentationen hingewiesen.

15.1 Protokolle

Die Kommunikation zwischen Programmen verschiedener Rechner wird mittels mehrschichtiger Protokollstapel modelliert. Damit ist gemeint, dass eine Schicht auf dem Protokoll der darunterliegenden basiert. Zum Beispiel basiert HTTP (Hypertext Transfer Protocol), mit dem die HTML-Seiten übertragen werden, auf dem Protokoll TCP (Transmission Control Protocol) der sogenannten Transportschicht nach dem OSI-Modell (siehe zum Beispiel <http://www.selflinux.org/selflinux/html/osi.html>). HTTP gehört zur Anwendungsschicht. Andere Protokolle der Anwendungsschicht:

- `https` ist eine verschlüsselte HTTP-Verbindung über SSL (Secure Sockets Layer), TLS (Transport Layer Security) oder ein ähnliches Protokoll zur Verschlüsselung und Authentifizierung.
- `ftp` (file transfer protocol) zur Übertragung von Dateien über das Netz.
- `smtp` (simple mail transfer protocol) für E-Mail.
- `file` steht für die Übertragung vom lokalen Rechner.

Während das IP (Internet Protocol) die physikalische Verbindung eines Rechners zum Netzwerk identifiziert und die reine Datenübermittlung leistet, ist das TCP für die Zerlegung einer Sendung in Datenpakete und deren Versand zuständig, gegebenenfalls auf unterschiedlichen Transportwegen durch das Internet. Außerdem sorgt das TC-Protokoll dafür, dass die Pakete beim Empfänger in der richtigen Reihenfolge wieder zusammengesetzt werden. TCP/IP ist die Verbindungsart, die im Internet überwiegend verwendet wird.

Ein weiteres übliches Protokoll ist UDP (User Datagram Protocol, auch scherzhaft: Unreliable Datagram Protocol), welches Datenpakete versendet, für die weder garantiert wird, dass sie in der gesendeten Reihenfolge beim Empfänger ankommen, noch dass sie überhaupt ankommen. Das UDP-Protokoll ist einfacher und schneller als TCP/IP. Es ist besonders geeignet, wenn gelegentlicher Datenverlust tolerierbar ist, zum Beispiel bei der Übertragung von Video.

15.2 Adressen

Um eine Verbindung aufnehmen zu können, muss die Adresse bekannt sein. Dabei wird unterschieden zwischen der Adresse des Rechners im Netzwerk und, zur weiteren Unterteilung innerhalb eines Rechners, dem sogenannten Port. Eine Internet-Adresse wird URL (Uniform Resource Locator) oder URI (Uniform Resource Identifier) genannt. URL bezieht sich auf eine Untermenge von URIs. Die stark vereinfachte Syntax ist

protokoll://[host[:port]][/verzeichnisOderDatei]

Je nach Protokoll und Zweck können die Teile in eckigen Klammern optional sein. `host` kann ein Name oder eine IP-Adresse sein. Beispiele:

```
http://www.cppbuch.de  
file:///home/user/dokument.txt  
http://localhost:9090/  
http://192.168.1.12/
```



Mehr zu URIs lesen Sie in [\[URI\]](#).

IP-Adresse

Die Internet-Adresse eines Rechners, wegen des Internet-Protokolls (IP) auch IP-Adresse genannt, ist eine Liste von Zahlen, getrennt durch Punkte. Bei dem zurzeit üblichen Protokoll IPv4 sind es vier Zahlen, von denen jede im Bereich von 0 bis 255 liegt, sodass sich aus den vier mal 8 Bit theoretisch insgesamt etwa 4,3 Milliarden Adressen bilden lassen. Es hat sich gezeigt, dass diese Zahl in vielen Bereichen nicht ausreicht, weswegen das IPv6 entwickelt wurde, dessen IP-Adresse aus *acht* 16-Bit-Zahlen besteht. Die damit mögliche Adressenzahl von 2^{128} oder etwa $3,4 \cdot 10^{38}$ ist so groß, dass sich jedem Ameisenbein und jedem Menschenhaar eine Adresse zuordnen ließe, und es blieben immer noch welche übrig. Die Internetanbieter haben noch nicht auf IPv6 umgestellt; das ist aber nur eine Frage der Zeit. Anbieter von Inhalten haben eine feste, statische IP-Adresse, private PCs wegen der knappen IPv4-Adressenanzahl dagegen in der Regel nicht – ihnen wird bei dem Verbindungsaufbau eine IP-Adresse zugeteilt. Deswegen heißt sie auch dynamische IP-Adresse, die dem Rechner mit dem Dynamic Host Configuration Protocol (DHCP) vom Server zugewiesen wird. In einem kleinen Netzwerk (Intranet) können den einzelnen Computern statische Adressen zugeordnet werden; der Router als Schnittstelle zum Internet bekommt eine dynamische Adresse und bildet sie intern auf die Adressen im Intranet ab.

Hostname

Internet-Adressen sind nicht gut zu merken, weswegen es für die Computer im Internet eingängige Namen gibt. Diesen Namen bezeichnet man als *Domain*, wobei der letzte Teil, zum Beispiel *.de* in *www.cppbuch.de*, Toplevel-Domain genannt wird. *www* kann eine Unterstruktur im Rechner sein oder ein anderer Rechner, der von dem Domain-Rechner angesteuert wird. Zu einem Namen kann es mehrere IP-Adressen geben, um die Last der Anfragen zu verteilen. Das Domain Name System (DNS) ist eine Datenbank, die auf eine großen Anzahl von Servern verteilt ist. Mit Hilfe dieser Datenbank lässt sich die IP-Adresse zu einem Hostnamen ermitteln. Die dazu notwendige Arbeit wird von einem Nameserver genannten Rechner erledigt, der eine Anfrage entweder aufgrund seiner eigenen Tabellen beantworten kann oder die Anfrage an andere Nameserver weiterreicht. Im folgenden Programm wird aus einem Rechnernamen die IP-Adresse ermittelt. Dabei wird zuerst auf dem eigenen Rechner nachgesehen, ob die Adresse notiert ist (Datei */etc/hosts* unter Linux). Wenn das keinen Erfolg hat, wird der Nameserver gefragt. Da das im Hintergrund geschieht, muss man sich darum nicht weiter kümmern. Das Beispielprogramm *ihost* erwartet die Eingabe eines Rechnernamens und gibt die IP-Adresse aus, vorausgesetzt, die Verbindung des Rechners zum Internet ist hergestellt. Der Aufruf *ihost www.example.com* zeigt IP4- und IP6-Adressen auf dem Bildschirm an. Das Pro-

gramm leistet Ähnliches wie das Unix-Kommando *host*. Einzelheiten zu den verwendeten Funktionen `getaddrinfo()` und `getnameinfo()` erhält man auf einem Unix-System mit der Abfrage `man gethostbyname`.

Listing 15.1: Host-IP-Adresse ermitteln

```
// cppbuch/k15/host/ihost.cpp
/*
HINWEISE für die Benutzung unter Windows:

1. Bei der Compilation in einem MSDOS-Eingabeaufforderungs-
   Fenster die Optionen -lwsock32 und -lws2_32 setzen, z.B.
   g++ ihost.cpp -ihost.exe -lwsock32 -lws2_32
   (wird bei Aufruf von make automatisch erledigt)
2. Bei der Compilation mit einer IDE im Fenster Werkzeuge->
   Compiler-Optionen o.ä. ein Häkchen bei
   "Diese Befehle zur Linker-Kommandozeile hinzufügen"
   machen und im Feld darunter -lwsock32 -lws2_32 eintragen.
*/

#ifdef WIN32
#define _WIN32_WINNT 0x0501
#include<winsock2.h>
#include<ws2tcpip.h>
#else
#include<netdb.h>
#endif
#include<iostream>
#include<string>
using namespace std;

int main(int argc, char* argv[]) {
    if(argc == 1) {
        cout << "Gebrauch: ihost Rechnername\n";
    }
    else {
#ifdef WIN32
        WSADATA wsaData = {0};
        if(WSAStartup(MAKEWORD( 2, 2 ), &wsaData ) !=0) {
            cerr << "WSAStartup() gescheitert!" << endl;
            return 1;
        }
#endif
        struct addrinfo* result;
        int error = getaddrinfo(argv[1], NULL, NULL, &result);
        if(error == 0) {
            struct addrinfo* item = result;
            while(item) {
                char hostname[NI_MAXHOST] = "";
                // Namen extrahieren, wenn möglich
                error = getnameinfo(item->ai_addr, item->ai_addrlen,
                                   hostname, NI_MAXHOST, NULL, 0, 0);
                if(error == 0) {
```

```

        cout << hostname;
    }
    // IP-Adresse als String extrahieren, wenn möglich
    error |= getnameinfo(item->ai_addr, item->ai_addrlen,
                        hostname, NI_MAXHOST, NULL, 0,
                        NI_NUMERICHOST);

    if(error == 0) {
        cout << " " << hostname << endl;
    }
    else {
        cout << argv[1] << " : Fehler in getnameinfo()." << endl;
    }
    item = item->ai_next;
}
freeaddrinfo(result);
}
else {
    cout << argv[1] << " kann nicht ermittelt werden." << endl;
}
}
#ifdef WIN32
    WSACleanup();
#endif
}

```

Manchen Namen wie etwa `www.google.com` sind mehrere Websites und IP-Nummern zugeordnet, die in der `while`-Schleife ausgegeben werden.



Hinweis

Die obigen Makro-Abfragen steuern die Übersetzung abhängig vom Betriebssystem. Es ist mühselig und fehleranfällig, für jedes Programm entsprechende Makro-Abfragen zu konstruieren und zu testen, um die Portabilität zu gewährleisten, ganz zu schweigen von der schlechteren Lesbarkeit. Man könnte natürlich eigene Klassen und Header-Dateien zum Verstecken betriebssystemabhängiger Programmteile schreiben – aber warum das Rad zwei Mal erfinden? Aus diesem Grund verwende ich im Folgenden die Library *Boost.Asio*, die alle betriebssystemabhängigen Teile kapselt, sodass man sich darum nicht kümmern muss.

localhost

In den obigen Adressbeispielen sehen Sie den Namen *localhost*. Dieser Name ist dem eigenen Rechner zugeordnet; die IP-Adresse ist `127.0.0.1`. Beide sind nur intern gültig und keine von außen erreichbaren Adressen. Für TCP/IP-Anwendungen ist der eigene Rechner genauso wie ein entfernter Rechner ansprechbar, sodass diese Anwendungen rein lokal getestet werden können. Ein echtes Netzwerk ist dafür nicht notwendig. Im Betriebssystem ist eine Art virtuelle Netzwerkkarte eingerichtet, Loopback-Device genannt, die alle an diese Adresse gesendeten Daten verarbeitet.

Port

Ein Port ist eine Adresse innerhalb eines bestimmten Computers, die zwischen 0 und 65535 liegen kann. Jede Anwendung (Browser, ftp-Programm usw.) legt einen Port fest, damit sie feststellen kann, ob eine Sendung an sie gerichtet ist. Die Ports 0 bis 1023 sind für reservierte Netzwerkdienste vergeben, zum Beispiel wird Port 80 typischerweise für HTTP genutzt. Die reservierten Port-Nummern finden Sie in [Port]. Beispiel einer Adresse, die den Port 8080 enthält: `http://localhost:8080/`.

Datei

Eine weitere Verfeinerung der Adresse wird durch den Dateinamen gegeben. Wenn nur ein Hostname angegeben wird, ergänzt der Webserver im Fall des HTTPs meistens die Adresse durch Anhängen von `index.html` (im Webserver einstellbar). Beispiel einer Adresse, die eine Datei anspricht: `http://localhost:8080/tomcat-docs/servletapi/index.html`. In einer html-Datei können sogar bestimmte, durch eine Marke ausgezeichnete Stellen angesprungen werden: `http://www.provider.de/index.html#marke`.

15.3 Socket

Ein Socket (dt. etwa Steckdose) stellt das grundlegende Software-Element zur Verbindung zweier Programme auf verschiedenen Computern oder auch auf demselben Computer dar. Man unterscheidet zwischen dem Client-Socket des anfragenden Programms und dem Server-Socket des dienst anbietenden Programms. Die verwendeten Protokolle sind typischerweise TCP und UDP.

Es gibt einen Betriebssystemdienst, »daytime« genannt, der über Port 13 läuft und einen String mit dem aktuellen Datum und der aktuellen Zeit liefert. Der folgende TCP-Client kann diesen Dienst abfragen (daytime gibt es auch als UDP-Dienst).

Listing 15.2: Daytime-Client

```
// cppbuch/k15/tcpsocket/zeit/zeitclient.cpp
#include<iostream>
#include<cstdlib>
#include<boost/asio.hpp>
using namespace std;

int main(int argc, char* argv[]) {
    if(argc != 3) {
        cout << "Gebrauch: " << argv[0] << " <IP-Adresse> <port>" << endl;
        return 1;
    }
    boost::asio::io_service ioService;
    unsigned short port = atoi(argv[2]);
    boost::asio::ip::tcp::endpoint
        server(boost::asio::ip::address::from_string(argv[1]), port);
    boost::asio::ip::tcp::socket socket(ioService);
```

```

socket.connect(server);
cout << "Lokaler Endpunkt: " << socket.local_endpoint() << endl;
cout << "Verbindung mit " << socket.remote_endpoint()
    << " hergestellt." << endl;
const int SZ = 80;
char buf[SZ+1];
// Es werden maximal SZ Bytes gelesen:
size_t anzahlBytes = socket.read_some(boost::asio::buffer(buf, SZ));
buf[anzahlBytes] = '\0'; // Rest für Ausgabe ignorieren
cout << buf << endl;
}

```

Das Objekt `io_service` ist die Schnittstelle zu den Ein-/Ausgabefunktionen des Betriebssystems. Die Funktion `from_string()` wandelt die als Zeichenkette übergebene IP-Adresse in ein Boost.Asio-internes Format des Typs `address` um. Wie Sie statt der Adresse einen Namen verwenden können, sehen Sie in Abschnitt 15.3.3 unten. `server` ist ein `endpoint`-Objekt, das den entfernten Endpunkt der zu schaffenden Verbindung, gekennzeichnet durch Adresse und Port, repräsentiert. Das `socket`-Objekt versucht nach Erzeugung, mit `connect(server)` die Verbindung mit dem Ziel aufzunehmen. Wenn das gelingt, werden mit `socket.read_some()` Daten vom Server empfangen. Die Anzahl der gelesenen Bytes wird zurückgegeben. `read_some()` benötigt als Parameter eine Sequenz von Puffern. `buffer(buf, SZ)` gibt eine einelementige Sequenz zurück ([asio]). Nur zur Dokumentation werden der lokale und der entfernte Endpunkt ausgegeben. Um das Programm zu testen, wird es wie folgt aufgerufen: `zeitclient.exe <IP-Adresse> 13`.

Jedoch ist der Dienst »daytime« bei den meisten Rechnern abgeschaltet. Um Hackern möglichst wenig Angriffsfläche zu bieten, werden verständlicherweise nur die unbedingt notwendigen Ports freigegeben. In diesem Fall reagiert das obige Programm mit einer Exception »Connection refused«. Man kann aber zum Testen auf dem eigenen Betriebssystem den Dienst »daytime« aktivieren. Unter Windows sehen Sie bitte in der Windows-Hilfe nach, Stichwort: einfache TCP/IP-Dienste. Dort ist das Verfahren erläutert. Unter Linux kann der Netzwerkdienst mit einem Tool aktiviert werden (`yast2` bei SuSE-Linux). Der Aufruf

```
zeitclient.exe 127.0.0.1 13
```

zeigt dann die aktuelle Systemzeit an. Im Folgenden wird »daytime« mit einem eigenen Server realisiert, der dasselbe leistet wie der Dienst des Betriebssystems. Der Server kommuniziert mit dem obigen Clienten vom selben oder einem anderen Rechner aus. Weil die Portnummern bis 1023 reserviert sind, kommt eine größere Nummer zum Tragen.

Listing 15.3: Daytime-Server

```

// cppbuchi/k15/tcpsocket/zeit/zeitserver.cpp
#include<ctime>
#include<iostream>
#include<boost/asio.hpp>
#include<cstring>
#include<cstdlib>
using boost::asio::ip::tcp;
using namespace std;

int main(int argc, char* argv[]) {

```

```

if(argc != 2) {
    cout << "Gebrauch: " << argv[0] << " <port>" << endl;
    return 1;
}
boost::asio::io_service ioService;
unsigned short port = atoi(argv[1]);
tcp::acceptor acceptor(ioService, tcp::endpoint(tcp::v4(), port));
while(true) { // Abbruch mit Strg+C
    tcp::socket socket(ioService);
    cout << "lauschen an Port " << port
        << " ... (Abbruch mit Strg+C)" << endl;
    acceptor.accept(socket);
    cout << "Lokaler Endpunkt: " << socket.local_endpoint() << endl;
    cout << "Verbindung mit " << socket.remote_endpoint()
        << " hergestellt." << endl;
    time_t jetzt = time(NULL);
    const char* zeitstring = ctime(&jetzt);
    socket.write_some(boost::asio::buffer(zeitstring, strlen(zeitstring)+1));
}
}

```

Im Unterschied zum Clienten gibt es ein `acceptor`-Objekt, das für die Verbindungsaufnahme zuständig ist. In der `while`-Schleife wird ein `socket`-Objekt angelegt, das der Funktion `accept()` übergeben wird. `accept()` blockiert solange, bis ein Client die Verbindung aufnimmt. An der Anzeige von lokalem und dem entfernten Endpunkt und dem Vergleich bei Client und Server können Sie gut sehen, dass der Client sich einen beliebigen freien Port als lokalen Endpunkt nimmt. Dieser Port ist der entfernte Endpunkt des Servers. Der Server wird zum Beispiel mit

```
zeitserver.exe 3000
```

gestartet. Statt 3000 kann jede andere freie Portnummer gewählt werden. Eine IP-Adresse ist nicht notwendig, da der Server auf dem Port `PortNr` des Rechners lauscht, auf dem das Programm gestartet wurde. Der Client wird zweckmäßig in einem anderen Shell-Fenster mit `zeitclient.exe <IP-Adresse des Servers> 3000` gestartet. Falls der Client sich auf demselben Rechner befindet, ist 127.0.0.1 anzugeben. Nach Verbindungsaufbau wird die Zeichenkette mit der Zeitinformation erzeugt und mit `write_some()` an den Clienten gesendet. Der Destruktor des lokalen `socket`-Objektes schließt am Ende des Schleifenkörpers die Verbindung, sodass der Server beim nächsten Schleifendurchlauf bereit ist für eine neue eingehende Anfrage. Der Aufruf `zeitclient.exe 192.168.1.2 3000` ergab in einem kleinen Netzwerk die Ausgaben

Client:

```

Lokaler Endpunkt: 192.168.1.4:34034
Verbindung mit 192.168.1.2:3000 hergestellt.
Sun Jan 16 13:45:32 2011

```

Server:

```

lauschen an Port 3000 ... (Abbruch mit Strg+C)
Lokaler Endpunkt: 192.168.1.2:3000
Verbindung mit 192.168.1.4:34034 hergestellt.
lauschen an Port 3000 ... (Abbruch mit Strg+C)

```

Im Ergebnis sind beide lokalen Endpunkte miteinander verbunden, wie hier gut zu sehen.

Was tun bei zu kleinem Puffer?

Im obigen Clienten-Programm ist der Puffer auf 80 Zeichen begrenzt. Letztlich werden nur Bytes übertragen, und das können sehr viele sein – wie viele, ist vorab oft nicht bekannt. Am einfachsten ist es, wenn der Puffer groß genug gewählt wird; es kann aber sein, dass das nicht praktikabel ist. Außerdem wird das Problem damit nicht gelöst, sondern nur in Richtung größerer Zahlen verschoben. Es gibt aber die Möglichkeit, nach dem Lesevorgang den Puffer auszuwerten und dann den Lesevorgang zu wiederholen usw., bis es nichts mehr zu lesen gibt. Dies sei an einem kleinen Beispiel gezeigt, in dem die Auswertung des Puffers nur aus der Ausgabe mit `cout` besteht, aber natürlich ganz anders gestaltet werden kann.

```
// bei zu kleinem Puffer
do {
    boost::system::error_code error;
    anzahl = socket.read_some(boost::asio::buffer(buf, SZ), error);
    if(error == boost::asio::error::eof) {
        break;          // kein Fehler, normales Ende
    }
    else {
        // Puffer auswerten
        for(size_t i=0; i < anzahl; ++i) {
            cout << buf[i];
        }
        cout << endl;
    }
} while(anzahl > 0);
```

15.3.1 Bidirektionale Kommunikation

Im obigen Beispiel sendet der Server sofort nach Zustandekommen der Verbindung seine Antwort. Es ist aber auch möglich, dass er erst eine Nachricht des Clienten empfängt und in Abhängigkeit davon agiert. Die Kommunikation ist bidirektional. Um dieses zu zeigen, wird das obige Beispiel erweitert. Der Client fragt interaktiv das gewünschte Zeitformat ab und sendet es dem Server. Der Server wertet die Information aus und sendet die aktuelle Zeit im gewünschten Format zurück.

Listing 15.4: Client mit Formatwahl

```
// cppbuch/k15/tcpsocket/bidirectional/zeitclient2.cpp
#include<iostream>
#include<boost/asio.hpp>
using namespace std;

char formatWahl() {
    char ein;
    do {
        cout << "\nFormatwahl : 1 = Standard, 2 = Langform für Tag und Monat,\n"
              "3 = nur die Uhrzeit, 4 = Sekunden seit 1.1.1970, 0 = Programmende: ";
        cin >> ein;
```

```

    } while(ein < '0' || ein > '4');
    return ein;
}

int main(int argc, char* argv[]) {
    if(argc != 3) {
        cout << "Gebrauch: " << argv[0] << " <IP-Adresse> <port>" << endl;
        return 1;
    }
    boost::asio::io_service ioService;
    unsigned short port = atoi(argv[2]);
    boost::asio::ip::tcp::endpoint
        server(boost::asio::ip::address::from_string(argv[1]), port);
    char auftrag[2] = {0};
    while((auftrag[0] = formatWahl()) != '0') {
        boost::asio::ip::tcp::socket socket(ioService);
        socket.connect(server);
        socket.write_some(boost::asio::buffer(auftrag, 2)); // 2 Bytes
        const int SZ = 80;
        char buf[SZ+1];
        // Antwort lesen
        size_t anzahlBytes = socket.read_some(boost::asio::buffer(buf, SZ));
        buf[anzahlBytes] = '\0'; // Rest für Ausgabe ignorieren
        cout << buf << endl;
    }
}

```

Neu ist im Vergleich, dass nach dem `connect()` der Auftrag an den Server gesendet wird. Dementsprechend wartet der Server erst diese Nachricht ab, ehe er sie auswertet und die Antwort zurückschickt.

Listing 15.5: Server mit Formatwahl

```

// cppbuch/k15/tcpsocket/bidirectional/zeitserver2.cpp
#include <ctime>
#include<iostream>
#include<boost/asio.hpp>
using boost::asio::ip::tcp;
using namespace std;

const char* getZeitstring(const char* format) {
    const size_t MAX = 80;
    static char buf[MAX] = {0};
    time_t jetzt = time(NULL);
    // Sonderbehandlung für %s, das nicht jeder Compiler kennt
    if(format[1] == 's') {
        size_t pos = MAX;
        while(jetzt > 0) { // Zahl > 0 in C-String umwandeln
            buf[--pos] = jetzt % 10 + '0';
            jetzt /= 10;
        }
        return (buf + pos);
    }
    else { // Umwandlung je nach Format-String mit strftime()

```

```

        tm *z = localtime(&jetzt);
        strftime(buf, MAX, format, z);
        return buf;
    }
}

int main(int argc, char* argv[]) {
    if(argc != 2) {
        cout << "Gebrauch: " << argv[0] << " <port>" << endl;
        return 1;
    }
    boost::asio::io_service ioService;
    unsigned short port = atoi(argv[1]);
    tcp::acceptor acceptor(ioService, tcp::endpoint(tcp::v4(), port));
    while(true) { // Abbruch mit Strg+C
        tcp::socket socket(ioService);
        cout << "lauschen an Port " << port
            << " ... (Abbruch mit Strg+C)" << endl;
        acceptor.accept(socket);
        char daten[1] = {0};
        // 1 Byte lesen
        socket.read_some(boost::asio::buffer(daten, 1));
        const char* format = "%c"; // Vorgabe
        switch(daten[0]) {
            case '1' : format = "%c"; break;
            case '2' : format = "%A, %d. %B %Y, %X Uhr"; break;
            case '3' : format = "%X"; break;
            case '4' : format = "%s"; break;
        }
        const char* zeitstring = getZeitstring(format);
        socket.write_some(boost::asio::buffer(zeitstring, strlen(zeitstring)+1));
    }
}

```

15.3.2 UDP-Sockets

Das UDP-Protokoll realisiert die Versendung von Datenpaketen, Datagramme genannt. UDP bietet weder eine Garantie dafür, dass die Pakete in der richtigen Reihenfolge, noch dass sie überhaupt beim Empfänger ankommen. Deswegen ist im Gegensatz zu TCP/IP kein Quittungsmechanismus erforderlich, und der Transport geht schneller. Es gibt weitere Unterschiede in der Anwendung und Realisierung:

- Beim TCP wird eine zuverlässige Verbindung aufgebaut. Die Kommunikation kann mit Strömen (streams) realisiert werden, auch wenn die unterliegende Schicht (IP) auf einer verbindungslosen Datenpaketübertragung beruht. Die Reihenfolge von TCP-Datenpaketen ist wesentlich.
- UDP ist dagegen verbindungslos. Jedes Datagramm enthält die Zieladresse und ist nicht abhängig von anderen Datagrammen.
- Im Gegensatz zur Client-Server-Kommunikation ist die UDP-Verbindung symmetrisch – es gibt keinen ausgezeichneten Server bzw. Client, allenfalls auf der Ebene der Anwendung, wie im Beispiel unten.

Zum Vergleich seien Zeitclient und -server in der UDP-Variante gezeigt, leicht gekürzt, um Wiederholungen zu vermeiden:

Listing 15.6: UDP-Zeitclient

```
// Auszug aus cppbuch/k15/udpsocket/clientudp.cpp
int main(int argc, char* argv[]) {
    // ...
    boost::asio::io_service ioService;
    unsigned short port = atoi(argv[2]);
    boost::asio::ip::udp::endpoint
        server(boost::asio::ip::address::from_string(argv[1]), port);
    char auftrag[2] = {0};
    while((auftrag[0] = formatWahl()) != '0') {
        boost::asio::ip::udp::socket socket(ioService);
        socket.open(boost::asio::ip::udp::v4());
        socket.send_to(boost::asio::buffer(auftrag, 2), server);
        const int SZ = 80;
        char buf[SZ+1];
        // Antwort lesen
        boost::asio::ip::udp::endpoint hier;
        size_t anzahlBytes =
            socket.receive_from(boost::asio::buffer(buf, SZ), hier);
        buf[anzahlBytes] = '\0'; // Rest für Ausgabe ignorieren
        cout << buf << endl;
    }
}
```

Nicht nur der Namespace `udp` statt `tcp` ist anders, auch die Methodennamen: `send_to()` statt `write_some()` und `receive_from()` statt `read_some()`. Der `connect()`-Aufruf wird durch `open()` ersetzt. Ansonsten ist die Struktur recht ähnlich.

Listing 15.7: UDP-Zeitserver

```
// Auszug aus cppbuch/k15/udpsocket/serverudp.cpp
int main(int argc, char* argv[]) {
    // ...
    boost::asio::io_service ioService;
    unsigned short port = atoi(argv[1]);
    udp::socket socket(ioService, udp::endpoint(udp::v4(), port));
    while(true) { // Abbruch mit Strg+C
        cout << "lauschen an Port " << port
            << " ... (Abbruch mit Strg+C)" << endl;
        boost::asio::ip::udp::endpoint entfernt;
        char daten[256] = {0};
        socket.receive_from(boost::asio::buffer(daten), entfernt);
        const char* format = "%0c";
        // nur erstes Byte ist relevant
        switch(daten[0]) { // wie oben
            case '1' : format = "%0c"; break;
            case '2' : format = "%0A, %0d. %0B %0Y, %0X Uhr"; break;
            case '3' : format = "%0X"; break;
            case '4' : format = "%0s"; break;
        }
    }
}
```

```

const char* zeitstring = getZeitstring(format);
socket.send_to(boost::asio::buffer(zeitstring, strlen(zeitstring)+1),
               entfernt);
}
}

```

15.3.3 Atomuhr mit UDP abfragen

Manche Dienste gibt es ausschließlich in der UDP-Version, zum Beispiel den Standarddienst zum Verteilen der Zeitinformation. Es gibt hochgenaue Atomuhren, deren Zeitinformation über eine geschichtete Serverstruktur dem Internet zur Verfügung gestellt wird (Schicht 1, Schicht 2, andere). Nach Anfordern einer Zeitinformation ist ein weiteres Halten der Verbindung nicht erforderlich; deshalb reicht UDP. Und wenn mal ein Paket verlorengeht, wird eben ein neues angefordert, gegebenenfalls von einem anderen Server. Der Grund für diesen Abschnitt liegt aber nicht in dieser Vorrede, sondern auf einem anderen Aspekt, der bisher nicht zum Tragen kam: Die Byte-Reihenfolge (englisch *byte order*) innerhalb eines Integer-Werts. Es gibt zwei wesentliche Varianten:

- *Big-Endian*: Das Byte mit den *höchstwertigen* Bits wird an der kleinsten Speicheradresse gespeichert.
- *Little-Endian*: Das Byte mit den *niederwertigen* Bits wird an der kleinsten Speicheradresse gespeichert.

Die Byte-Reihenfolge kann je nach Prozessortyp verschieden sein: Motorola verwendet Big-Endian, Intel hingegen Little-Endian. Bei der Übertragung von Daten ist nicht bekannt, auf welchen Rechnertyp man trifft, der die Daten interpretieren soll. Um die Kommunikation zwischen verschiedenen Computern zu ermöglichen, schreibt das Internet-Protokoll die Reihenfolge, *Network Byte Order* genannt, vor, und zwar Big-Endian. Die auf einem Rechner verwendete Reihenfolge heißt *Host Byte Order*. Sie kann mit der Network Byte Order übereinstimmen, muss es aber nicht. Zur Umwandlung gibt es die Funktionen

```

htonl(u32) : Host to Network byte order-Umwandlung (u32 = 32-Bit unsigned)
htons(u16) : Host to Network byte order-Umwandlung (u16 = 16-Bit unsigned)
ntohl(u32) : Network to Host byte order-Umwandlung (u32 = 32-Bit unsigned)
ntohs(u16) : Network to Host byte order-Umwandlung (u16 = 16-Bit unsigned)

```

Der letzte Buchstabe des Funktionsnamens steht für long bzw. short. Falls Network Byte Order und Host Byte Order übereinstimmen, tun diese Funktionen nichts. Man soll sie dennoch verwenden, weil dann der Programmcode unabhängig von der Host Byte Order geschrieben werden kann. Netzwerkadressen und Port müssen in der Network Byte Order übertragen werden. Dass man das in den obigen Beispielen nicht sieht, liegt daran, dass die Boost.Asio Library im Hintergrund dafür sorgt. Um jedoch die Zeit bei einem Zeitserver abzufragen, muss ihm ein Header im Big-Endian-Format gesendet werden, wie im »Simple Network Time Protocol« (RFC 4330, [\[NTP\]](#)) festgelegt. Das heißt nicht nur, dass die Anfrage entsprechend codiert sein muss, sondern es ist auch die Antwort in die Host Byte Order zu konvertieren. Beides ist im Beispiel zu sehen, das anschließend im Einzelnen erläutert wird.

Listing 15.8: Atomuhr mit UDP abfragen

```

1 // cppbuch/k15/udpsocket/ntp.cpp
2 #include<iostream>
3 #include<boost/asio.hpp>
4 #include<cassert>
5 #include<ctime>
6 using boost::asio::ip::udp;
7
8 int main() {
9     boost::asio::io_service ioService;
10    udp::socket socket(ioService);
11    assert(sizeof(size_t) == 4);
12    size_t buf[12] = {0};
13    const size_t BUFSIZE = sizeof(buf);
14    buf[0] = htonl ( ( 3 << 27) | (3 << 24)); // siehe RFC 4330 [NTP]
15    udp::resolver resolver(ioService);
16    udp::resolver::query query(udp::v4(), "europe.pool.ntp.org", "123");
17    udp::endpoint zeitserver = *resolver.resolve(query);
18    socket.open(boost::asio::ip::udp::v4());
19    socket.send_to(boost::asio::buffer(buf, BUFSIZE), zeitserver);
20
21    boost::asio::ip::udp::endpoint hier;
22    socket.receive_from(boost::asio::buffer(buf, BUFSIZE), hier);
23    time_t secs = ntohl(buf[8]) - 2208988800u;
24    tm *z = localtime(&secs);
25    double secfrac = (double)ntohl(buf[9])/4294967296.0;
26    std::cout << z->tm_mday << '.' << z->tm_mon + 1 << '.'
27                << z->tm_year + 1900 << " "
28                << z->tm_hour << ':' << z->tm_min << ':'
29                << secfrac + z->tm_sec << std::endl;
30 }

```

Erläuterungen zu den einzelnen Zeilen:

- 11 Das Programm funktioniert nur auf einem System, in dem ein `int` bzw. `size_t` 32 Bits = 4 Bytes lang ist. Wenn nicht, muss das Programm angepasst werden. Bei einem neueren Compiler könnte hier das auf Seite 134 beschriebene `static_assert` eingesetzt werden, um eine entsprechende Meldung nicht erst nach Start des Programms, sondern schon während der Compilation zu erhalten.
- 12 `buf` nimmt erst die Anfrage und nachher die Antwort auf. Nach [NTP] sind mindestens 12 32-Bit-Felder vorgesehen.
- 14 Version und Modus (Details siehe [NTP]) werden entsprechend codiert und vor Zuweisung an `buf[0]` in die Network Byte Order umgewandelt. Die anderen Felder haben ihre Bedeutung, können jedoch für eine minimale Anfrage leer bleiben.
- 15 In den vorangegangenen Abschnitten dieses Kapitels wird mit der IP-Adresse operiert. Die Angabe des Hostnamens ist aber bequemer und aus folgenden Gründen oft besser geeignet als die Angabe einer festen IP-Adresse:
 - Einem Namen können mehrere IP-Adressen zugewiesen sein, um die Last zu verteilen.

- Ein Server kann umgezogen sein, d.h. er hat seinen Namen behalten, aber seine IP-Adresse hat sich geändert.

Das Objekt `resolver` löst Namen und Port in einen Iterator auf einen Endpunkt auf und erledigt damit eine ähnliche Aufgabe wie das Programm auf Seite 476. Der `resolver` gibt immer einen gültigen Iterator zurück – oder er wirft eine Exception.

- 16 Das Protokoll für den Zeitdienst ist fest dem Port 123 zugeordnet. Die angegebene Adresse *europe.pool.ntp.org* ist ein Server, an den viele weitere angeschlossen sind und der die Anfrage an einen von diesen weiterleitet (Schicht »andere« im Sinne der auf Seite 485 genannten Schichten). Es ist sinnvoll, diese Adresse zu benutzen, um die Haupt-Zeitserver zu entlasten. Einer davon ist der zur Schicht 1 gehörende Server *ptbtime1.ptb.de* der Physikalisch-Technischen Bundesanstalt in Braunschweig, deren Atomuhren alle Funkwecker und Bahnhofsuhren in Deutschland steuern.
- 17 Der Iterator wird dereferenziert und ergibt den Endpunkt.
- 18 Der Socket wird für das IPv4-Protokoll geöffnet.
- 19 Die Anfrage wird an den Zeitserver verschickt.
- 22 Die Antwort wird gelesen.
- 23 `buf[8]`, in die Host Byte Order umgewandelt, enthält die Anzahl der seit dem 1.1.1900 verstrichenen Sekunden. Weil die Anfangszeit aller Computer-Systeme sich auf den 1.1.1970 bezieht, muss die Differenz, also 70 Jahre in Sekunden, abgezogen werden. Nach Angabe des RFC 4330-Autors sind das 2208988800 Sekunden, wie auf seiner Seite (<http://www.cis.udel.edu/~mills/y2k.html>) nachgelesen werden kann.
- 24 Die berechnete Anzahl der Sekunden wird in eine `tm`-Struktur umgewandelt, wie von Seite 335 bekannt.
- 25 `buf[9]`, in die Host Byte Order umgewandelt, enthält den Sekundenbruchteil als 32-Bit-Feld, weswegen durch $2^{32} = 4294967296$ geteilt wird. Das letzte Bit entspricht etwa 233 Pico-Sekunden. Die extrem hohe Auflösung ist rein hypothetisch, weil allein durch die Netzübertragung eine Genauigkeit von weniger als einer Sekunde kaum erreichbar ist. Der Sekundenbruchteil spielt nur dann eine wesentliche Rolle, wenn die Übertragungszeit herausgerechnet wird (siehe unten).
- 26 Datum und Uhrzeit werden ausgegeben.
- 29 Zum Wert der Sekunde wird der errechnete Bruchteil addiert.

Der Kürze halber wird nur ein Teil der Nachricht ausgewertet. Daher nur als Ergänzung die Information, dass sich mit Hilfe der anderen Einträge in `buf` die Dauer, wie lange die Übertragung gedauert hat, sowie der Unterschied in den Uhren von Client und Zeitserver berechnen lassen, wenn die eigene Zeit mitgeschickt wird. Wenn Sie mehr darüber wissen möchten, empfehle ich Ihnen <http://www.ntp.org> sowie [NTP]. Bitte beachten Sie bei der Nutzung der Zeitdienste die Kapitel 9 und 10 von [NTP].

15.4 HTTP

HTTP ist ein weitverbreitetes Übertragungsprotokoll auf Anwendungsebene für Hypermedia-Informationen. Es ist ein zustandsloses Protokoll, eine zweite HTTP-Anfrage weiß also nichts von der ersten, weswegen mehrere logisch zusammengehörige Informationen, die mit HTTP hin und her transportiert werden, besonderer Erkennungsmechanismen bedürfen. Ein Beispiel dafür ist ein Online-Kauf, bei dem sich erst nach und nach der virtuelle Einkaufskorb füllt und am Ende durch Angabe der Kreditkartennummer bezahlt wird. HTTP ist auch für viele andere Dinge wie zum Beispiel Nameserver nutzbar. Mit HTTP können verschiedene Rechner miteinander Datenformate »aushandeln«, und es gibt eine Menge Fehlercodes. Es benutzt verschiedene Methoden zum Datentransfer, von denen GET und POST die bekanntesten sind. Die HTTP/1.1-Spezifikation RFC 2616 ist über die Website des World Wide Web Consortiums (W3C) oder direkt von der Internet Engineering Task Force erhältlich [Fiel]. HTTP ist ein Anfrage-Antwort-Protokoll. Ein Client sendet eine Anfrage an einen Server. Die Anfrage besteht aus einer Zeile, die dem Server sagt, mit welcher Methode er die Anfrage auswerten soll (wie GET oder POST), einem URI und der Angabe des Protokolls mit Version, heutzutage HTTP/1.1. Anschließend folgt ein Kopfteil (englisch *header*) mit aus Attribut/Wert-Paaren bestehenden Zeilen zur Beschreibung des nachfolgenden Inhalts. Zum Beispiel besagen die Zeilen

```
Content-Length: 400
Content-Type: text/plain; charset=ISO-8859-1
```

dass der nachfolgende Inhalt 400 Bytes lang ist und aus schlichtem Text des Zeichensatzes ISO-8859-1 besteht. Die wichtigsten HTTP-Methoden sind GET, POST und HEAD. HEAD unterscheidet sich von GET darin, dass nur der Header, aber nicht der Inhalt vom Server zurückgesendet wird. Das erlaubt es dem Clienten, die Informationen auszuwerten und in Abhängigkeit davon die GET-Anfrage zu starten. Es könnte ja sein, dass die Content-Length so groß ist, dass bei einem mobilen Gerät (Handy, PDA) auf ein Laden verzichtet werden soll. Der Content-Type (MIME-Typ, siehe Glossar) sagt, wie der Inhalt zu interpretieren ist. Zum Beispiel steht `gzip` für einen zip-komprimierten Inhalt, und `application/x-www-form-urlencoded` heißt, dass Umlaute, Sonderzeichen und Leerzeichen durch andere Symbole ersetzt wurden. Die Regeln sind in [URI] festgelegt und relativ einfach: Alle alphanumerischen Zeichen und die Sonderzeichen `-_.!~*'()` bleiben unverändert. Alle anderen Zeichen werden hexadezimal codiert, wobei ein `%`-Zeichen vorangestellt wird. Nach einer Leerzeile folgt der Inhalt, falls erforderlich. Alle Zeilen enden mit CRLF bzw. `\r\n`, nicht nur mit LF bzw. `\n`.

Der Server antwortet in einem ähnlich strukturierten Format. Es enthält unter anderem einen Antwort-Code. `200 OK` ist der Code, wenn die Anfrage erfolgreich beantwortet werden kann. Der Code `404 Not Found` ist fast jedem Internet-Benutzer bekannt. Der Inhalt der Antwort ist typischerweise ein HTML-Text, wenn der Client ein Browser ist. Der Ablauf einer Interaktion zwischen HTTP-Client und HTTP-Server ist:

1. Server wartet auf eingehende HTTP-Abfrage.
2. Client erzeugt URL `http://... .`
3. Client baut TCP-Verbindung auf.

4. Server akzeptiert Verbindungswunsch.
5. Client sendet HTTP-Nachricht und fordert spezifizierte Ressource an.
6. Server verarbeitet die Anfrage (z.B. Datenbankabfrage und Erzeugung einer HTML-Seite mit dem Ergebnis).
7. Server sendet die geforderte Ressource (z.B. HTML-Seite oder Datei).
8. Client verarbeitet die Antwort.
9. Client und/oder Server schließen die TCP-Verbindung.

15.4.1 Verbindung mit GET

Die GET-Methode des HTTP enthält die vollständige Anforderung im URI. Die Herstellung der Verbindung löst im Server die gewünschte Reaktion aus, das Senden der Daten. Jeder hat wohl schon mal sehr lange URIs im Adressfenster eines Browsers gesehen. Häufig sind es URIs, die Parameter enthalten. Dabei folgt der eigentlichen Adresse entweder ein Pfad (Verzeichnis und Dateiname) oder eine Liste von Parametern, deren Beginn durch ein `?` markiert wird. Beispiele:

```
www.hs-bremen.de/__assets/images/logo.gif
www.google.com/webhp?hl=de&q=c%2B%2B
```

In der zweiten Zeile markiert in der Google-Festlegung `?hl=de` die Sprache, `de` steht für Deutsch. Vor dem `=`-Zeichen steht der Name des Parameters, danach der Wert. Weitere GET-Parameter können folgen, abgetrennt durch jeweils ein `&`-Zeichen, zum Beispiel `&q=Wert` für Query (Anfrage). Die Parameter müssen entsprechend dem oben schon erwähnten MIME-Typ `application/x-www-form-urlencoded` codiert sein. `c%2B%2B` ist die URL-codierte Form von `c++`. Der erste Schrägstrich nach dem Host-Namen und alles, was danach kommt, heißt *Pfad*. GET muss ein Pfad übergeben werden; falls er leer ist, muss / erhalten.

Im folgenden Beispiel wird gezeigt, wie eine GET-Anfrage und die Auswertung der Antwort realisiert werden. Die Klasse `ClientAnfrage` zerlegt eine übergebene Webadresse in Host-Name und Pfad und sorgt für das Absenden der Anfrage. Falls der Pfad einen Dateinamen enthält, wird er für die Antwort verwendet, ansonsten wird die Antwort in der Datei `antwort.html` abgelegt. Weil es sein kann, dass auch binäre Daten zurückgesendet werden, etwa wenn eine Bilddatei angefordert wird, geht das Programm zweistufig vor. Es zeigt die Statusinformation und den Header der Antwort im Klartext an, der Inhalt (englisch *content*) wird jedoch in eine Datei geschrieben. Das Programm eignet sich damit zum Download einer Datei. Die verschiedenen möglichen Codes für den Status sind in [Fiel], Abschnitt 10, definiert. Der Kürze wegen analysiert `main()` nur die Fehlerklassen:

Listing 15.9: GET-Abfrage

```
// cppbuch/k15/http/get/main.cpp
#include<iostream>
#include"ClientAnfrage.h"
#include"AntwortAuswerter.h"
using namespace std;

int main(int argc, char* argv[]) {
    if(argc != 2) {
```

```

    cout << "Gebrauch: " << argv[0]
        << " WWW-Adresse (ohne http:// eingeben)" << endl;
    return 1;
}
ClientAnfrage clientAnfrage(argv[1]);
AntwortAuswerter antwortAuswerter(clientAnfrage);
clientAnfrage.send();
antwortAuswerter.receive();
cout << "Header:\n" << antwortAuswerter.getHeader() << endl;
cout << "Status (200=OK): " << antwortAuswerter.getStatus() << " ";
string stat;
switch(antwortAuswerter.getStatus()/100) {
case 2 : stat = "erfolgreich"; break;
case 3 : stat = "Umleitung u.a."; break;
case 4 : stat = "Clienten-Fehler"; break;
case 5 : stat = "Server-Fehler"; break;
default: stat = "darf nicht vorkommen"; break;
}
cout << stat << endl;
if(antwortAuswerter.getStatus() >= 300) {
    cout << "siehe Datei " << clientAnfrage.getDateiname() << endl;
}
if(antwortAuswerter.getHeader().find("chunked") != string::npos) {
    cout << "Transfer-Encoding: chunked nicht implementiert\n"
        << "nur 1 Chunk gelesen, siehe "
        << clientAnfrage.getDateiname() << endl;
}
}
}

```

Zur letzten Anweisung: Es gibt den Modus, die Antwort häppchenweise (englisch *chunked*) zu versenden. Der Kürze halber ist dieser Modus nicht implementiert, wie auch andere Dinge nicht, wie etwa bei Umleitung die neue Adresse anzusprechen. Alle in [Fiel] aufgeführten Möglichkeiten zu realisieren, würde den hier sinnvollen Umfang sprengen. Der Konstruktor der folgenden Klasse `ClientAnfrage` ermittelt aus der übergebenen Web-Adresse den Host-Namen und den Pfad und stellt daraus gemäß [Fiel], Abschnitt 5.1.2, die GET-Anfrage zusammen. Die Anfrage muss mit einer Leerzeile enden. »Connection: close« bedeutet, dass der Server die Verbindung nach Übersenden der Nachricht schließen kann ([Fiel], Abschnitt 14.10). Der Konstruktor legt unter anderem den Socket an. Damit das `AntwortAuswerter`-Objekt darauf (und andere Informationen) zugreifen kann, bekommt es in `main()` das `ClientAnfrage`-Objekt übergeben.

Listing 15.10: Klasse `ClientAnfrage`

```

// cppbuch/k15/http/get/ClientAnfrage.h
#ifndef CLIENTANFRAGE_H
#define CLIENTANFRAGE_H
#include<string>
#include<iostream>
#include<boost/asio.hpp>
#include<cstring> // strchr()
#include<cctype> // isalnum()

```

```

namespace {
    std::string urlencode(const std::string& s) {
        std::string ergebnis;
        for(size_t i = 0; i < s.length(); ++i) {
            char c = s[i];
            if(isalnum(c) || strchr("_-!~*'()", c)) { // RFC 3986, 2.3
                ergebnis += c;
            }
            else {
                ergebnis += '%';
                size_t ic = c/16;
                if(ic < 10) ic += 48; // 0..9
                else      ic += 55; // A..F
                ergebnis += (char)ic;
                ic = c%16;
                if(ic < 10) ic += 48; // 0..9
                else      ic += 55; // A..F
                ergebnis += (char)ic;
            }
        }
        return ergebnis;
    }
}

class ClientAnfrage {
public:
    ClientAnfrage(const std::string& wwwseite)
        : socket(ioService), gesendet(false) {
        std::string pfad;
        std::string hostname;
        size_t schraegstrichPosition = wwwseite.find("/");
        if(schraegstrichPosition == std::string::npos) { // nicht vorhanden
            hostname = wwwseite;
            pfad = "/";
        }
        else { // extrahieren:
            hostname = wwwseite.substr(0, schraegstrichPosition);
            pfad = wwwseite.substr(schraegstrichPosition);
        }
        // neu
        std::string getQuery("");
        size_t queryPosition = pfad.find("?");
        if(queryPosition != std::string::npos) { // vorhanden
            // Query url-encodieren:
            getQuery = "?" + urlencode(pfad.substr(queryPosition+1));
            pfad = pfad.substr(0, queryPosition);
        }
        anfrage = "GET "; // s. RFC 2616 Kap. 5.1.2
        anfrage += pfad + getQuery + " HTTP/1.1\r\nHost: " + hostname
            + "\r\nAccept: */*\r\nConnection: close\r\n\r\n"; // Extra-Leerzeile

        dateiname = "antwort.html"; // Vorgegebener Dateiname für dieses Programm
        boost::asio::ip::tcp::resolver resolver(ioService);
    }
};

```



```

    boost::asio::ip::tcp::resolver::query query(hostname, "http");
    server = *resolver.resolve(query);
}

void send() {
    // Anfrage senden
    socket.connect(server);
    socket.write_some(boost::asio::buffer(anfrage.c_str(), anfrage.length()));
    gesendet = true;
}

boost::asio::ip::tcp::socket& getSocket() {
    return socket;
}

const std::string& getDateiname() const {
    return dateiname;
}

bool istGesendet() const {
    return gesendet;
}

private:
    boost::asio::io_service ioService;
    boost::asio::ip::tcp::socket socket;
    boost::asio::ip::tcp::endpoint server;
    std::string anfrage;
    std::string dateiname;
    bool gesendet;
};
#endif

```

Ein AntwortAuswerter-Objekt liest die Antwort mit `receive()` und wertet sie aus. Der Header ist stets durch eine extra Leerzeile vom Inhalt getrennt, was die Erkennung vereinfacht. Oft wird die Menge der zu übertragenden Daten im Header als »Content-Length« übertragen. Das ist aber nicht garantiert und in manchen Fällen sinnlos (wie etwa bei »Transfer-Encoding: chunked«). Deshalb ist es am besten, wie auf Seite 481 vorzugehen: einen Puffer fixer Kapazität anzulegen und wiederholt zu lesen, bis die `error`-Variable das Ende des Empfangs meldet.

Listing 15.11: Klasse AntwortAuswerter

```

// cppbuch/k15/http/get/AntwortAuswerter.h
#ifndef ANTWORTAUSWERTER_H
#define ANTWORTAUSWERTER_H
#include<string>
#include<cstring>
#include<cstdlib>
#include<boost/asio.hpp>
#include<fstream>
#include"ClientAnfrage.h"

class AntwortAuswerter {
public:
    AntwortAuswerter(ClientAnfrage& r)
        : clientAnfrage(r), socket(r.getSocket()), dateiname(r.getDateiname()),

```

```

        header("noch undefiniert"), status(-1) {
    }

    void receive() {
        if(!clientAnfrage.istGesendet()) {
            clientAnfrage.send();
        }
        std::ofstream ausgabe; // zur Speicherung des Contents
        ausgabe.open(dateiname.c_str(), std::ios::binary | std::ios::out);
        size_t anzahl;
        const size_t SIZE = 4096;
        char buf[SIZE];
        do {
            boost::system::error_code error;
            anzahl = socket.read_some(boost::asio::buffer(buf, SIZE), error);
            if(error == boost::asio::error::eof) {
                break; // kein Fehler, normales Ende
            }
            else { // Puffer auswerten
                if(status == -1) {
                    std::string tmp(buf, anzahl);
                    size_t headerEnde = tmp.find("\r\n\r\n");
                    header = tmp.substr(0, headerEnde);
                    int startInhalt = headerEnde + 4; // CRLF überspringen
                    // restliche Bytes wegschreiben
                    int rest = (int)anzahl - startInhalt;
                    if(rest > 0) {
                        ausgabe.write(buf + startInhalt, rest);
                    }
                    status = atoi(buf + header.find(' ') + 1);
                }
                else {
                    ausgabe.write(buf, anzahl);
                }
            }
        } while(anzahl > 0);
        ausgabe.close();
    }

    int getStatus() const {
        return status;
    }

    const std::string& getHeader() const {
        return header;
    }
private:
    ClientAnfrage& clientAnfrage;
    boost::asio::ip::tcp::socket& socket;
    const std::string& dateiname;
    std::string header;
    int status;
};
#endif

```

15.4.2 Verbindung mit POST

Auch POST ist eine Methode des HTTP. Im Unterschied zu GET ist die vollständige Anforderung nicht im URI enthalten, sondern sie wird im Inhaltsteil der Anfrage übertragen. Dadurch ist die Anforderung nicht in der Browser-Adresszeile sichtbar. Es können letztlich beliebige Daten gesendet werden. Eine häufige Anwendung ist die Übertragung von Formulareinträgen, wie Vorname, Nachname usw. Der Inhaltsteil besteht dann üblicherweise aus Attribut/Wert-Paaren der Form `attribut=wert`. Mehrere Paare werden bei der Übertragung durch das `&`-Zeichen getrennt. Die Codierung der Werte ist wie von oben bekannt `application/x-www-form-urlencoded`.

15.5 Mini-Webserver

Um das Zusammenspiel zwischen Internet-Browser und Server zu zeigen, beschreibe ich abschließend einen Mini-Webserver, der zwei einfache Aufträge erledigen kann: Anzeige des aktuellen Datums und Anzeige einer Begrüßung nach Eingabe des Vornamens. Der Funktionsumfang ist zwar minimal verglichen mit üblichen Webservern, dafür ist er aber sehr klein und kann die wesentlichen Funktionen demonstrieren. Wenn der Server mit `server.exe 9090` gestartet und der Browser auf die Webadresse gerichtet wird, ist im Browser die Abbildung 15.1 zu sehen. Bei einer lokalen Anwendung ist die Adresse `http://localhost:9090/`. 9090 ist die Portnummer.



Abbildung 15.1: Erscheinungsbild des Mini-Webservers im Browser

Klicken auf den »Datum«-Button zeigt Datum und Uhrzeit an. Eingabe eines Namens und Bestätigung mit der Return- bzw. Enter-Taste lässt eine Begrüßung erscheinen. Gleich-

zeitig protokolliert der Server die eingehenden Anfragen und die ausgehenden Header. Das `main()`-Programm erzeugt den Server und startet ihn mit `run()`:

Listing 15.12: Hauptprogramm des Servers

```
// cppbuch/k15/http/server/main.cpp
#include<iostream>
#include<cstdlib>
#include"Server.h"

int main(int argc, char* argv[]) {
    if(argc != 2) {
        std::cout << "Gebrauch: " << argv[0] << " <port>" << std::endl;
        return 1;
    }
    unsigned short port = atoi(argv[1]);
    Server server(port, "web");
    server.run();
}
```

Dem Server wird das Verzeichnis *web* übergeben; es enthält die Datei *index.html* und das angezeigte Bild. *index.html* enthält zwei Formular-Tags:

Listing 15.13: *index.html*

```
<html>
<head><title>C++ MiniWebServer</title></head>
<body>
  <br>
  <hr>
  <p>Auftrag mit Button-Klick absenden:
  <form action="" method="GET">
    <input class="button" type="submit" name="zeitabfrage" size="12"
      value="Datum"/>
  </form>
  </p>
  <p>Vornamen eingeben und mit ENTER best&auml;tigen:
  <form action="" method="GET">
    <input name="begruessung" type="text" value="" maxlength="40"/>
  </form>
  </p>
</body>
</html>
```



HTML

Einige wenige Basiskenntnisse in HTML werden für das Beispiel vorausgesetzt. Sollten diese Kenntnisse nicht vorhanden sein, bitte ich Sie, sich bei <http://de.selfhtml.org/> zu informieren, der besten deutschsprachigen Webseite zu HTML.

Die Klasse `Server` verwaltet die Verbindungen. Eine Abbruchmöglichkeit außer der Eingabe von `Strg+C` ist nicht vorgesehen. In der Methode `run()` wird das Objekt `conn` vom

Typ `HttpConnection` erzeugt. `conn.accept()` wartet auf eine eingehende Verbindung. Ist sie zustande gekommen, wird `conn.operator()()` ausgeführt.

Listing 15.14: Klasse `Server`

```
// cppbuch/k15/http/server/Server.h
#ifndef SERVER_H
#define SERVER_H
#include "HttpConnection.h"

class Server {
public:
    Server(int p, const std::string& v)
        : port(p), verzeichnis(v),
          acceptor(ioService,
                  boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(),
                                                  port)) {

    }

    void run() {
        while(true) { // Abbruch mit Strg+C
            std::cout << "lauschen an Port " << port
                      << " ... (Abbruch mit Strg+C)" << std::endl;
            HttpConnection conn(acceptor, verzeichnis);
            conn.accept();      // warten auf eingehende Verbindung
            conn();              // bearbeiten
        }
    }
private:
    int port;
    const std::string verzeichnis;
    boost::asio::io_service ioService;
    boost::asio::ip::tcp::acceptor acceptor;
};
#endif
```

Die vom `Server`-Objekt aufgerufene Funktion `operator()()` liest die Daten ein und analysiert, ob es sich um eine Anfrage oder um eine gewünschte Datei handelt. Im ersten Fall wird die Aufgabe dem `AnfrageHandler` übergeben, im zweiten Fall dem `DateiHandler`:

Listing 15.15: Klasse `HttpConnection`

```
// cppbuch/k15/http/server/HttpConnection.h
#ifndef HTTPCONNECTION_H
#define HTTPCONNECTION_H
#include <iostream>
#include <string>
#include <boost/asio.hpp>
#include "AnfrageHandler.h"
#include "DateiHandler.h"

class HttpConnection {
public:
    HttpConnection( boost::asio::ip::tcp::acceptor& a, const std::string& v)
```

```

    : acceptor(a), socket(a.get_io_service()), verzeichnis(v) {
}

void accept() { acceptor.accept(socket); }

void operator()() {
    const size_t BUFSIZE = 1024;
    char daten[BUFSIZE+1] = {0};
    // Anfrage lesen
    size_t anzahl = socket.read_some(boost::asio::buffer(daten, BUFSIZE));
    daten[anzahl] = '\0';
    std::cout << daten << std::endl;
    std::string header(daten);
    size_t leerzeichen = header.find(' ');
    // Fehlermeldung bei falschem Protokoll
    if(header.substr(0, leerzeichen) != "GET") {
        // Header zusammenbauen und senden
        HttpResponse httpResponse(400);
        httpResponse.addToHeader("Content-Type", "text/html");
        std::string msg = httpResponse.getHTMLMessage(
            "Protokoll nicht implementiert!");
        httpResponse.addToHeader("Content-Length", i2string(msg.length()));
        httpResponse.sendHeader(socket);
        // msg senden (html-Seite)
        socket.write_some(boost::asio::buffer(msg.c_str(), msg.length()));
        return; // Abbruch!
    }
    size_t leerzeichen2 = header.find(' ', leerzeichen+1);
    std::string pfad = header.substr(leerzeichen+1,
                                    leerzeichen2-leerzeichen-1);
    // Gibt es eine Anfrage oder ist eine Datei gewünscht?
    size_t qpos = pfad.find('?');
    if(qpos != std::string::npos) { // Anfrage
        AnfrageHandler ah(socket, pfad.substr(qpos+1));
        ah.process();
    }
    else { // Datei
        std::string dateiname = verzeichnis + pfad;
        if(pfad == "/" || pfad == "/favicon.ico") {
            dateiname = verzeichnis + "/index.html";
        }
        DateiHandler dh(socket, dateiname);
        dh.process();
    }
    socket.close();
}

private:
    boost::asio::ip::tcp::acceptor& acceptor;
    boost::asio::ip::tcp::socket socket;
    std::string verzeichnis;
};
#endif

```

Die Klasse `HttpResponse` bereitet den HTTP-Header vor und sendet ihn gegebenenfalls. Sie wird oben in der Reaktion auf ein nicht implementiertes Protokoll und auch in den nachfolgenden Klassen `AnfrageHandler` und `DateiHandler` verwendet.

Listing 15.16: Klasse `HttpResponse`

```
// cppbuch/k15/http/server/HttpResponse.h
#ifndef HTTPRESPONSE_H
#define HTTPRESPONSE_H
#include<string>
#include<iostream>

namespace {
    std::string i2string(int i) { // wandelt int in einen String um
        // Alternative: boost::lexical_cast aus Abschnitt 24.1.3
        std::string ergebnis;
        if(i == 0) {
            ergebnis = "0";
        }
        else {
            bool negativ = (i < 0);
            while(i != 0) {
                ergebnis.insert(ergebnis.begin(), abs(i % 10) + '0');
                i /= 10;
            }
            if(negativ) {
                ergebnis.insert(ergebnis.begin(), '-');
            }
        }
        return ergebnis;
    }
}

class HttpResponse {
public:
    HttpResponse(int st)
        : status(st) {
        std::string zeile("HTTP/1.1 ");
        zeile += i2string(status);
        zeile += std::string(" ") + getStatusText() + "\r\n";
        header = zeile;
    }

    void addToHeader(const std::string& key, const std::string& value) {
        header += key + ": " + value + "\r\n";
    }

    void sendHeader(boost::asio::ip::tcp::socket& socket) {
        header += "\r\n";
        socket.write_some(boost::asio::buffer(header.c_str(), header.length()));
        std::cout << "GESENDETER HEADER:\n" << header;
    }
}
```

```

    std::string getHTMLMessage(const std::string& text = "") {
        std::string statusMsg(i2string(status) + " " + getStatusText());
        std::string msg("<html><head><title>");
        msg += statusMsg + "</title></head>";
        msg += "<body><h1>" + statusMsg + "</h1>" + text + "</body></html>";
        return msg;
    }
private:
    std::string getStatusText() {
        std::string txt;
        switch (status) { // reduzierte Auswahl
            case 200: txt = "OK"; break;
            case 400: txt = "Bad Request"; break;
            case 404: txt = "Not Found"; break;
            case 500: txt = "Internal Server Error"; break;
            case 501: txt = "Not Implemented"; break;
            default: txt = "undefined";
        }
        return txt;
    }
    int status;
    std::string header;
};
#endif

```

Die Hilfsfunktion `i2string()` wandelt den Status in einen String um, weil ein String in einem bestimmten Format als erste Zeile des Headers zurückgegeben werden muss. Dem Protokoll HTTP/1.1 folgt der Status und darauf der Text, der zu diesem Status gehört. Diesen Text liefert die Funktion `getStatusText()`, wobei hier nur sehr wenige Möglichkeiten aus [Fiel] realisiert werden. Die Funktion `getHTMLMessage()` wird nur benutzt, um dem Clienten einen Fehler als HTML-Seite anzeigen zu können. So wird zum Beispiel die Meldung »Datei nicht gefunden!« gegebenenfalls zusätzlich zum Fehlerstatus im Browser angezeigt, wie die Klasse `DateiHandler` zeigt:

Listing 15.17: Klasse `DateiHandler`

```

// cppbuch/k15/http/server/DateiHandler.h
#ifndef DATEIHANDLER_H
#define DATEIHANDLER_H
#include<fstream>
#include<string>
#include<boost/asio.hpp>
#include<boost/filesystem/operations.hpp>
#include"HttpResponse.h"

class DateiHandler {
public:
    DateiHandler(boost::asio::ip::tcp::socket& s, const std::string& d)
        : socket(s), dateiname(d) {
    }

    void process() {
        std::ifstream quelle(dateiname.c_str(), std::ios::in | std::ios::binary);
    }
};

```



```

        if(quelle) {
            sendeDatei(quelle);
        }
        else { // Datei nicht vorhanden. Header zusammenbauen und senden:
            HttpResponse httpResponse(404); // not found
            httpResponse.addToHeader("Content-Type", "text/html");
            std::string msg = httpResponse.getHTMLMessage("Datei nicht gefunden!");
            httpResponse.addToHeader("Content-Length", i2string(msg.length()));
            httpResponse.sendHeader(socket);
            // msg senden (html-Seite)
            socket.write_some(boost::asio::buffer(msg.c_str(), msg.length()));
        }
    }
private:
    boost::asio::ip::tcp::socket& socket;
    std::string dateiname;

    std::string getContentType() {
        std::string typ("application/octet-stream"); // Vorgabe
        // Datei-Extension ermitteln und in Großschreibung umwandeln
        size_t punkt = dateiname.rfind('.');
        if(punkt < std::string::npos) {
            std::string extension = dateiname.substr(punkt+1);
            for(size_t i = 0; i < extension.length(); ++i) {
                extension[i] = toupper(extension[i]);
            }
            // unvollständige Auswahl aus http://de.selfhtml.org/diverses/mimetypen.htm:
            if(extension == "HTML") typ = "text/html";
            else if(extension == "TXT") typ = "text/plain; charset=iso-8859-1";
            else if(extension == "JPG") typ = "image/jpeg";
            else if(extension == "PNG") typ = "image/png";
            else if(extension == "PDF") typ = "application/pdf";
        }
        return typ;
    }

    void sendeDatei(std::ifstream& quelle) {
        boost::filesystem::path p(dateiname);
        size_t bufsize = boost::filesystem::file_size(p);
        // Header zusammenbauen und senden
        HttpResponse httpResponse(200);
        httpResponse.addToHeader("Content-Type", getContentType());
        httpResponse.addToHeader("Content-Length", i2string(bufsize));
        httpResponse.sendHeader(socket);
        // Datei senden
        char* const buf = new char[bufsize];
        size_t pos = 0;
        while(quelle.get(buf[pos++])); // Datei lesen und in buf abspeichern
        socket.write_some(boost::asio::buffer(buf, bufsize));
        delete [] buf;
    }
};
#endif

```

Falls es die gewünschte Datei gibt, wird sie gesendet. Dazu muss zunächst der Header mit dem Status 200 für OK erzeugt werden. Der Header enthält auch den Content-Type (MIME-Typ), der in Abhängigkeit vom Dateityp von der Funktion `getContentType()` ermittelt wird, sowie die Größe der Datei. Letztere ist mit Standardmitteln der Sprache C++ nicht herauszufinden; deshalb kommt die Boost-Bibliothek zum Einsatz.



Mehr zu Dateioperationen lesen Sie ab Seite 727.

Die Klasse `AnfrageHandler` analysiert die Anfrage und schickt je nach Ergebnis (Begrüßung, Zeitinformation oder Fehler) eine HTML-Seite zurück. In diesem einfachen Programm wird angenommen, dass es nur ein Schlüssel/Wert-Paar gibt. Tatsächlich sind im GET-Protokoll beliebig viele erlaubt.

Listing 15.18: Klasse `AnfrageHandler`

```
// cppbuch/k15/http/server/AnfrageHandler.h
#ifndef ANFRAGEHANDLER_H
#define ANFRAGEHANDLER_H
#include <ctime>
#include <cstring>
#include <string>
#include <boost/asio.hpp>
#include "HttpResponse.h"

class AnfrageHandler {
public:
    AnfrageHandler(boost::asio::ip::tcp::socket& s, const std::string& a)
        : socket(s), anfrage(a) {
        std::cout << "ANFRAGE=" << a << std::endl;
    }

    void process() {
        size_t pos = anfrage.find('=');
        std::string key = anfrage.substr(0, pos);
        std::string val = anfrage.substr(pos+1);
        std::cout << "KEY=" << key << " VAL=" << val << std::endl;
        if(key == "zeitabfrage" && val == "Datum") {
            time_t jetzt = time(NULL);
            std::string zeitstring(ctime(&jetzt));
            // Header zusammenbauen und senden
            HttpResponse httpResponse(200);
            httpResponse.addToHeader("Content-Type", "text/html");
            std::string msg =
                "<html><head><title>C++ MiniWebServer</title></head><body>"
                "<img src=\"bild.png\" width=\"80%\"<br><hr><p>";
            msg += zeitstring + "</body></html>";
            httpResponse.addToHeader("Content-Length", i2string(msg.length()));
            httpResponse.sendHeader(socket);
            // Ergebnis als html-Seite senden
            socket.write_some(boost::asio::buffer(msg.c_str(), msg.length()));
        }
        else if(key == "begruessung") {
```

```

    // Header zusammenbauen und senden
    HttpResponse httpResponse(200);
    httpResponse.addToHeader("Content-Type", "text/html");
    std::string msg =
        "<html><head><title>C++ MiniWebServer</title></head><body>"
        "<img src=\"bild.png\" width=\"80%\"><br><hr><p><h1>Guten Tag, ";
    msg += val + "!<h1></body></html>";
    httpResponse.addToHeader("Content-Length", i2string(msg.length()));
    httpResponse.sendHeader(socket);
    // Ergebnis als html-Seite senden
    socket.write_some(boost::asio::buffer(msg.c_str(), msg.length()));
}
else {
    // Header zusammenbauen und senden
    HttpResponse httpResponse(400);
    httpResponse.addToHeader("Content-Type", "text/html");
    std::string msg = httpResponse.getHTMLMessage("unbekannte Abfrage!");
    httpResponse.addToHeader("Content-Length", i2string(msg.length()));
    httpResponse.sendHeader(socket);
    // Fehlermeldung als html-Seite senden
    socket.write_some(boost::asio::buffer(msg.c_str(), msg.length()));
}
}
private:
    boost::asio::ip::tcp::socket& socket;
    std::string anfrage;
};
#endif

```

Dieses Kapitel zeigt die wichtigsten Aspekte der Netzwerkprogrammierung in kurzer und beispielhafter Form. Etliche Themen werden jedoch ausgeklammert, von denen ich einige herausgreife:

- Es gibt die Möglichkeit des asynchronen Lesens, Schreibens und Akzeptierens von Verbindungen.
- Es kann mit `iostream`-entsprechenden Strömen gearbeitet werden.
- Das Bearbeiten einer Verbindung wird einem eigenen Thread gegeben, damit der Server sofort bereit ist, eine neue Verbindung zu akzeptieren, auch wenn die Bearbeitung der vorhergehenden noch andauert.
- In der Server-Praxis werden Thread-Pools verwendet, weil Anlegen und Zerstören eines Threads teure Operationen im Sinn der notwendigen Laufzeit sind.
- Das Server-Beispiel bietet nur fest einprogrammierte Funktionen. In der Praxis ist Flexibilität möglich.

Wenn keine oder kaum HTML-Kenntnisse vorhanden sein sollten: <http://de.selfhtml.org/> bietet eine Fülle von Informationen und Beispielen. Wenn Sie mehr zu den anderen Themen wissen möchten, empfehle ich, [asio] und [SSRB] zu Rate zu ziehen.

16

Datenbankanbindung

Dieses Kapitel behandelt die folgenden Themen:

- Datenbankanbindung am Beispiel SQLite
- C++-Schnittstelle
- Anwendungsbeispiel

Viele Applikationen bauen auf der Nutzung einer Datenbank auf. Ein Datenbanksystem besteht aus der eigentlichen Datenbank und einer Software zur Verwaltung (DBMS = Datenbankmanagementsystem). Die meisten Datenbanksysteme sind sogenannte relationale Datenbanken (RDBMS), die intern als Sammlungen von Tabellen (Relationen) organisiert sind. Mittlerweile gibt es auch objektorientierte Datenbanken oder objektrelationale Mischformen, die nicht sehr verbreitet sind. Operationen auf diesen Tabellen werden mit Hilfe der Structured Query Language (SQL) bewerkstelligt. Es gibt zwei typische Szenarien:

- Die Datenbank ist als Server realisiert, gegebenenfalls auf einem entfernten Rechner. Clienten greifen über ein Netzwerk auf die Datenbank zu. Weil die Clienten voneinander unabhängig sind, kann es viele gleichzeitige Zugriffe auf die Datenbank geben. Wegen der Trennung von den Applikationen ist dieses Modell geeignet, wenn sehr viele Daten verwaltet werden oder/und ein hoher Durchsatz erforderlich ist, wie er bei einer stark frequentierten Website auftritt. Typische Vertreter so einer Datenbank

sind Oracle oder DB2 oder im Open Source Bereich die verbreiteten Systeme MySQL und PostgreSQL.

- Die Datenbank ist an eine Applikation gebunden. Sie kann eine Menge von Dateien ersetzen, in denen vorher lokale Daten abgelegt wurden. So eine Datenbank eignet sich für Desktop-Anwendungen verschiedener Art und wegen ihrer geringen Größe für mobile Geräte. SQLite [[SQLite](#)] ist ein freies RDBMS, das dafür sehr gut einsetzbar ist und sehr wenig administrativen Aufwand erfordert.

Die C++-Anbindung an eine Datenbank ist in beiden Fällen ähnlich, weil entsprechende APIs zur Verfügung gestellt werden. Für dieses Kapitel habe ich mich entschieden, die Datenbankankbindung am Beispiel von SQLite zu zeigen, weil der Einsatz einer solchen Datenbank in vielen Fällen ausreichend ist. Alle zu einer Datenbank gehörenden Daten werden von SQLite in einer einzigen Datei abgelegt. SQLite ist portabel und wird millionenfach in bekannten Produkten eingesetzt (Mozilla Firefox, Google Gears, Adobe Photoshop Lightroom, Handys mit Symbian-Betriebssystem und andere) und ist integraler Bestandteil der Programmiersprachen PHP und Python.



Hinweis

Dieses Kapitel konzentriert sich auf die Ansprache von SQLite mit C++, um Ihnen einen leichten *Einstieg* in das Thema Datenbankankbindung zu ermöglichen. In den Beispielen wird auf SQL und auf die umfangreichen Funktionen von SQLite nur in aller Kürze eingegangen. Bei tiefergehendem Bedarf bitte ich Sie, ein Buch zu Datenbanken wie [[Date](#)] und die Webseite [[SQLite](#)] zu konsultieren.

SQLite ist leicht zu installieren, unter Linux am besten mit dem Werkzeug zur Systemverwaltung. Es muss nur darauf geachtet werden, die neuere Version *sqlite3* und auch das Package *sqlite3-devel* zu installieren. Unter Windows brauchen Sie nichts weiter zu tun, wenn Sie die Standardinstallation des Compilers und anderer Programme von der DVD zum Buch entsprechend der Anleitung vornehmen; SQLite ist dort bereits integriert.

16.1 C++-Interface

SQLite ist vollständig in C geschrieben und stellt an die 175 Funktionen zur Verfügung. Ergebnisse werden oft in einem Speicherbereich bereitgestellt, der intern mit der C-Speicherbeschaffungsfunktion `malloc` angelegt wird, für dessen Freigabe jedoch der Aufrufer der Funktion verantwortlich ist. Das kann auch mal vergessen werden, weswegen es guter Stil in C++ ist, die Freigabe in den Destruktor zu verlegen oder anderweitig automatisch sicherzustellen. Die Klasse `Datenbank` unten kapselt den Zugriff auf die Datenbank und garantiert die Freigabe, indem die entsprechenden `sqlite3_free...`-Funktionen aufgerufen werden.

Glücklicherweise kommt man in den meisten Fällen mit nur wenigen der vielen Funktionen aus, weil es darunter welche gibt, die sehr viel Funktionalität bieten. Die wichtigsten Funktionen sind:

- `int sqlite3_open(const char* dbname, sqlite** ptr)` öffnet die in der Datei `dbname` vorliegende Datenbank, wenn die Datei existiert. Wenn nicht, wird sie neu erzeugt. Falls der `dbname` `:memory` lautet, wird eine temporäre Datenbank im Speicher angelegt. Falls der `dbname` ein leerer C-String ist, wird eine Datenbank als temporäre Datei geschaffen. Diese temporäre Datei wird sofort nach Schließen der Datenbank gelöscht. `ptr` ist die Adresse eines Zeigers `db`, der nach dem Aufruf auf ein `sqlite3`-Objekt verweist. Mit Hilfe des Handles `db` sind Zugriffe auf die Daten möglich, wie im Folgenden zu sehen. Die Funktion gibt im Erfolgsfall `SQLITE_OK` zurück, andernfalls einen Fehlercode.
- `int sqlite3_close(sqlite* db)`: Verbindung zur Datenbank schließen. Die Funktion gibt im Erfolgsfall `SQLITE_OK` zurück, andernfalls einen Fehlercode.
- `int sqlite3_get_table(`
`sqlite3 *db,           // geöffnete Datenbank`
`const char *sqlAnw,   // SQL-Anweisung`
`char*** pTabelle,   // Hierhin wird das Ergebnis der Anweisung geschrieben.`
`int *pZ,             // Die Anzahl der Zeilen wird an pZ geschrieben.`
`int *pS,             // Die Anzahl der Spalten wird an pS geschrieben.`
`char **pfehlerrmsg   // Hierhin wird ggf. eine Fehlermeldung geschrieben.`
`)`

Falls die auszuwertende SQL-Anweisung kein Ergebnis liefert, etwa bei einer `insert`-Anweisung, bleibt die Tabelle leer. Die Funktion gibt im Erfolgsfall `SQLITE_OK` zurück, andernfalls einen Fehlercode.

- `void sqlite3_free(void* fehlermsg)`: gibt den für die Fehlermeldung angelegten Speicherplatz frei.
- `int sqlite3_free_table(char** tabelle)`: gibt den für die Tabelle angelegten Speicherplatz frei. `sqlite3_free()` genügt nicht!
- `const char* sqlite3_errmsg(sqlite* db)`: gibt die zum zuletzt aufgetretenen Fehler passende Fehlermeldung zurück.
- `int sqlite3_exec(`
`sqlite3 *db,           // geöffnete Datenbank`
`const char *sqlAnw,   // SQL-Anweisung`
`int (*f)(void*, int, char**, char**), // Callback-Funktion`
`void* arg,            // Erstes Argument der Callback-Funktion`
`char **pfehlerrmsg   // Hierhin wird ggf. eine Fehlermeldung geschrieben.`
`)`

Ähnlich wie `sqlite3_get_table()` führt diese Funktion eine SQL-Anweisung aus. Der Unterschied besteht darin, dass keine Tabelle zurückgegeben, sondern die Funktion `f` für jede Zeile aufgerufen wird. Der Funktion `f` werden übergeben: `arg` (Parameter 1), die Anzahl der Spalten des aktuellen Ergebnisses (Parameter 2), ein Array von Zeigern auf C-Strings für jede Spalte (Parameter 3), ein Array von Zeigern auf C-Strings für jede Spaltenüberschrift (Parameter 4). Die selbst zu schreibende Funktion kann diese Parameter nach Wunsch auswerten.

Bis auf die letzte Funktion, für die es in [SQLite] ein Beispiel gibt, werden alle genannten Funktionen unten eingesetzt.


```

        &zeilen, &spalten, &fehlermeldung);
Array2d<std::string> ergebnis(1, 1, ""); // Platzhalter
if(erg != SQLITE_OK) { // Fehler?
    std::string msg(fehlermeldung);
    sqlite3_free(fehlermeldung); // Freigaben nicht vergessen!
    sqlite3_free_table(cstringarray);
    throw SQLError(msg);
}
else {
    if(zeilen > 0) { // Zeilen einsammeln, eine mehr für die Überschrift
        ergebnis = Array2d<std::string>(++zeilen, spalten);
        for(int z = 0; z < zeilen; ++z) {
            for(int s = 0; s < spalten; ++s) {
                const char* str = cstringarray[z*spalten + s];
                ergebnis[z][s] = str ? str : ""; // NULL berücksichtigen
            }
        }
        sqlite3_free_table(cstringarray); // Freigabe nicht vergessen!
    }
    return ergebnis;
}
private:
    sqlite3 *db;
    Datenbank(const Datenbank&); // kopieren verbieten
    Datenbank& operator=(const Datenbank&); // zuweisen verbieten
};
#endif

```

Das folgende kleine Programm bedient sich der obigen Schnittstelle. Mit ihm können interaktiv SQL-Anweisungen eingegeben und deren Ergebnisse angezeigt werden. Die globale Funktion `printArray()` gehört zur Klasse `Array2d`.

Listing 16.2: Programm zur interaktiven Eingabe von SQL-Anweisungen

```

// cppbuch/k16/abfrage.cpp
#include<iostream>
#include"Datenbank.h"

int main(int argc, char **argv){
    std::string zeile;
    std::cout << "Datenbankname: ";
    std::getline(std::cin, zeile);
    try {
        Datenbank db(zeile.c_str());
        do {
            std::cout << "SQL-Anweisung: ";
            std::getline(std::cin, zeile);
            if(zeile.length() > 0) {
                printArray(db.execute(zeile.c_str()));
            }
        } while(zeile.length() > 0);
    } catch(const SQLError& e) {
        std::cout << e.what() << std::endl;
    }
}

```



```
}
}
```

Zuerst wird der Name der Datenbankdatei abgefragt, dann die SQL-Anweisungen. Eine leere Anweisung bewirkt den Abbruch des Programms. Diese Struktur erlaubt die Abarbeitung von Skripten, zum Beispiel `abfrage.exe < skript.txt`, wobei das Skript etwa wie folgt aussehen könnte:

```
shopdb.db
select * from Kunde;
select * from Rechnung;
```

16.2 Anwendungsbeispiel

Um eine konkrete Anwendung zu zeigen, wird zunächst eine kleine Datenbank definiert. Sie enthält Kunden, Adressen, Artikel, Rechnungen und Rechnungspositionen. Die Aufgabe besteht darin, per Programm Rechnungen zu schreiben. Das Datenbankschema ist durch die folgenden erzeugenden SQL-Anweisungen gegeben (Erläuterungen folgen):

Listing 16.3: Datenbankschema

```
-- cppbuch/k16/shopdb.txt
create table Adresse (
  Id integer primary key autoincrement,
  Ort text,
  Plz text,
  StrNr text);

create table Kunde (
  Id integer primary key autoincrement,
  Name text,
  AdressId integer not null,
  foreign key(AdressId) references Adresse(Id));

create table Artikel (
  Nr integer primary key,
  Bezeichnung text,
  Preis numeric(8,2) not null);

create table Rechnung (
  Id integer primary key autoincrement,
  Datum date,
  -- KundenNr darf NULL sein (Barverkauf)
  KundenNr integer,
  foreign key(KundenNr) references Kunde(Id));

create table Position (
  RechnungsNr integer not null,
```

```

ArtikelNr integer not null,
Menge integer,
check (Menge > 0),
foreign key(Rechnungsnr) references Rechnung(Id),
foreign key(ArtikelNr) references Artikel(Nr));

create trigger datumtrigger after insert on Rechnung
begin
  update Rechnung set Datum = DATE('NOW') where rowid = new.rowid;
end;

create view positionen as
  select Rechnungsnr, ArtikelNr, Menge, Bezeichnung, Preis, Preis*Menge
  from (Position p join Rechnung r on Rechnungsnr=r.Id)
  left outer join Artikel art on ArtikelNr=art.Nr;

-- alle Rechnungsinfos
create view alles as
  select Rechnungsnr, Datum, Name, Plz, Ort, StrNr, Menge, ArtikelNr,
    Bezeichnung, Preis, Preis*Menge
  from (((Position p join Rechnung r on Rechnungsnr=r.Id)
    left outer join Kunde k on KundenNr=k.Id)
    left outer join Artikel art on ArtikelNr=art.Nr)
    left outer join Adresse a on AdressId=a.Id;

```

Die Datenbank *shopdb.db* wird neu erzeugt, indem eine möglicherweise vorher existierende Datei gleichen Namens gelöscht und dann

```
sqlite3 shopdb.db < shopdb.txt
```

aufgerufen wird. Einem Kunden ist eine Adresse nicht direkt, sondern als Verweis (Fremdschlüssel) zugeordnet, weil erstens ein Kunde umziehen kann und zweitens mehrere Kunden dieselbe Adresse haben können (Mehrfamilienhaus). Damit werden die Informationen entkoppelt. *primary key* besagt, dass der Eintrag zur eindeutigen Identifizierung genommen werden kann. *not null* bedeutet, dass der Eintrag nicht leer sein darf.

Eine Rechnung enthält das Datum und die Kundennummer. Das Datum wird automatisch beim Anlegen einer neuen Rechnung eingetragen, bewirkt durch den Trigger *datumtrigger*. Eine Position verweist auf die zugehörige Rechnung und den Artikel. Die Abbildung 16.1 zeigt die Struktur in grafischer Form.

Am Ende der Datei sind zwei Sichten (englisch *view*) definiert. Die Sicht *positionen* selektiert die Positionen. Die Abfrage `select * from positionen where Rechnungsnr = 1;` gibt alle zur Rechnung 1 gehörenden Positionen als Tabelle zurück, wobei eine Zeile aus den Informationen *Rechnungsnr*, *ArtikelNr*, *Menge*, *Bezeichnung*, *Preis* und *Preis*Menge* besteht. Wie Sie sehen, können auch arithmetische Operationen vorgenommen werden. Die Sicht *alles* liefert sämtliche in der Datenbank enthaltenen Informationen, die mit *where* natürlich wie oben gefiltert werden können. Um die Datenbank zu füllen, wird

```
sqlite3 shopdb.db < shopdbfuellen.txt
```

aufgerufen. Die Datei *shopdbfuellen.txt* enthält SQL-Anweisungen, um Daten in die verschiedenen Tabellen einzutragen. Dabei werden teilweise die Fremdschlüssel der Kürze halber direkt angegeben, teilweise mit *select* ermittelt.

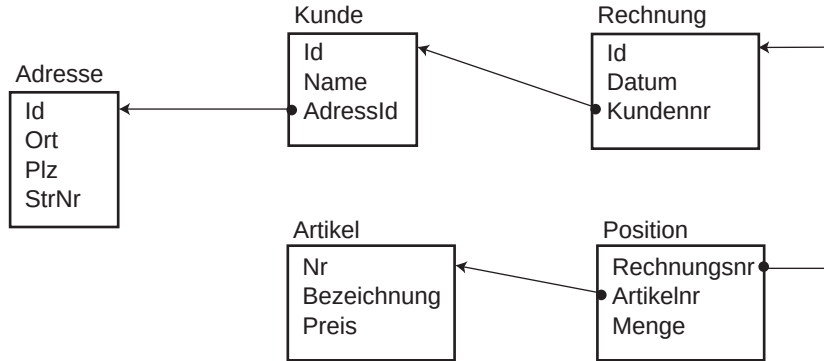


Abbildung 16.1: Datenbankschema

Listing 16.4: Datenbankinhalt für das Beispiel

```

-- cppbuch/k16/shopdbfuellen.txt
insert into Adresse (Ort, Plz, StrNr) values(
    'Entenhausen', '12345', 'Erpelweg 7');
insert into Adresse (Ort, Plz, StrNr) values(
    'Muenchen', '81679', 'Kolbergerstr. 22');
insert into Adresse (Ort, Plz, StrNr) values(
    'Barcelona', '08037', 'Roger De Lluria 21');

-- Zahl = Adress-Index
insert into Kunde (Name, AdressId) values('Donald Duck', 1);
insert into Kunde (Name, AdressId) values('Addy Woollen', 3);
insert into Kunde (Name, AdressId) values('Carl Hanser', 2);
-- alternativ: mit select ermitteln:
insert into Kunde (Name, AdressId) values('Daisy Duck',
    (select Id from Adresse where Ort='Entenhausen'));

insert into Artikel values(4727, 'Schokolinsen', 1.10);
insert into Artikel values(123, 'Champagner', 39.99);
insert into Artikel values(9282, 'Kaffeemaschine', 21.95);
insert into Artikel values(1912, 'Rezeptbuch', 9.00);

-- Daisy kaufte 20 Schokolinsen, 1 Rezeptbuch, 1 Champagner
insert into Rechnung (KundenNr) values((select Id from Kunde where
    Name='Daisy Duck'));

-- RechnungsNr, ArtikelNr, Stueckzahl
insert into Position values(1, 4727, 20);
insert into Position values(1, 1912, 1);
-- oder mit select:
insert into Position values((select Id from Rechnung where Rechnung.KundenNr=
    (select Id from Kunde where Name == 'Daisy Duck')), 123, 1);

-- Barverkauf, keine KundenNr
insert into Rechnung (KundenNr) values(NULL);
insert into Position values(2, 9282, 2);

```

```
-- Rechnung 3 fuer Kunde Nr. 3 (Carl Hanser)
insert into Rechnung (KundenNr) values(3);
insert into Position values(3, 123, 10);
insert into Position values(3, 1912, 1);

-- Rechnung 4 fuer Kunde Nr. 2 (Addy Woollen)
insert into Rechnung (KundenNr) values(2);
insert into Position values(4, 123, 50);
```

Rechnungen schreiben

Die obige Datenbank soll so ausgewertet werden, dass alle Rechnungen, denen ein Kunde zugeordnet ist (kein Barverkauf), zum Versand ausgedruckt werden sollen. Der Einfachheit halber wird auf das Währungssymbol und die Mehrwertsteuerberechnung verzichtet; beide sind bei Bedarf leicht nachzurüsten. Die Aufgabe wird auf zwei Arten gelöst:

- Die Anfrage `select * from alles;` wird ausgewertet.
- Im zweiten Fall wird gezeigt, wie die benötigten Informationen gezielt mit einzelnen `select`-Anweisungen ermittelt werden.

Listing 16.5: Rechnungen schreiben mit `select * from alles;`

```
// cppbuch/k16/rechnung.cpp
#include<iostream>
#include"Datenbank.h"

namespace {
    // SQL-Zahlen auf 2 dargestellte Nachkommazahlen bringen
    std::string formatPreis(const std::string& s) {
        size_t punkt = s.find('.');
        if(punkt == std::string::npos) {
            return s + ".00";
        } else if(s.length() - punkt == 2) {
            return s + "0";
        }
        return s;
    }
}

int main(int argc, char **argv){
    enum Struktur {NR, DATUM, NAME, PLZ, ORT, STRNR, MENGE, ARTNR, BEZ,
                  EPREIS, GPREIS};
    try {
        Datenbank db("shopdb.db");
        Array2d<std::string> rechnungen = db.execute("select * from alles");
        std::string rechnNr("");
        int z = 1;
        bool rechnungskopfGedruckt = false;
        while(z < rechnungen.getZeilen()){
            if(rechnNr != rechnungen[z][NR]) { // neue Rechnung beginnt
                rechnNr = rechnungen[z][NR];
                rechnungskopfGedruckt = false;
            }
        }
    }
```

```

if(rechnungen[z][NAME] != "") { // Kunde existiert, kein Barverkauf
    int position = 0;
    if(!rechnungskopfGedruckt) {
        std::cout
            << "Herrn/Frau/Firma " << rechnungen[z][NAME] << std::endl
            << rechnungen[z][STRNR] << std::endl
            << rechnungen[z][PLZ] << " " << rechnungen[z][ORT]
            << "\n\nRechnung Nr. " << rechnNr
            << " vom " << rechnungen[z][DATUM] << std::endl
            << "Pos. Menge Artikelnr. Bezeichnung "
            << "Einzelpreis Gesamtpreis" << std::endl;
        rechnungskopfGedruckt = true;
        position = 1;
    }
    // Positionen ausgeben
    std::cout.width(4);
    std::cout << position++;
    std::cout.width(6);
    std::cout << rechnungen[z][MENGE];
    std::cout.width(12);
    std::cout << rechnungen[z][ARTNR];
    std::cout.width(20);
    std::cout << rechnungen[z][BEZ];
    std::cout.width(12);
    std::cout << formatPreis(rechnungen[z][EPREIS]);
    std::cout.width(12);
    std::cout << formatPreis(rechnungen[z][GPREIS]) << std::endl;
}
++z;
if(rechnungskopfGedruckt &&
    (z == rechnungen.getZeilen()
    || (z < rechnungen.getZeilen()
        && rechnNr != rechnungen[z][NR]))) {
    Array2d<std::string> summe
        = db.execute(
            "select sum(Preis*Menge) from alles where Rechnungsnr="
            + rechnNr + ";");
    std::cout.width(66);
    std::cout << "-----" << std::endl;
    std::cout.width(66);
    std::cout << formatPreis(summe[1][0]) << std::endl
        << '\f' << std::endl; // Seitenvorschub
}
}
} catch(const SQLError& e) {
    std::cout << e.what() << std::endl;
}
}

```

Die zurückgegebene Tabelle enthält als erste Zeile die Spaltenüberschriften und in den Folgezeilen alle Positionen mit Rechnungsdatum, Kundenname usw. Wegen der zu langen Zeilen verkürzter Auszug:

```
Rechnungsnr Datum Name Plz Ort StrNr Menge Artikelnr Bezeichnung Preis ...
1 2011-01-16 Daisy Duck 12345 Entenhausen Erpelweg 7 20 4727 Schokolinsen 1.1 ..
1 2011-01-16 Daisy Duck 12345 Entenhausen Erpelweg 7 1 1912 Rezeptbuch 9 ...
USW.
```

Am besten ist es, Sie probieren selbst die Anfrage mit dem Programm *abfrage.exe* oder *sqlite3* aus, um die Tabelle zu sehen. Bei der Auswertung ist zu beachten, dass sich die Rechnungsnummer und Kundennamen bei allen zur selben Rechnung gehörenden Positionen wiederholen. Ein Wechsel in der Rechnungsnummer bedeutet damit Abschluss der laufenden Rechnung und Beginn einer neuen Rechnung. So sieht eine der erzeugten Rechnungen aus:

```
Herrn/Frau/Firma Daisy Duck
Erpelweg 7
12345 Entenhausen
```

```
Rechnung Nr. 1 vom 2011-01-16
Pos. Menge Artikelnr.      Bezeichnung Einzelpreis Gesamtpreis
1   20      4727      Schokolinsen      1.10      22.00
2    1      1912      Rezeptbuch       9.00       9.00
3    1      123       Champagner     39.99     39.99
                                     -----
                                     70.99
```

Die Preise werden mit einer SQL-Anweisung aufsummiert, die auf die Spalte Gesamtpreis wirkt (d.h. Preis*Menge):

```
select sum(Preis*Menge) from alles where Rechnungsnr=X;
```

X ist hier nur ein Platzhalter. In der zweiten Variante werden zur Demonstration gezielt die Rechnungsnummern ermittelt, und zu jeder Rechnungsnummer werden der zugehörige Kunde und alle zugehörigen Positionen bestimmt. Für jeden Kunden wird die Adresse erfragt. Damit entfallen redundante Einträge wie die Wiederholungen von Rechnungsnummer, Name und Adresse im ersten Beispiel. Das und die Demonstration der verschiedenen Abfragen und Auswertungen bilden den Vorteil dieses Programms. Der Vorteil wird jedoch mit einer erhöhte Komplexität des Codes und durch einen Laufzeitnachteil, bedingt durch die vergleichsweise vielen SQL-Anweisungen, erkauft.

Listing 16.6: Rechnungen schreiben; zweite Variante

```
// Auszug aus cppbuch/k16/rechnung1.cpp
int main(int argc, char **argv){
    try {
        Datenbank db("shopdb.db");
        Array2d<std::string> rechnungen = db.execute("select * from Rechnung");
        // z=1: Ueberschrift ausblenden
        for(int z = 1; z < rechnungen.getZeilen(); ++z) {
            std::string abfrage("select Name,Id from Kunde where Kunde.Id ="
                                " (select Kundennr from Rechnung where "
                                "Rechnung.Kundennr=Kunde.Id and Rechnung.Id=)");
            abfrage += rechnungen[z][0] + ");";
            // Kunde zur Rechnung ermittel
            Array2d<std::string> kunde = db.execute(abfrage);
            if(kunde.getZeilen() > 1) { // Kunde existiert, kein Barverkauf
```

```

std::cout << "Herrn/Frau/Firma " << kunde[1][0] // Name
    << std::endl;
// Adresse des Kunden ermitteln
abfrage = "select Plz, Ort, StrNr from Adresse where "
    "(select Id from Kunde where Kunde.AdressId=Adresse.Id "
    " and Id=" + kunde[1][1] + ")";
Array2d<std::string> adresse = db.execute(abfrage);
std::cout << adresse[1][2] << std::endl // Strasse Hausnr
    << adresse[1][0] << " " << adresse[1][1] // PLZ Ort
    << "\n\nRechnung Nr. " << rechnungen[z][0]
    << " vom " << rechnungen[z][1] << std::endl;
// Rechnungspositionen ermitteln (View nutzen)
abfrage = "select * from positionen where Rechnungsnr="
    + rechnungen[z][0] + ";";
Array2d<std::string> positionen = db.execute(abfrage);
// Positionen ausgeben
std::cout << "Pos. Menge Artikelnr.          Bezeichnung "
    "Einzelpreis Gesamtpreis" << std::endl;
enum Positionen {NR, ARTNR, MENGE, BEZ, EPREIS, GPREIS};
// p=1: Ueberschrift ausblenden
for(int p = 1; p < positionen.getZeilen(); ++p) {
    std::cout.width(4);
    std::cout << p;
    std::cout.width(6);
    std::cout << positionen[p][MENGE];
    std::cout.width(12);
    std::cout << positionen[p][ARTNR];
    std::cout.width(20);
    std::cout << positionen[p][BEZ];
    std::cout.width(12);
    std::cout << formatPreis(positionen[p][EPREIS]);
    std::cout.width(12);
    std::cout << formatPreis(positionen[p][GPREIS]) << std::endl;
}
// Summe bilden
abfrage = "select sum(Preis*Menge) from positionen where "
    "Rechnungsnr=" + rechnungen[z][0] + ";";
Array2d<std::string> summe = db.execute(abfrage);
std::cout.width(66);
std::cout << "-----" << std::endl;
std::cout.width(66);
std::cout << formatPreis(summe[1][0]) << std::endl
    << '\f' << std::endl; // Seitenvorschub
}
}
} catch(const SQLError& e) {
    std::cout << e.what() << std::endl;
}
}
}

```



Teil III: Praktische Methoden und Werkzeuge der Softwareentwicklung

17

Abläufe automatisieren mit make

Dieses Kapitel behandelt die folgenden Themen:

- Einführung in make
- Variablen und Muster
- Universelles Makefile für einfache Projekte

Bei der Programmentwicklung gibt es immer wiederkehrende Aufgaben. Darunter sind die Compilation und das Testen eines Programms besonders häufig. Andere häufige Aufgaben sind das Herstellen einer aktuellen Dokumentation, das Sichern von Dateien, Bauen einer Zusammenstellung¹ von Programm und Dokumentation zur Verteilung oder das Laden von Programmen auf einen Web-Server. Auch das Löschen nicht mehr benötigter Dateien gehört dazu. Diese Aufgaben in Einzelschritten per Hand durchzuführen, ist mühsam und fehleranfällig, weswegen schon früh dafür Werkzeuge ersonnen wurden. Der eigentliche Ablauf der Übersetzung und Erzeugung des lauffähigen Programms, Build-Prozess genannt, findet bei einer Entwicklungsumgebung (IDE) im Hintergrund statt. Programme wie *make*, die über die Kommandozeile aufgerufen werden können, erlauben das Bauen (englisch *build*) eines Programms auch ohne IDE.

make wird mit Regeln gesteuert, die sich in einer Textdatei, allgemein Makefile genannt, befinden. Im C/C++-Bereich gab und gibt es das Programm *make*, das auch für Win-

¹ Distribution genannt.

dows und Mac OS erhältlich ist. In der Java-Programmierung setzte sich das ebenso mächtige Tool *ant* durch, das von einer XML-Datei gesteuert wird wie auch MSBuild von Microsoft. Ein Makefile hat einen speziellen syntaktischen Aufbau. Es gibt mehrere »Dialekte«, die sich allerdings ähneln. Von den vielen make-Programmen ist GNU-make das Bekannteste [make]. Es bietet nicht nur einen großen Funktionsumfang, sondern ist im Gegensatz zu manch anderen Programmen auch noch portabel. Aus diesem Grund beschränke ich mich im Folgenden auf GNU-make, dessen Grundlagen und sinnvoller Einsatz kurz beschrieben werden. Aus Platzgründen werden nur die Funktionsweise, die wichtigsten Eigenschaften und konkrete Beispiele für den täglichen Gebrauch beleuchtet. Wer mehr wissen möchte, sei auf die angegebene Quelle [make] verwiesen und auf das Kapitel 23, das fortgeschrittene Techniken auf der Basis von make zum Inhalt hat. Unter Linux gibt es eine kurze Hilfe mit den Befehlen `man make` oder `info make`.

Makefiles sind Dateien, die Abläufe von Dienstprogrammen des Betriebssystems steuern. Dabei wird auf Datum und Uhrzeit der beteiligten Dateien geachtet, sodass überflüssige Operationen vermieden werden, zum Beispiel die Compilation einer längst übersetzten Datei, die nicht geändert wurde. Bei Änderungen werden also nur die davon betroffenen Dateien neu übersetzt – ein großer Vorteil! Dieser Mechanismus wird auch von Entwicklungsumgebungen genutzt, von denen manche direkt *make* benutzen und auch ein Makefile erzeugen, das man sich ansehen kann.



17.1 Quellen

Linux

Auf Linux-Systemen sind in aller Regel nicht nur der GNU-C++-Compiler, sondern auch *make* und andere Hilfsprogramme enthalten.

Windows

Es gibt für Windows Umgebungen, die eine Linux-Umgebung emulieren. Für die Beispiele dieses Buchs wird MinGW zusammen mit MSYS (<http://www.mingw.org/>) verwendet. MinGW stellt das Programm *make* und einen C++-Compiler zur Verfügung. Die Abkürzung MinGW steht für »Minimalist GNU for Windows«. Diese Umgebungen enthalten die wichtigsten Open Source-Werkzeuge zur Entwicklung von Software. Die IDE Code::Blocks benötigt die Installation von MinGW. Für die Funktionsweise einiger unten vorgestellter Skripte wird die Installation von MinGW und MSYS entsprechend der Anleitung auf der DVD vorausgesetzt. Damit steht Ihnen *make* mit all seinen Möglichkeiten zur Verfügung.

17.2 Wirkungsweise

Ein Makefile hat eine spezielle Struktur, in der die Begriffe *Ziel* (englisch *target*), *Abhängigkeit* (englisch *dependency*) und *Aktion* eine besondere Rolle spielen. Der syntaktische Aufbau ist wie folgt:

```
# Kommentar
Ziel1: Abhaengigkeiten1
→Aktion1

Ziel2: Abhaengigkeiten2
→Aktion2
# ... und so weiter
```



Tipp

Das durch → symbolisierte Tabulatorzeichen kann bei manchen make-Programmen durch Leerzeichen ersetzt werden, nicht aber bei GNU-make!

Das Ziel kann (muss aber nicht) eine Datei sein, die erzeugt werden soll. Unter Abhängigkeiten werden die Dateien aufgelistet, von denen das Ziel abhängt, und die Aktion definiert, wie das Ziel erreicht wird. Alles, was in der Zeile nach dem →-Zeichen folgt, wird einer Instanz des Kommandointerpreters des Betriebssystems übergeben. Dabei ist jede →-Zeile unabhängig von den anderen!

Die nach dem Ziel notierte Abhängigkeit bezieht sich auf Datum und Uhrzeit der Dateien: Nur wenn eine der Dateien in der Liste *neuer* ist als das Ziel, wird die Aktion ausgeführt. Typische Aufrufe:

```
make -f makedatei.mak ziel   Ziel ziel der Datei makedatei.mak erzeugen.
make -f makedatei.mak      Das erste Ziel in makedatei.mak erzeugen.
make ziel                  Ziel ziel erzeugen, die Steuerdatei heißt
                           GNUmakefile, makefile oder Makefile.
make                      Das erste Ziel erzeugen, die Steuerdatei
                           heißt GNUmakefile, makefile oder Makefile.
```

Der Mechanismus sei am Beispiel der Klasse Rational von Seite 164 ff. gezeigt. Es gibt die Dateien *main.cpp*, *rational.h* und *rational.cpp*. Daraus soll die Datei *projekt.exe* als ausführbare Datei erzeugt werden. Wie Sie aus Kapitel 3 wissen (siehe Abbildung 3.9 auf Seite 124), müssen erst die Objektdateien *main.o* und *rational.o* erzeugt werden, die dann zum Endergebnis *projekt.exe* zusammengelinkt werden. Daraus ergibt sich die folgende Make-Datei, sie sei *m1.mak* genannt:

Listing 17.1: Einfache Make-Datei

```
# cppbuch/k17/m1.mak
# Regel 1
projekt.exe: rational.o main.o
→g++ -o projekt.exe rational.o main.o
```

```
# Regel 2
rational.o: rational.cpp rational.h
→ g++ -c rational.cpp

# Regel 3
main.o: main.cpp rational.h
→ g++ -c main.cpp

# Regel 4
clean:
→ rm -f rational.o main.o
```

make prüft zuerst, ob das erste Ziel *projekt.exe* (wenn kein anderes angegeben wird), überhaupt vorhanden ist. Wenn nicht, sucht *make* nach einer Regel zur Erzeugung dieses Ziels und findet Regel 1 mit der entsprechenden Aktion. Falls jedoch das Ziel existiert, werden Datum und Uhrzeit der Dateien rechts vom Doppelpunkt : geprüft, von denen das Ziel abhängt. Falls sie neuer als das Ziel sind, wird die Aktion ebenfalls ausgeführt, d.h. *projekt.exe* neu erzeugt.

Nun kann es aber sein, dass *main.o* oder *rational.o* noch nicht existieren. Dann sucht *make*, bevor die Aktion aus Regel 1 ausgeführt wird, nach einer Regel zur Erzeugung dieser Dateien und findet sie in den Regeln 2 und 3, deren Aktionen dann ausgeführt werden. Ein Aufruf *make -f m1.mak* würde die Datei *projekt.exe* erzeugen, verbunden mit der Bildschirmausgabe

```
g++ -c rational.cpp
g++ -c main.cpp
g++ -o projekt.exe rational.o main.o
```

Dem Parameter *-f* folgt der Dateiname des Makefiles. Wenn man ihn weglässt, sucht *make* nach den Dateien *makefile* oder *Makefile*. Die Wiederholung des Aufrufs würde nur die Meldung *make: "projekt.exe" ist bereits aktualisiert. ergeben* – schließlich hat sich ja auch nichts geändert. Falls nun zum Beispiel die Datei *rational.cpp* geändert wird, würden nur die Schritte

```
g++ -c rational.cpp
g++ -o projekt.exe rational.o main.o
```

ausgeführt; die erneute Übersetzung von *main.cpp* ist nicht erforderlich. Gerade bei großen, aus sehr vielen Dateien bestehenden Programmen erspart dieser Mechanismus viel Compilationszeit. *make* wählt sich stets das erste Ziel aus, es sei denn, ein anderes wird angegeben. So löscht der Aufruf

```
make -f m1.mak clean
```

die Objektdaten, die man abschließend nicht mehr benötigt und die bei Bedarf stets neu erzeugt werden können. Die Option *-f* unterdrückt Fehlermeldungen bei nicht existierenden Dateien. Mit dem Ziel *clean* ist eine Besonderheit verbunden: Es ist keine Datei, und es gibt auch keine Aktion, die so eine Datei erzeugt. *make* merkt das und legt die Datei nicht an. Man kann *make* die Prüfung ersparen, indem man die Zeile *.PHONY: clean* einfügt.

17.3 Variablen und Muster

Bei Projekten mit mehreren oder vielen Dateien ist es lästig, für jede `cpp`-Datei eine eigene Regel zu schreiben. Das haben die *make*-Entwickler erkannt: Die Regeln 2 und 3 oben können entfallen; sie werden von *make* durch *implizite Regeln* ergänzt. Dabei wird auch die Variable `CXXFLAGS` berücksichtigt. Das ist jedoch nicht ausreichend, wenn Sie dem Compiler bestimmte Informationen mittels weiterer Variablen mitgeben wollen.

Variablen haben den Vorteil, dass man die Definition ändern kann, ohne den Rest eines Makefiles bearbeiten zu müssen. *Muster* haben den Vorteil, nicht für jede Datei eine Regel schreiben zu müssen. Das Makefile `m2.mak` zeigt ein Beispiel, das nachfolgend erläutert wird:

Listing 17.2: Make-Datei mit Variablen und Mustern

```
# cppbuch/k17/m2.mak
.PHONY: clean

# Definition der Variablen
CXX := g++
CXXFLAGS := -c -g -Wall
objs := rational.o main.o

# Regel 1
projekt.exe: $(objs)
    ↪ $(CXX) -o $@ $^

# Regel 2
%.o: %.cpp rational.h
    ↪ $(CXX) $(CXXFLAGS) $<

clean:
    ↪ rm -f $(objs)
```

`CXX` ist eine Variable, die den Compiler bestimmt. Sie wird nachfolgend mit `$(CXX)` angesprochen. Damit kann nur durch Ändern dieser Variablen ein anderer Compiler benutzt werden. Der Name ist natürlich (relativ) frei wählbar, aber `CXX` ist üblich und bei GNU-make die Voreinstellung für den C++-Compiler. `CXXFLAGS` beschreibt an den Compiler zu übergebende Flags. Hier sollen außer der reinen Compilation (`-c`) Debug-Informationen erzeugt (`-g`) und alle Warnungen ausgegeben (`-Wall`) werden. `objs` gibt die zu erzeugenden Objektdateien an. Regel 1 zeigt die Verwendung der Dateien. Dabei sehen Sie auch merkwürdige Zeichen am Ende der Regeln 1 und 2 – dies sind von *make* bereitgestellte automatische Variablen mit den folgenden Bedeutungen:

- `$<` gibt die erste Abhängigkeit an.
- `$^` gibt alle Abhängigkeiten an.
- `$@` gibt das Ziel an.

Es gibt noch mehr mit dem `$`-Zeichen beginnende automatische Variablen, siehe [\[make\]](#). Man kann sich die Werte leicht ausgeben lassen, indem zum Beispiel bei Regel 1 die Zeilen

```

❯ @echo 1 $^
❯ @echo 2 $@
❯ @echo 3 $<

```

eingefügt werden. Diese drei Zeilen bewirken nach Löschen aller Objektdateien und dem Aufruf `make -f m2.mak` die Ausgabe

```

1 rational.o main.o
2 projekt.exe
3 rational.o

```

Regel 2 zeigt die Verwendung eines Musters, mit dem die implizite Regel, cpp-Dateien mit `g++ -c` zu übersetzen, neu definiert wird. Die Zeile

```
%o: %.cpp rational.h
```

bedeutet, dass jede Datei mit der Endung `.o` von einer ansonsten gleichnamigen Datei mit der Endung `.cpp` abhängt sowie von der Datei `rational.h`. An dieser Stelle `%h` zu schreiben, wäre falsch, weil es in diesem Beispiel keine Datei `main.h` gibt und zur Erzeugung von `main.o` die implizite Regel zur Geltung käme. Hier gibt es einen kleinen Schönheitsfehler zu sehen: Typischerweise sind die cpp-Dateien in einem Projekt von verschiedenen Header-Dateien abhängig. Wie dieses Problem gelöst wird, sehen Sie in Abschnitt 23.1. Die Zeile

```
❯ $(CXX) $(CXXFLAGS) $<
```

bedeutet, dass die jeweilige cpp-Datei übersetzt wird. Diese Datei ist die erste Datei, von der das Ziel `%o` abhängt, deswegen `$<` (siehe oben).

`SHELL` ist eine eingebaute Variable,² die mit `/bin/sh` für den Kommandointerpreter voreingestellt ist, aber geändert werden kann.

17.4 Universelles Makefile für einfache Projekte

Im obigen Makefile mussten noch die Objektdateien angegeben werden. Schöner wäre ein Makefile, dass sich selbst die Informationen zusammensucht. Dies sei in `m3.mak` gezeigt. Dabei sei angenommen, dass sich alle Dateien zu diesem Projekt (aber keine anderen *.cpp-Dateien!) im aktuellen Verzeichnis befinden. Es darf in den cpp-Dateien exakt eine `main()`-Funktion geben.

Listing 17.3: Universelles Makefile für einfache Projekte

```

# cppbuch/k17/m3.mak
.PHONY: all clean
CXX := g++
CXXFLAGS := -c -g -Wall

```

² `make -p`, in einem Verzeichnis *ohne Makefile* aufgerufen, zeigt alle eingebauten Variablen und mehr.

```

LDFLAGS := -g
HEADERS := $(wildcard *.h)
objs := $(patsubst %.cpp,%.o,$(wildcard *.cpp))

all: projekt.exe

projekt.exe: $(objs)
→ $(CXX) $(LDLAGS) -o $@ $^

%.o: %.cpp $(HEADERS) # (noch) nicht optimal, siehe unten
→ $(CXX) $(CXXFLAGS) $<

clean:
→ rm -f $(objs)

```

Diese Datei unterscheidet sich von der vorherigen Datei *m2.mak* in folgenden Punkten:

- Es gibt eine Variable `LDLAGS` für Informationen an den Linker.
- Es gibt eine Variable `HEADERS`, die mithilfe der Funktion `wildcard` gebildet wird, und die die Namen aller `.h`-Dateien des Verzeichnisses enthält.
- Die Variable `objs` wird mit Hilfe der Funktion `patsubst` gebildet. `patsubst` steht für »pattern substitution« und hat drei Parameter. `$(patsubst gesucht, ersatz, quelle)` bewirkt, dass jedes Auftreten von `gesucht` in `quelle` durch `ersatz` ersetzt wird. `objs` wird also gebildet, indem in der Liste der Namen aller `cpp`-Dateien die Endung `.cpp` durch `.o` ersetzt wird.
- Es gibt ein neues Ziel `all`, das nur von `projekt.exe`, der zu erzeugenden ausführbaren Datei, abhängt. Dies ist eigentlich nicht notwendig, folgt aber der Konvention, dass jedes Makefile ein Ziel `all` haben sollte, das die Übersetzung anstößt. Der Name `all` ist nicht ganz zutreffend, weil es andere Ziele geben kann, die gar nicht über `all` aktiviert werden. Zu dieser Gruppe gehören alle Ziele, die nicht so oft aufgerufen werden müssen, wie etwa das von oben bekannte Ziel `clean`. Es gibt weitere Ziele mit konventionellen Namen, die im obigen Makefile nicht enthalten sind. Eine Auswahl:
 - `install`: Das lauffähige Programm auf dem Computer (oder einem Web-Server) installieren.
 - `docs`: Die zugehörige Dokumentation erzeugen.
 - `check`: Tests des Programms laufen lassen.
 - `dist`: Eine Distribution erzeugen.
- Die zu erzeugenden Dateien hängen von der Variable `HEADERS` ab; es wird also alles neu übersetzt, wenn sich nur eine Header-Datei geändert hat – nicht optimal, aber bei kleinen Projekten tolerierbar. Im Abschnitt 23.1 sehen Sie, wie die tatsächlichen Abhängigkeiten automatisch ermittelt werden können.

Mit *m3.mak* liegt ein universelles Makefile vor, das Sie für einfache Projekte einsetzen können. Falls Sie die Grundlagen vertiefen möchten oder Tipps für spezielle Problemstellungen suchen, empfehle ich Ihnen einen Blick in das Kapitel 23.

18

Unit-Test

Dieses Kapitel behandelt die folgenden Themen:

- Unit-Test
- Werkzeuge
- Test Driven Development
- Boost Unit Test Framework

Das Testen von Software ist aufwendig: Es nimmt typischerweise etwa ein Drittel des gesamten Software-Entwicklungsaufwandes ein. Der Anteil ist bei sicherheitskritischen Systemen noch erheblich höher. Es gibt verschiedene Arten von Tests: Modul(Unit)-Test, Integrationstest, Systemtest, Abnahmetest. So dient der Abnahmetest dazu, die Funktionalität des Systems bzw. der Software dem Auftraggeber gegenüber nachzuweisen. Ein Bestehen dieses Tests ist in der Regel vertragliche Voraussetzung für die Bezahlung des Werks. Testen hat zwei Aufgaben:

- **Fehler finden:** Je früher ein Fehler erkannt und beseitigt wird, desto besser. Wenn Sie testen, legen Sie sich eine sehr kritische Haltung gegenüber dem zu testenden Prüfling zu! Seien Sie geradezu »sadistisch« und versuchen Sie, das Programm mit Ihren Testfällen möglichst zum Absturz und zu falschen Ergebnissen zu bringen. Unbewusst oder bewusst Schwächen eines Programms auszublenken, spart zwar kurzfristig Zeit, gibt aber langfristig Ärger.

- **Qualität nachweisen:** Qualität ist der Grad, in dem ein System bzw. eine Software Anforderungen erfüllt. Bei dem oben genannten Abnahmetest liegen die Anforderungen in schriftlich festgelegter Form vor (Spezifikation/Pflichtenheft). Es geht dabei nicht nur um funktionale Anforderungen (etwa korrekte Berechnung), sondern auch um nicht-funktionale, zum Beispiel Berechnung innerhalb einer maximalen Dauer, Robustheit gegen fehlerhafte Daten usw.

Eine Übersicht über Softwaretests geben [SL]. Dem Softwareentwickler am nächsten ist der *Unit-Test*. Dabei testet jeder Entwickler selbst die von ihm programmierte Einheit (englisch *unit*), bevor er sie weitergibt. Der Vorteil: Ein Entwickler kennt seine Software genau – und kann deshalb leichter als andere Fehler finden. Der Nachteil: Ein Entwickler kennt seine Software genau – und ist deswegen »betriebsblind«, das heißt, er sieht sein eigenes Produkt zu unkritisch. Es mag unbewusste emotionale Widerstände geben, weswegen Tests auf höherer Ebene nicht von denselben Personen geplant und ausgeführt werden sollen, die die Software entwickelt haben. Erst nach bestandenem Unit-Test ist der Programmteil bereit zu Integration mit anderen Komponenten. Der Integrationstest dient zur Prüfung, ob alle integrierten Komponenten wie beabsichtigt zusammenspielen.

18.1 Werkzeuge

Ein Programm mal eben auszuprobieren, ist kein Testen. Testfälle müssen systematisch mithilfe von Testverfahren spezifiziert werden, zum Beispiel Äquivalenzklassenbildung. Darunter versteht man die Aufteilung von Tests in Klassen, die äquivalent sind, also Gemeinsamkeiten haben. Ein Beispiel für den Test eines Sortierprogramms: Die Testfolgen »1, 3, 7, 4, 5, 8« und »9, 2, 22, 6, 1« gehören zur Äquivalenzklasse »alle Zahlen sind unterschiedlich«, im Gegensatz zur Folge »100, 100, 4, 40, 9, 7«, die mindestens zwei gleiche Zahlen enthält. Zum Finden des Fehlers im Sortierprogramm der Aufgabe von Seite 144 ist gerade dieser Unterschied wichtig. Äquivalenzklassen reduzieren die Anzahl von Tests in den Fällen, in denen die Zahl *aller möglichen* Testfälle einfach zu groß ist, um noch vernünftig handhabbar zu sein. Schon zwanzig voneinander unabhängige *if*-Anweisungen können sich zu über einer Million (genau: 2^{20}) verschiedener Möglichkeiten der Ausführung eines Programms addieren. Oder, um das Beispiel des Sortierprogramms wieder aufzugreifen: Wenn *alle* möglichen Kombinationen einer Folge von N Werten getestet werden sollen, ist der Test bereits bei einem kleinen N (zum Beispiel 20) nicht mehr durchführbar.

Die notwendige Anzahl von Testfällen kann also recht groß werden, schon bei Software mittlerer Größe. Weil nach einer Änderung der Software möglicherweise auch entfernte Softwareteile betroffen sind, müssen alle Tests wiederholt werden (Regressionstest). Das lässt sich nur dann ökonomisch bewerkstelligen, wenn der Testprozess automatisiert abläuft.

Der Test selbst ist ebenfalls Software, die programmiert werden muss. Da wesentliche Elemente des Testens wiederkehren, ist es sehr empfehlenswert, Frameworks für die Programmierung der Tests einzusetzen und auf eine rein individuelle Lösung zu verzichten.

Für diese Frameworks hat sich der Name XUnit eingebürgert, in Anlehnung an JUnit, ein Unit-Test-Framework für die Programmiersprache Java. Dabei steht X für den Kontext, zum Beispiel DB für ein Framework zum Testen von Datenbankanwendungen. Es gibt mehrere Frameworks für Unit-Tests in C++. Ich stelle Ihnen in Auszügen das Boost Unit Test Framework vor, nicht nur, weil die Boost-Libraries den C++-Standard beeinflusst haben, sondern auch, weil sie portabel und für ihre Qualität bekannt sind. Außerdem verwende ich in diesem Buch an mehreren Stellen die Boost Libraries (Threads, reguläre Ausdrücke, Internet-Anbindung) und sehe keinen Anlass zu wechseln.

18.2 Test Driven Development

Mit dem Aufkommen der agilen Softwareentwicklung (<http://agilemanifesto.org/>), besonders befördert durch das Extreme Programming (XP)-Buch [Beck], nimmt die Bedeutung der testgetriebenen Entwicklung (englisch *TDD = Test Driven Development*) zu. Dabei wird zuerst ein Testfall spezifiziert, also *bevor* der zu prüfende Code überhaupt existiert. Der Testfall wird mit einem XUnit-Werkzeug ausgeführt und schlägt natürlich fehl. Anschließend entsteht nach und nach der Programmcode, wobei der Test wiederholt ausgeführt wird – solange, bis er bestanden wird. Im nachfolgenden Schritt wird der nächste Testfall spezifiziert und es wird der Ablauf wiederholt. Dabei werden auch alle vorherigen Testfälle ausgeführt, um sicher zu sein, dass eine Änderung nicht andere Programmteile ungünstig beeinflusst. Dieses Vorgehen hat einige Vorteile:

- Die Planung der Testfälle setzt eine gründliche Auseinandersetzung mit der Anforderungsdefinition und dem Design voraus. Dabei entstehende Unklarheiten und Widersprüche werden noch vor der Codierung beseitigt.
- Die Testfälle dienen als Spezifikation für die zu erstellende Software. Damit reduziert sich der nachträgliche Testaufwand.
- Es wird keine Software geschrieben, die nicht gebraucht wird. Ich habe gelegentlich beobachtet, dass bei dem Schreiben einer Klasse in guter Absicht möglicherweise nützliche Methoden gleich mitprogrammiert werden, ohne dass klar ist, ob sie jemals gebraucht werden. In der industriellen Wirklichkeit führt dies zu unnötigen Kosten – jede Methode muss zusätzlich getestet und dokumentiert werden.
- Die regelmäßigen Regressionstest garantieren bei Erfolg, dass die Software ein (durch die Testfälle definiertes) Mindestmaß an Qualität besitzt.
- Der in manchen Projekten am Ende zu beobachtende verstärkte Zeitdruck führt zu Aussagen wie »Zum ausführlichen Testen haben wir keine Zeit mehr!«, verbunden mit teuren Änderungen nach Auslieferung der Software. Die Wahrscheinlichkeit für solche Probleme ist bei der testgetriebenen Entwicklung wegen der ständig mitlaufenden Tests gering.

Ein testgetrieben entwickeltes System ist normalerweise von höherer Qualität als ein auf herkömmliche Art entwickeltes (Test nach Abschluss der Programmierung). Daraus folgt jedoch nicht unbedingt, dass es ausreichend getestet ist, nämlich dann, wenn die

Testfälle nicht systematisch auf Basis der Anforderungsdefinition und unter Verwendung guter Testverfahren entwickelt wurden. Das Qualitätsproblem verlagert sich von der Programmierung auf die Testfallspezifikation. Ein weiterer negativer Aspekt ist die isolierte Betrachtung des aktuellen Testfalls. Bei sofortiger Kenntnis aller Anforderungen an eine Komponente ist möglicherweise ein besseres Design möglich, und es müssten in eine Klasse nicht nachträglich viele Änderungen vorgenommen werden. Diese Argumente sprechen nicht gegen die testgetriebene Entwicklung, wenn man sie berücksichtigt. In Abschnitt 18.3.1 unten finden Sie ein ausführliches Beispiel für die testgetriebene Entwicklung einer Operatorfunktion.

18.3 Boost Unit Test Framework

Das Boost Unit Test Framework stellt Komponenten zur Verfügung, die das Schreiben von Tests, die Organisation von Tests und den kontrollierten Ablauf erlauben. Obwohl das Framework aus vielen Klassen besteht, ist die einfachste Schnittstelle zur Benutzung ein Satz von Makros. Mit wenigen Ausnahmen werde ich mich im Folgenden darauf beschränken, weil mit den Makros die wichtigsten Bereiche abgedeckt werden. Allen, die mehr wissen möchten, empfehle ich [Roz].

Installation

Das Boost Unit Test Framework wird automatisch mitinstalliert, wenn Sie die Boost Libraries entsprechend den Hinweisen auf der DVD einrichten.

Compilations- und Link-Optionen

Es wird davon ausgegangen, dass die Boost-Libraries installiert sind. Die Optionen sind:

- Statisches Linken¹ der Boost-Test-Library.
- Dynamisches Linken der Boost-Test-Library.
- »Header-only«: Damit ist gemeint, dass an einer Stelle die Quelltexte aller benötigten Funktionen mit einer `#include`-Anweisung eingebunden werden. Der Nachteil: Die Compilation dauert deutlich länger. Der Vorteil: Beim Linken muss keine Bibliothek angegeben werden. Alle Funktionen werden statisch zur ausführbaren Datei gebunden.
- »Auto-Linking«: Eine spezielle Möglichkeit des Frameworks für Microsoft Compiler, die zu linkenden Teile automatisch zu ermitteln.

Wegen der schnellen Compilation und der Verwendung des GNU C++-Compilers habe ich in den Beispielen die zweite Möglichkeit gewählt und die Makefiles entsprechend konfiguriert. Es genügt, in einem Shell-Fenster `make` einzugeben, um alles zu übersetzen und zu linkern. In den Quellprogrammen sind die passenden Flags und `#include`-Anweisungen zu setzen, was allerdings sehr einfach ist, wie hier zu sehen:

¹ Zum Begriff »Linken« siehe Glossar Seite 953.

```
// Header-only (compiliert langsamer, statisches Linken)
#include<boost/test/included/unit_test.hpp>
// ... Rest der Datei
```

```
// dynamisches Linken erwünscht:
#define BOOST_TEST_DYN_LINK
#include<boost/test/unit_test.hpp>
// ... Rest der Datei
```

Testaufbau

Eine *Test-Suite* besteht aus einer Menge von *Testfällen*. Das Makro `BOOST_AUTO_TEST_SUITE( suite_name )` startet eine Test-Suite, mit `BOOST_AUTO_TEST_SUITE_END()` wird sie beendet. Jedes `BOOST_AUTO_TEST_CASE`-Makro fügt einen Testfall hinzu. Wenn `BOOST_TEST_MAIN` definiert ist, wird automatisch eine `main()`-Funktion erzeugt. Das folgende Beispiel zeigt eine Test-Suite mit zwei einfachen Testfällen. Im ersten wird die Funktion `length()` der Klasse `string` geprüft, im zweiten ihr Gleichheitsoperator. `BOOST_CHECK(bedingung)` prüft die Bedingung und gibt eine Fehlermeldung aus, wenn die Bedingung nicht erfüllt ist.

Listing 18.1: Einfache Test-Suite

```
#define BOOST_TEST_MAIN
// dynamisches Linken erwünscht:
#define BOOST_TEST_DYN_LINK
#include<boost/test/unit_test.hpp>

BOOST_AUTO_TEST_SUITE( einfacher_stringtest )

BOOST_AUTO_TEST_CASE( laenge )
{
    std::string s("xyz");
    BOOST_CHECK( s.length() == 3 );
}

BOOST_AUTO_TEST_CASE( gleichheits_operator )
{
    std::string s("abc");
    BOOST_CHECK( s == "abc" );
}

BOOST_AUTO_TEST_SUITE_END()
```

Weil `main()` automatisch erzeugt wird, ist die Test-Suite nach Compilation lauffähig. Die Ausgabe des Programms ist

```
Running 2 test cases...
*** No errors detected
```

Falls nun fälschlicherweise 4 statt 3 im ersten Test eingetragen wird, ist die Ausgabe

```
Running 2 test cases...
main.cpp(11): error in "laenge": check s.length() == 4 failed
*** 1 failure detected in test suite "Master Test Suite"
```

Die Zahl in Klammern gibt die Zeilennummer der Testdatei an, in der der Fehler auftrat. Wie Sie sehen, bedeutet ein Fehlschlagen des Tests nicht unbedingt, dass der Prüfling einen Fehler hat – es kann auch der Test falsch sein. Die Wurzel des Testfallbaums ist die »Master Test Suite«, die mehrere Test-Suiten enthalten kann. Der Name kann geändert werden, wenn statt `BOOST_TEST_MAIN` zum Beispiel `BOOST_TEST_MODULE never_name` geschrieben wird. Die Prüfung einer einfachen Bedingung gibt es in drei Varianten:

- `BOOST_WARN(Bedingung)`: Die Nichterfüllung der Bedingung wird nicht als Fehler gesehen und auch nicht als solcher gezählt. Bei der Standardeinstellung gibt es keine Meldung, erst wenn die ausführbare Datei mit einem passenden Log-Level aufgerufen wird, zum Beispiel `testprog.exe --log_level=warning`. Das Testprogramm läuft weiter.
- `BOOST_CHECK(Bedingung)`: Die Nichterfüllung der Bedingung wird als Fehler gesehen und gemeldet. Das Testprogramm läuft weiter.
- `BOOST_REQUIRE(Bedingung)`: Die Nichterfüllung der Bedingung wird als so kritischer Fehler gesehen, dass eine Fortsetzung des Tests nicht sinnvoll ist. Das Testprogramm bricht ab.

Die Informationen dieses Abschnitts genügen, um die Methode der testgetriebenen Entwicklung am Beispiel zu zeigen, wie Sie im nächsten Abschnitt sehen. Im Anschluss daran werden weitere Möglichkeiten des Frameworks vorgestellt und demonstriert.

18.3.1 Beispiel: Testgetriebene Entwicklung einer Operatorfunktion

In diesem Beispiel gehe ich von der Klasse `Datum` des Abschnitts 9.3 ab Seite 334 aus und nehme an, dass der `operator++()` geschrieben werden soll, also im Gegensatz zum genannten Abschnitt noch nicht existiert. Der erste Testfall ist die Prüfung, ob das Hochzählen des Tages funktioniert:

Listing 18.2: `operator++()`-Test

```
// cppbuch/k18/tdd/main.cpp
#define BOOST_TEST_MAIN
#define BOOST_TEST_DYN_LINK
#include<boost/test/unit_test.hpp>
#include"datum.h"

BOOST_AUTO_TEST_SUITE( datum_operator_plus_plus )

BOOST_AUTO_TEST_CASE( tag )
{
    Datum d(1, 1, 2011);
    for(int tag = 2; tag <= 31; ++tag) {
        ++d;
        BOOST_CHECK( d.tag() == tag );
    }
}

BOOST_AUTO_TEST_SUITE_END()
```

Dieser Testfall kann leicht mit der Funktion

Listing 18.3: Unvollständiger operator++(), Version 0

```
Datum& Datum::operator++() { // Datum um 1 Tag erhöhen
    ++tag_;
    return *this;
}
```

erfüllt werden. Der nächste Testfall prüft, ob das Weiterschalten des Monats funktioniert:

```
BOOST_AUTO_TEST_CASE( einfacher_monatswechsel )
{
    Datum d(31, 1, 2011);
    ++d;
    BOOST_CHECK( d.tag() == 1 );
    BOOST_CHECK( d.monat() == 2 );
}
```

Der Testfall scheitert natürlich. Die Ausgabe des Testprogramms ist:

```
Running 2 test cases...
main.cpp(25): error in "monatswechsel": check d.tag() == 1 failed
main.cpp(26): error in "monatswechsel": check d.monat() == 2 failed
*** 2 failures detected in test suite "master_testsuite"
```

Die Zahlen nach main.cpp geben die Zeilennummern des Testprogramms an, in denen der Fehler aufgetreten ist. Die geänderte Funktion löst scheinbar das Problem:

Listing 18.4: Unvollständiger operator++(), Version 1

```
Datum& Datum::operator++() {
    int tmp[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    if(tag_ == tmp[monat_-1]) {
        tag_ = 1;
        ++monat_;
    }
    else
        ++tag_;
    return *this;
}
```

Funktioniert das für alle Monate? Das offensichtliche Schaltjahrproblem hebe ich für später auf. Der folgende Testfall gibt Auskunft:

```
BOOST_AUTO_TEST_CASE( alle_monatswechsel )
{
    int monate[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    for(int monat = 1; monat <= 12; ++monat) {
        Datum d(monate[monat-1], monat, 2011);
        ++d;
        BOOST_CHECK( d.tag() == 1 );
        if(monat < 12) {
            BOOST_CHECK( d.monat() == monat+1 );
        }
        else {
            BOOST_CHECK( d.monat() == 1 );
        }
    }
}
```



```

    }
}

```

Dieser Test bringt an den Tag, dass die letzte Lösung noch nicht in Ordnung ist. Ein kleiner Einschub direkt nach Inkrementieren des Monats beseitigt das Problem:

```

    if(monat_ == 13) {
        monat_ = 1;
        ++jahr_;
    }

```

Auf den Test, ob das Jahr korrekt hochgezählt wird, verzichte ich aus Platzgründen. Nun bleibt noch das Schaltjahr. Dazu hilft der folgende Testfall, in dem `BOOST_CHECK` durch `BOOST_REQUIRE` ersetzt wird, damit der Test bei der großen Anzahl getesteter Jahre nach dem ersten Fehler abbricht.

```

BOOST_AUTO_TEST_CASE( schaltjahr )
{
    for(int jahr = 1800; jahr < 2300; ++jahr) {
        bool schaltjahr = // durch 4 teilbar, aber nicht durch 100, es sei denn, durch 400
            ((jahr % 4 == 0) && (jahr % 100 != 0)) || (jahr % 400 == 0);
        Datum d(28, 2, jahr);
        ++d;
        if(schaltjahr) {
            BOOST_REQUIRE( d.tag() == 29 );
            BOOST_REQUIRE( d.monat() == 2 );
        }
        else {
            BOOST_REQUIRE( d.tag() == 1 );
            BOOST_REQUIRE( d.monat() == 3 );
        }
    }
}

```

Nun könnte man im `operator++()` das Schaltjahr explizit berücksichtigen, ähnlich wie im Testfall. Geschickter ist es jedoch, die Prüfung auf die Eigenschaft »Schaltjahr« durch eine eigene, unabhängig aufrufbare Funktion `istSchaltjahr()` zu realisieren, die direkt nach der Definition des Feldes `tmp` aufgerufen wird:

Listing 18.5: `operator++()`

```

Datum& Datum::operator++() {
    int tmp[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    if(istSchaltjahr()) {
        tmp[1] = 29;
    }
    if(tag_ == tmp[monat_-1]) {
        tag_ = 1;
        ++monat_;
        if(monat_ == 13) {
            monat_ = 1;
            ++jahr_;
        }
    }
}

```

```

    else {
        ++tag_;
    }
    return *this;
}

```

Damit ist `operator++()` vollständig. Wie sinnvoll ein Regressionstest ist, zeigt sich an dem folgenden Optimierungsversuch. Bei jedem Aufruf wird das Array `tmp` neu auf dem Laufzeit-Stack abgelegt. Um das zu vermeiden, wird es als `static` deklariert.

```

Datum& Datum::operator++() {
    // static vermeidet Neuinitialisierung:
    static int tmp[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; // ???
    if(istSchaltjahr()) {
        tmp[1] = 29;
    }
    // usw.
}

```

Überlegen Sie, ob das wirklich eine gute Idee ist, bevor Sie weiterlesen. Was geschieht? Der Test scheitert! Der Grund liegt in der `if`-Anweisung, die `tmp` dauerhaft verändert. Wenn beim nächsten Test kein Schaltjahr vorliegt, wird dennoch 29 angenommen. Der wiederholte Testfall `schaltjahr` hat sehr schnell gezeigt, dass die mal eben eingeführte »Verbesserung« fehlerhaft ist. Das `static`-Feld ist zudem nicht thread-sicher, weil auf das Feld von zwei Programmen gleichzeitig zugegriffen werden kann. Korrekturen: Feld als `const` anlegen, um es thread-sicher zu machen, und dann (natürlich) nur lesend darauf zugreifen. Ergebnis:

Listing 18.6: `operator++()` mit `static`-Feld

```

Datum& Datum::operator++() {
    static const int tmp[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    int letzterTagImMonat = tmp[monat_-1];
    if(monat_ == 2 && istSchaltjahr()) {
        letzterTagImMonat = 29;
    }
    if(tag_ == letzterTagImMonat) {
        tag_ = 1;
        ++monat_;
        if(monat_ == 13) {
            monat_ = 1;
            ++jahr_;
        }
    }
    else {
        ++tag_;
    }
    return *this;
}

```

Die isolierte Sicht auf einen Testfall versperrt möglicherweise den Blick auf eine andere Möglichkeit. Wenn es die Anforderung gibt (und sie schon bekannt ist), dass der Konstruktor der Klasse `Datum` und die Methode `set(int t, int m, int j)` eine Exception

werfen sollen, wenn ein ungültiges Datum verwendet wird, liegt es nahe, eine Prüfmethode `istGueultigesDatum(int t, int m, int j)` zu schreiben, die von allen anderen Methoden aufrufbar ist. Sie finden sie auf Seite 336. Davon profitiert auch `operator++()`, der nun deutlich kürzer geschrieben werden kann. Es wird ausgenutzt, dass es sich um ein Monatsende handeln muss, wenn das Datum nach dem Hochzählen des Tages ungültig wird.

Listing 18.7: Kürzerer `operator++()`

```
Datum& Datum::operator++() {
    ++tag_;
    if(!istGueultigesDatum(tag_, monat_, jahr_)) { // Monatsende erreicht?
        tag_ = 1;
        if (++monat_ > 12) {
            monat_ = 1;
            ++jahr_;
        }
    }
    return *this;
}
```

18.3.2 Fixture

Im JUnit-Test-Sprachgebrauch bezeichnet Fixture durchzuführende vorbereitende Maßnahmen, mit `setUp()` aufgerufen, und nachbereitende Maßnahmen (Aufräumarbeiten), mit dem Namen `tearDown()` verbunden. Diese Maßnahmen können zum Beispiel dafür sorgen, dass vor jedem Testfall dieselben Bedingungen herrschen. Die Durchführung eines Testfalls besteht aus vier Phasen:

- Vorbereitende Maßnahmen (`setUp`)
- Test durchführen
- Ergebnis prüfen und protokollieren
- Aufräumarbeiten (`tearDown`)

Der Vorteil der Verwendung von Fixtures besteht in der Trennung der vor- und nachbereitenden Maßnahmen vom eigentlichen Test. Das C++-Prinzip »Resource Acquisition Is Initialization« (RAII, siehe Glossar) erlaubt es, auf die Methoden `setUp()` und `tearDown()` zu verzichten. An ihre Stelle treten Konstruktor und Destruktor. Ein einfaches Beispiel: Wenn ein `Datum`-Objekt dynamisch angelegt werden soll, lässt sich das leicht mit einem `shared_ptr<Datum>` als Fixture realisieren:

```
BOOST_AUTO_TEST_CASE( konstruktor_mit_fixture )
{
    // shared_ptr<Datum> als Fixture
    std::shared_ptr<Datum> p(new Datum(1, 1, 2011)); // Konstruktor (setUp)
    BOOST_CHECK( p->jahr() == 2011 );
} // delete automatisch durch shared_ptr-Destruktor (tearDown)
```

Im folgenden Abschnitt wird ein Fixture auf globaler Ebene zum Öffnen und Schließen einer Log-Datei eingesetzt.

18.3.3 Testprotokoll und Log-Level

Es gibt zwei Möglichkeiten, eine Datei als Testprotokoll zu erzeugen. Die erste ist, die Bildschirmausgabe in eine Datei umzuleiten. Dabei kann der Log-Level als Parameter übergeben werden. Beispiel:

```
testprog.exe --log_level=error
```

Anstelle der Übergabe in der Kommandozeile kann dasselbe mit einer entsprechenden Definition der Umgebungsvariablen `BOOST_TEST_LOG_LEVEL` erreicht werden. Die Spalte eins der Tabelle 18.1 enthält weitere Werte für den Log-Level.

Die zweite Möglichkeit ist, per Programm eine Log-Datei anzulegen und den Log-Level zu definieren. Dazu wird eine Klasse mit den entsprechenden Einstellungen definiert:

```
class LogKonfiguration {
public:
    LogKonfiguration()
        : test_log("test.log") { // Dateiname
        boost::unit_test::unit_test_log.set_stream(test_log);
        boost::unit_test::unit_test_log.set_threshold_level(
            boost::unit_test::log_test_units); // Log-Level
    }
    ~LogKonfiguration() { // schließen und zurücksetzen
        test_log.close();
        boost::unit_test::unit_test_log.set_stream(std::cout);
    }
private:
    std::ofstream test_log;
};
```

Ein Objekt dieser Klasse wird mit

```
BOOST_GLOBAL_FIXTURE( LogKonfiguration );
```

in der Testdatei als globales Fixture erzeugt. Die Spalte zwei der Tabelle 18.1 enthält die möglichen, per Programm setzbaren Log-Level. Ein Log-Level schließt alle Ausgaben der Log-Level der darunter liegenden Tabellenzeilen ein.

Tabelle 18.1: Log-Level

Kommandozeile bzw. BOOST_TEST_LOG_LEVEL	per Programm festlegbar	protokolliert wird
success	log_successful_tests	alles
all		alles
test_suite	log_test_units	Suites und Testfälle
message	log_messages	BOOST_TEST_MESSAGE-Nachrichten
warning	log_warnings	Warnungen
error	log_all_errors	Fehler
cpp_exception	log_cpp_exception_errors	nicht gefangene Exceptions
system_error	log_system_errors	Systemfehler
fatal_error	log_fatal_errors	von Testmakros erkannte kritische Fehler oder fatale Systemfehler
nothing	log_nothing	nichts

Wenn eine Log-Datei mit dem Fixture angelegt wird, werden Log-Level-Einstellungen per Kommandozeilen-Option ignoriert. Die per Programm einstellbaren Log-Level sind im Namespace `boost::unit_test`.

18.3.4 Prüf-Makros

Sie haben schon verschiedene Prüf-Makros kennen gelernt, es gibt aber noch mehr für verschiedene Zwecke. Vielen der Makros ist gemeinsam, dass es sie für die drei Level `WARN`, `CHECK` und `REQUIRE` gibt, die oben auf Seite 530 beschrieben werden. In diesen Fällen wird im Folgenden einfach `<level>` als Platzhalter für eine der drei Varianten geschrieben, ohne weiter auf sie einzugehen.



Hinweis

Für alle Beispiele wird die Klasse `Datum` aus Abschnitt 9.3 zugrundegelegt, einschließlich der Lösungen der Übungsaufgaben. Die Klasse `Datum` und Beispiele befinden sich im Verzeichnis `cppbuch/k18/datum`. Die Testdatei ist `main.cpp`.

BOOST_<level>(Bedingung)

prüft die Bedingung (Beispiel siehe oben, Seite 529).

BOOST_<level>_MESSAGE(Bedingung, Text)

liefert bei ungültiger Bedingung den übergebenen Text.

BOOST_ERROR(Text) und BOOST_FAIL(Text)

verhalten sich wie `BOOST_CHECK_MESSAGE(false, Text)` und `BOOST_REQUIRE_MESSAGE(false, Text)`.

BOOST_<level>_PREDICATE(Prädikat, Argumente)

prüft das Prädikat. Im Unterschied zu `BOOST_<level>()` wird eine Funktion oder ein Funktor, gegebenenfalls mit Parametern, übergeben. Jeder Parameter muss von runden Klammern umschlossen sein. Beispiel für die Funktion `istSchaltjahr(int)`:

```
BOOST_AUTO_TEST_CASE( schaltjahrtest )
{
    BOOST_CHECK_PREDICATE( istSchaltjahr, (2012) );
}
```

BOOST_TEST_MESSAGE(Text)

ist eine Dokumentationshilfe. Das Makro gibt den übergebenen Text aus. Ein Beispiel sehen Sie unten bei `BOOST_<level>_THROW`.

BOOST_AUTO_TEST_CASE_EXPECTED_FAILURES(testfall, Anzahl)

definiert die Anzahl der zu erwartenden, das heißt, absichtlich hervorgerufenen Fehler für den Testfall. Die ausgegebene Fehleranzahl eines Testfalls wird um diese Zahl reduziert. Ein Beispiel sehen Sie unten bei `BOOST_<level>_THROW`.

BOOST_<level>_THROW (Ausdruck, Exceptiontyp)

prüft, ob der Ausdruck eine Exception des Typs `Exceptiontyp` oder einer davon abgeleiteten Klasse wirft. Falls keine Exception geworfen wird, gibt es einen Fehler. Beispiel:

```
// in der letzten Zeile provozierten Fehler nicht mitzählen:
BOOST_AUTO_TEST_CASE_EXPECTED_FAILURES( konstruktor_ungueltiges_Datum, 1 )

BOOST_AUTO_TEST_CASE( konstruktor_ungueltiges_Datum )
{
    BOOST_CHECK_THROW( Datum(30, 2, 2011), UngueltigesDatumException );
    BOOST_TEST_MESSAGE( "Gegenprobe: " ); // soll fehlschlagen, kein Fehler
    BOOST_CHECK_THROW( Datum(1, 2, 2011), UngueltigesDatumException );
}
```

BOOST_<level>_EXCEPTION (Ausdruck, Exceptiontyp, Prüffunktion)

wirkt wie `BOOST_<level>_THROW` mit dem Unterschied, dass die geworfene Exception der Prüffunktion übergeben wird. Gibt diese Funktion `false` zurück, ist der Test fehlgeschlagen; gibt sie `true` zurück, ist er bestanden. Die Prüffunktion kann zum Beispiel prüfen, ob der Text der Exception korrekt ist oder ob sie überhaupt vom richtigen Typ ist. Um Letztes zu zeigen, seien die folgende Klasse und Funktion vorausgesetzt:

```
class FalscheException : public std::exception {};

bool checkException(const std::exception& e) {
    return typeid(e) == typeid(UngueltigesDatumException);
}
```

Der Test

```
BOOST_CHECK_EXCEPTION( Datum(30, 2, 2011), UngueltigesDatumException,
                      checkException);
```

wird bestanden. Wenn aber das Makro wie gezeigt verändert wird, schlägt der Test fehl.

```
BOOST_CHECK_EXCEPTION( Datum(30, 2, 2011), FalscheException, checkException);
```

BOOST_<level>_NO_THROW(Ausdruck)

schlägt fehl, wenn der Ausdruck eine Exception wirft. Zum Beispiel schlägt

```
BOOST_CHECK_NO_THROW( Datum(30, 2, 2011));
```

fehl. Es ist möglich, mehrere Anweisungen anstelle des Ausdrucks auszuführen, wenn sie in einen `do-while(0)`-Block gepackt werden:

```
BOOST_CHECK_NO_THROW( do {
    int tag = 30;
    Datum(tag, 2, 2011);
} while(0));
```

BOOST_TEST_CHECKPOINT(Nachricht)

Dieses Makro ist sehr hilfreich zum Aufspüren von Fehlern. Die Nachricht kann so gewählt werden, dass ein direkt zu einem Fehler führender Wert ausgegeben wird. In allen anderen Fällen tut das Makro nichts. Ein Beispiel:

```
BOOST_AUTO_TEST_CASE( konstruktor_ungueltiges_Datum_checkpoint )
{
    for(int tag = 1; tag < 30; ++tag) {
        BOOST_TEST_CHECKPOINT( "Datum mit Tag = " << tag );
        Datum d(tag, 2, 2011); // Exception bei tag == 29: Fehler
    }
}
```

Am Ende der Schleife, wenn tag den Wert 29 annimmt, gibt es eine Exception, weil das zu erzeugende Datum ungültig ist. Das Testprogramm erzeugt folgende Ausgabe:

```
unknown location(0): fatal error in "konstruktor_ungueltiges_Datum_checkpoint": std::runtime_error: 29.02.2011 ist ein ungueltiges Datum!
main.cpp(132): last checkpoint: Datum mit Tag = 29
```

Die letzte Zeile weist auf den fehlerhaften Wert hin. Ohne das Makro BOOST_TEST_CHECKPOINT würde diese Zeile fehlen. In diesem Beispiel ist der Fehler leicht zu sehen. In anderen Fällen, zum Beispiel wenn sehr viele Daten in einer Schleife eingelesen und verarbeitet werden, ist es hilfreich, wenn ein fehlererzeugender Datensatz dokumentiert wird.

Bei meinen Tests unter Linux und Windows XP funktionierte BOOST_TEST_CHECKPOINT wie erwartet. Windows Vista, Business Edition, schmeckte das Programm nicht: Es tat sich ein Fenster auf mit der Information »testdatum.exe funktioniert nicht mehr. Es wird nach einer Lösung gesucht.« Natürlich wird keine gefunden, und das Programm kann durch Button-Klick beendet werden.

Relationale Makros

Diese Makros vergleichen zwei Werte links und rechts mit relationalen Operatoren. Der Unterschied zu BOOST_<level> werden im Falle eines false-Ergebnisses die verglichenen Werte ausgegeben. Die Tabelle 18.2 zeigt die relationalen Makros.

Tabelle 18.2: Relationale Makros

Makro	Wirkung wie
BOOST_<level>_EQUAL(links, rechts)	BOOST_<level>(links == rechts)
BOOST_<level>_NE(links, rechts)	BOOST_<level>(links != rechts)
BOOST_<level>_GE(links, rechts)	BOOST_<level>(links >= rechts)
BOOST_<level>_GT(links, rechts)	BOOST_<level>(links > rechts)
BOOST_<level>_LE(links, rechts)	BOOST_<level>(links <= rechts)
BOOST_<level>_LT(links, rechts)	BOOST_<level>(links < rechts)

Die Makros `BOOST_<level>_EQUAL` und `BOOST_<level>_NE` sollen *nicht* für `float`- und `double`-Werte genommen werden, weil bitweise verglichen wird. Weitere Informationen zum Vergleich von `float`- und `double`-Werten finden Sie auf Seite 561. Dazu passende Makros finden Sie unten.

BOOST_<level>_CLOSE(Links, Rechts, Toleranz)

Dieses Makro prüft, ob die ersten zwei Werte relativ (nicht absolut) dicht beieinanderliegen. Der dritte Wert gibt die Toleranz in *Prozent* an.

```
BOOST_CHECK_CLOSE( 1.25, 1.27, 1.0); // schlägt fehl
BOOST_CHECK_CLOSE( 1.25, 1.27, 2.0); // ok, Differenz ist kleiner als 2 %
```

Nach `boost/test/floating_point_comparison.hpp` ist der Test dann erfolgreich, wenn

```
(D/fabs(links) <= fabs(toleranz)) && (D/fabs(rechts) <= fabs(toleranz))
```

gilt, wobei `D` der Absolutbetrag der Differenz `links - rechts` ist. `links` und `rechts` müssen vom selben Typ sein. Das in Abschnitt 20.2.1 angesprochene Overflow-/Underflow-Problem wird durch Einsetzen des Maximalwerts für den Datentyp bzw. 0 gelöst.

BOOST_<level>_CLOSE_FRACTION(Links, Rechts, Toleranz)

arbeitet wie `BOOST_<level>_CLOSE`, nur dass die Toleranz anders definiert wird. Der Test ist erfolgreich, wenn `fabs(toleranz) > (fabs(rechts/links) - 1)` ist.

BOOST_<level>_SMALL(Wert, Toleranz)

ist erfolgreich, wenn der Betrag des Werts `>` als der Betrag der Toleranz ist.

BOOST_<level>_EQUAL_COLLECTIONS

vergleicht zwei Container (Collections) auf gleiche Inhalte, ähnlich wie der Algorithmus `mismatch` von Seite 692. Im Unterschied zu Letzterem muss das Ende des zweiten Bereichs angegeben werden. Unterschiedliche Elemente werden gemeldet. Dabei können Container der Standardbibliothek und einfache Arrays verwendet werden, sogar gemischt, wie das Beispiel zeigt:

```
BOOST_AUTO_TEST_CASE( eq_collection )
{
    std::vector<int> v;
    for(int i=1; i <= 8; ++i)
        v.push_back(i);
    int arr [] = { 1, 2, 3, 5, 5, 6, 7, 9};
    BOOST_CHECK_EQUAL_COLLECTIONS(v.begin(), v.end(), arr, arr+8 );
}
```

BOOST_<level>_BITWISE_EQUAL(a, b)

prüft, ob alle Bits der zwei Werte übereinstimmen. Falls nicht, schlägt der Test fehl, und es werden alle nicht übereinstimmenden Positionen ausgegeben.

18.3.5 Kommandozeilen-Optionen

Die Steuerung des Log-Levels mit einer Kommandozeilen-Option wird oben auf Seite 535 beschrieben. Daneben gibt es weitere Optionen, von denen die wichtigsten in der Tabelle 18.3 genannt werden. Alle Optionen sind auch durch entsprechende Umgebungsvariablen einstellbar. Die Spalte zwei zeigt die möglichen Werte. Der voreingestellte Wert ist jeweils zuerst genannt.

Tabelle 18.3: Kommandozeilen-Optionen

Option	Werte	Bedeutung
auto_start_dbg	no, yes	Bei Systemfehler Debugger starten
build_info	no, yes	Anzeige der Compiler-Version
catch_system_errors	yes, no	no: Systemfehler werden nicht aufgefangen. Sie können damit von einem übergeordneten Programm (GUI) analysiert werden.
detect_memory_leak	1, 0, > 1	nur für MS-Compiler (Details siehe [Roz])
detect_fp_exceptions	no, yes	Floating Point-Exceptions fangen (falls vom System unterstützt)
log_format	HRF XML	HRF = human readable format (ASCII-Text) für weitere automatisierte Verarbeitung
report_format	dasselbe wie log_format	
output_format	dasselbe wie log_format, hat aber ggf. Vorrang vor den beiden anderen	
random	0 1 z > 1	Tests der Reihe nach ausführen zufällige Reihenfolge, basierend auf der aktuellen Zeit zufällige Reihenfolge mit z als Initialisierungswert (random seed)
result_code	yes, no	no: es wird stets 0 zurückgegeben (bei Einbindung in GUI von Interesse)
run_test	durchzuführende Testfälle und -suiten. Wildcards sind erlaubt. Beispiele: testprog --runtest=test_a testprog --runtest=test_suite, test_c, xtest*	
show_progress	no, yes	Anzeige eines Fortschrittsbalkens
log_level	siehe Seite 535	

Die entsprechende Umgebungsvariable ergibt sich aus der Option, indem die Option in Großschreibung an die Zeichenkette BOOST_TEST_ gehängt wird. So gehört zu der Option show_progress die Umgebungsvariable BOOST_TEST_SHOW_PROGRESS. Die einzige Ausnahme ist nach [Roz] die Umgebungsvariable BOOST_TESTS_T0_RUN zur Option run_test.

19

Werkzeuge zur Verwaltung von Projekten

Dieses Kapitel behandelt die folgenden Themen:

- Dokumentation und Strukturanalyse von Programmen
- Versionsverwaltung
- Projektverwaltung, Kommunikation mit Wiki

Es gibt viele Werkzeuge, um das Management von Projekten zu erleichtern. Dieses Kapitel bietet eine subjektive Auswahl von Open Source-Tools, die sich an vielen Stellen der Welt bewährt haben und deren Einsatz in kleinen und mittleren Projekten sinnvoll ist.

19.1 Dokumentation und Strukturanalyse mit doxygen

Ein Programm sollte gut dokumentiert sein, damit nicht nur der Autor, sondern auch andere, die mit ihm zusammenarbeiten oder irgendwann an seine Stelle treten, das Programm gut verstehen. Es ist es sehr mühevoll, eine separate Programmdokumentation

zu erstellen und zu pflegen. Insbesondere wird häufig vergessen, Änderungen des Programms in der Dokumentation nachzuziehen. Aus diesem Grund hat es sich eingebürgert, die Dokumentation direkt im Programm selbst vorzunehmen (single-source-Prinzip) und mit einem Werkzeug zu extrahieren. Die Dokumentation wird mitsamt den steuernden Anweisungen innerhalb von Kommentaren der jeweiligen Programmiersprache untergebracht, damit der Compiler sie ignoriert.

Doxygen (<http://www.doxygen.org>) ist ein sehr bekanntes und für diesen Zweck hervorragend geeignetes Open Source-Werkzeug, das für die Sprachen C++, C, Java, PHP und mehr ausgelegt ist. Doxygen führt eine statische Analyse der Quelldateien durch, sucht dabei nach Dokumentations-Schlüsselwörtern und erzeugt eine gut lesbare Ausgabe, die als Programmdokumentation geeignet ist. Wenn auf Ihrem System außer Doxygen auch Graphviz, ein Tool zur Visualisierung von Graphen (<http://www.graphviz.org>), installiert ist, erzeugt Doxygen auch Klassendiagramme und Graphen, die die gegenseitigen Abhängigkeiten zeigen. Ein *Aufrufgraph* für eine Funktion zeigt, welche anderen Funktionen von ihr aufgerufen werden. Ein *Aufrufergraph* zeigt hingegen, welche anderen Funktionen diese Funktion aufrufen.

Sowohl Doxygen als auch Graphviz sind für die gängigen Betriebssysteme erhältlich. In vielen Linux-Distributionen sind sie bereits enthalten, und für Windows gibt es leicht installierbare Binär-Dateien. Doxygen erzeugt auf Knopfdruck aus den Quelltexten eine aktuelle Dokumentation für ein ganzes Projekt, wahlweise im RTF-, HTML-, XML- oder TeX-Format.

Für die Dokumentation von Schnittstellen gibt es viele Möglichkeiten, die sich vom Konzept aber nicht unterscheiden, sodass ich mich hier auf eine beschränke:

- Die Dokumentation von Schnittstellen geschieht durch Voranstellen eines (geeigneten) Kommentars.
- Doxygen arbeitet ähnlich wie Javadoc (und umgekehrt). Es sei JAVADOC AUTOBRIEF=YES gesetzt. Damit wird die erste mit einem Punkt endende Zeile als Kurzbeschreibung aufgefasst.
- Ich verwende einen Stil, der auch für Java verwendet wird.
- Als Ausgabeformat wähle ich HTML.

Weitere Möglichkeiten bitte ich Sie, dem Doxygen-Manual zu entnehmen. Das Beispiel einer Musterlösung der Übungsaufgabe vier am Ende des Kapitels 4 zeigt, wie einfach die Kommentierung ist und was sich mithilfe von Doxygen daraus ergibt. Der Kommentar befindet sich *direkt vor* der jeweiligen Methode oder Klasse und beginnt mit */***, also zwei Sternchen. Die Schlüsselworte *@param* und *@returns* sind selbsterklärend. Es gibt einige andere, wie etwa *@author* und *@date*.

Listing 19.1: Methoden der Klasse Taschenrechner

```
// Auszug aus cppbuch/loesungen/k4/6/taschenrechner.h
/** gibt das Ergebnis der Auswertung eines Summanden zurueck.
 * @param c erstes auszuwertendes Zeichen
 * @returns Ergebnis
 */
long summand(char& c);

/** gibt das Ergebnis der Auswertung eines Faktors zurueck.
```

```

* @param c erstes auszuwertendes Zeichen
* @returns Ergebnis
*/
long faktor(char& c);

```

Die Abbildung 19.1 gibt einen Überblick über die Klasse, während die Abbildung 19.2 Einzelheiten einschließlich eines Aufrufgraphen zeigt.

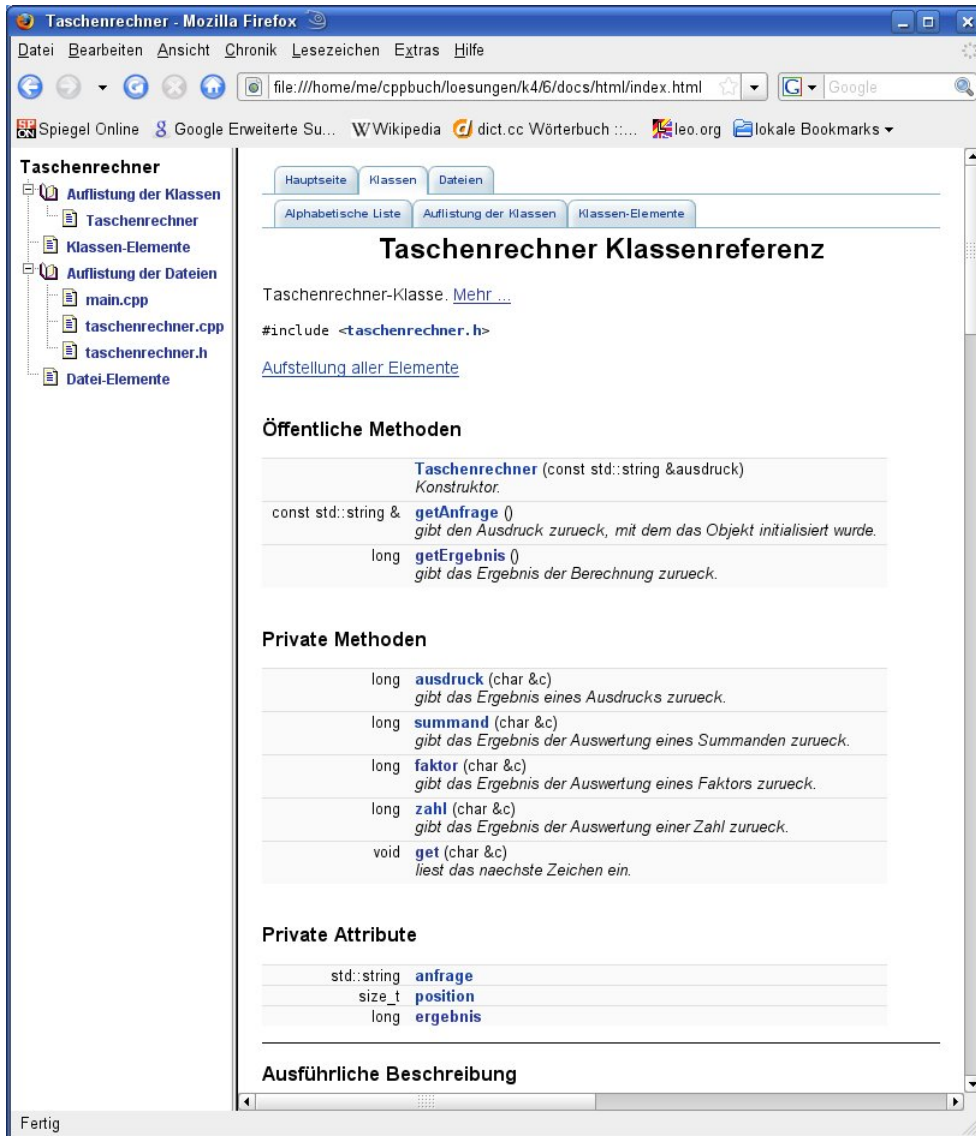


Abbildung 19.1: Auszug der HTML-Ausgabe von doxygen: Klassenüberblick

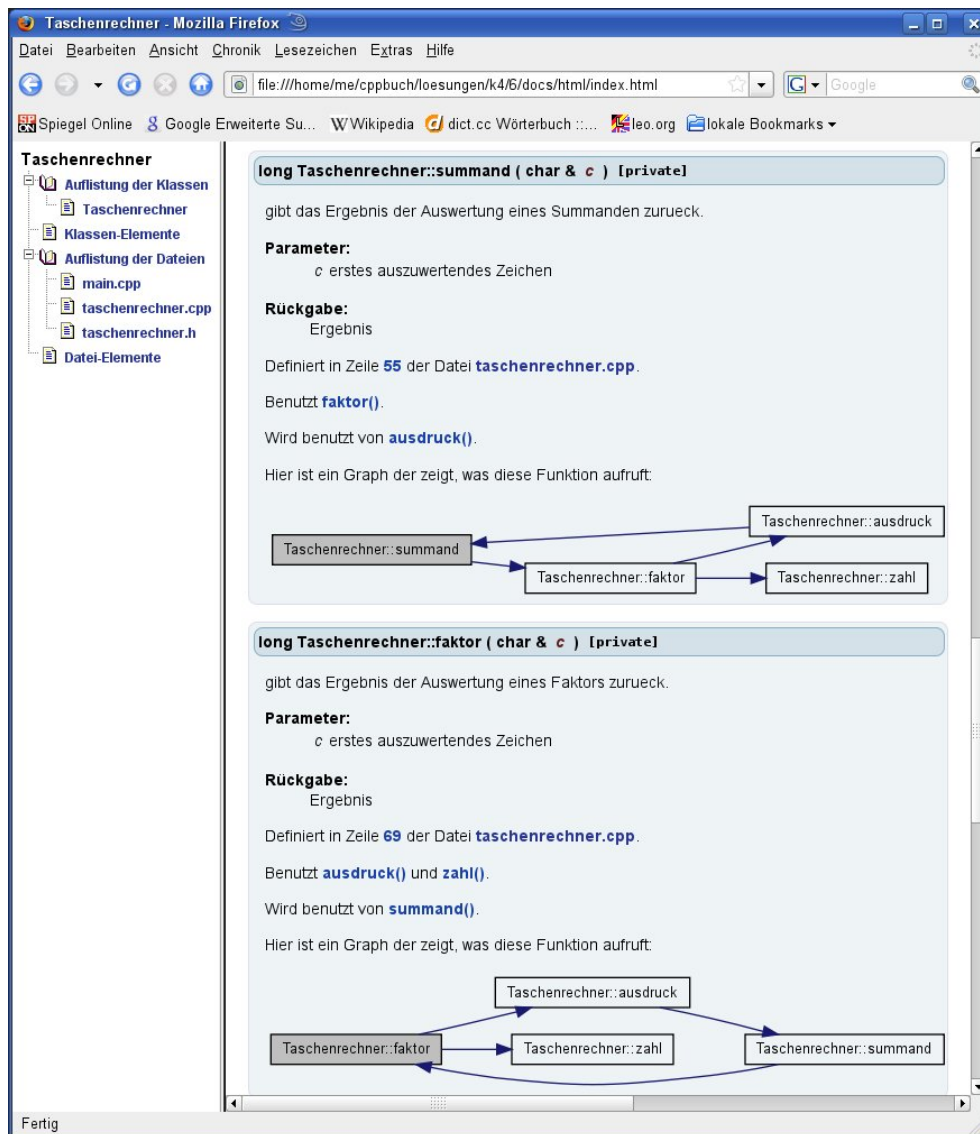


Abbildung 19.2: Auszug der HTML-Ausgabe von doxygen: Detail

Die Ausgabe wird erzeugt, indem im Verzeichnis *cppbuch/loesungen/k4/6/* das Kommando *doxygen* eingegeben wird. Doxygen wertet eine Konfigurationsdatei aus, die normalerweise vorher mit *doxywizard* oder mit *doxygen -g Doxyfile* erzeugt werden muss (im genannten Verzeichnis ist sie vorhanden). Diese Datei kann mit dem Wizard oder mit einem Editor bearbeitet werden. Der große Vorteil ist die Verlinkung aller Elemente, sodass man sehr leicht navigieren kann; auch der Quellcode wird mit einem Klick angezeigt. Die in der Header-Datei vorhandenen Kommentare brauchen in der Implementationsdatei nicht wiederholt zu werden. Dort können sich aber */** ... */*-Kommentare befinden, deren Inhalt zusätzlich angezeigt wird. Die Erzeugung von Graphen wird in der Konfigu-

rationsdatei (voreingestellter Name Doxyfile) durch den Schlüssel HAVE_DOT gesteuert, der mit dem Wert YES versehen werden muss. Der Name des Schlüssels rührt von der Komponente dot des Tools Graphviz her. Die Tabelle 19.1 enthält die wichtigsten Einstellungen der Konfigurationsdatei für eine Code-Analyse. Der Wert ist nach Wunsch jeweils auf YES oder NO zu setzen.

Tabelle 19.1: Die wichtigsten Einstellungen in der Konfigurationsdatei

Schlüssel	Bedeutung
CALL_GRAPH	Aufrufgraph erzeugen.
CALLER_GRAPH	Aufrufergraph erzeugen.
CLASS_DIAGRAMS	Klassen- und Vererbungshierarchiediagramme erzeugen.
EXTRACT_ALL	Alle vorhandenen Informationen extrahieren.
GENERATE_HTML	HTML-Dateien erzeugen.
GENERATE_LATEX	Latex-Dateien erzeugen.
HAVE_DOT	Grafiken erzeugen (setzt Graphviz voraus).
SOURCE_BROWSER	Vollständige Verlinkung erzeugen.

19.1.1 Strukturanalyse

Doxygen analysiert Programmcode und erzeugt eine Ausgabe, auch wenn keinerlei Dokumentationsanweisungen enthalten sind. Damit ist Doxygen ein hervorragendes Tool zur Analyse von Programmen, auch wenn sie schlecht oder nicht dokumentiert sind.

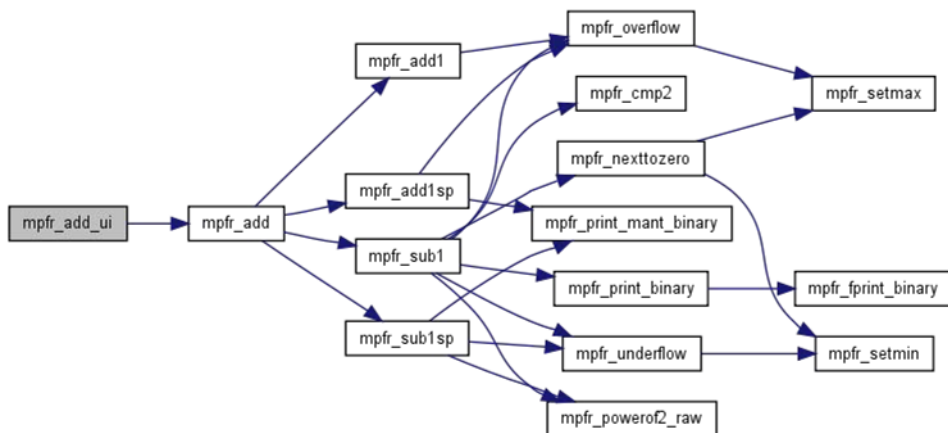


Abbildung 19.3: Aufrufgraph der MPFR-Library für die Funktion `mpfr_add_ui`

Wer sich jemals in ein unbekanntes Programm mit vielen Dateien einarbeiten musste, wird einen Aufruf- und einen Aufrufergraphen schätzen. Das obige Programmbeispiel ist zu einfach, daher zeigen die Abbildungen 19.3 und 19.4 als Beispiel die mit Doxygen erzeugten Graphen für die Funktion `mpfr_add_ui` der MPFR¹-Bibliothek, die vom G++-Compiler benötigt wird.

¹ Multiple-Precision Floating-point computations with correct Rounding

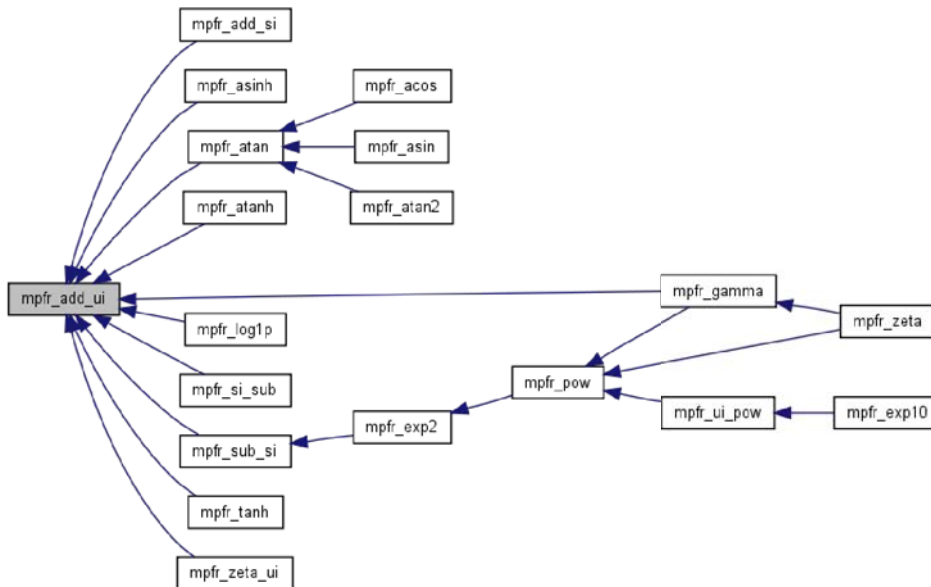


Abbildung 19.4: Aufrufergraph der MPFR-Library für die Funktion `mpfr_add_ui`

19.2 Versionskontrolle

Ebenso wichtig wie eine gute Dokumentation ist eine Versionskontrolle. Sie erlaubt es, einen vergangenen Stand einer Datei zu rekonstruieren oder kundenspezifische Varianten des Programmcodes zu verwalten. CVS (Concurrent Version System) war früher das Open Source-Werkzeug der Wahl. Vor einigen Jahren wurde CVS nach und nach durch Subversion abgelöst, ebenfalls Open Source-Software [svn]. Subversion verwaltet alle Dateien in einem sogenannten Repository, welches datenbank- oder dateisystembasiert sein kann. Dabei geht es nicht nur um Programmcode, sondern um alle zum Projekt gehörenden Dateien einschließlich Bilddateien. Wenn geänderte Dateien in das Repository eingespeist werden, erhöht sich die Änderungsnummer, Revision genannt, um 1. Einer Revision ist ein Datum und eine Uhrzeit zugeordnet. Subversion erlaubt es, die Unterschiede zwischen Revisionsständen festzustellen.

Typischerweise wird ein ganzes Projekt in Form eines Verzeichnisbaums im Repository abgelegt. Beim ersten Mal wird dieser Baum vollständig kopiert. Bei späteren Änderungen jedoch werden nur noch die *Unterschiede* nach einem gut durchdachten Schema abgelegt. Jeder Revision ist eine geschickt verlinkte Baumstruktur zugeordnet. Dadurch bedingt wird viel Speicherplatz gespart.

Die Abbildung 19.5 zeigt die Subversion-Architektur. Als Repository ist sowohl eine Datenbank als auch ein Filesystem möglich. Es gibt mehrere Wege der Kommunikation des Clienten mit dem Subversion-Server, gekennzeichnet durch das Protokoll:

- `file://` Direkter Zugriff auf ein Repository, das sich auf einer lokalen Festplatte befindet. Zum Ausprobieren ist dieser Weg gut geeignet. Er hat aber den Nachteil, dass

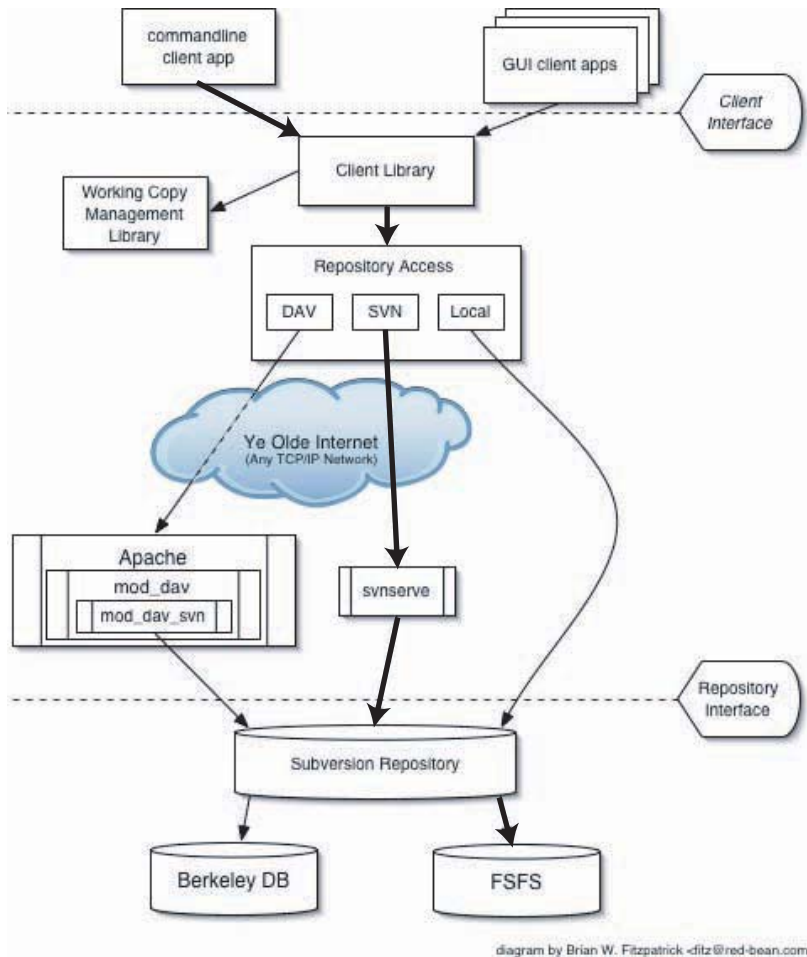


Abbildung 19.5: Subversion-Architektur (aus [SFP], einige Pfeile verstärkt)

es keine Datensicherheit bei einem Ausfall der Festplatte gibt. Wenn man eine andere Festplatte wählt, besteht die Möglichkeit des Ausfalls des Festplatten-Controllers. Immerhin kann man den ersetzen oder die Platte in einen anderen Rechner einbauen, um wieder an die Daten zu kommen.

- `http://` Zugriff über das WebDAV-Protokoll, das vom Apache-Server unterstützt wird. Der Apache-Server muss natürlich eingerichtet sein.
- `https://` Wie `http://`, aber mit SSL-Verschlüsselung (Secure Sockets Layer).
- `svn://` Subversion stellt einen eigenen Server zur Verfügung. Damit kann die Installation des Apache-Servers entfallen, falls er nicht für andere Zwecke gebraucht wird. `svn://` ist ein Subversion-eigenes Protokoll. Wie bei `http://` gehen alle Informationen im Klartext über die Leitung, auch Passwörter. `svn://` läuft über Port 3690.
- `svn+ssh://` Wie `svn://`, aber über SSH (Secure Shell). SSH ist ein Protokoll, das nicht nur die Verbindung verschlüsselt, sondern auch eine Authentifizierung verlangt. SSH

ist gleichzeitig auch der Name eines Programms, mit dem man sich über eine sichere Verbindung auf einem anderen Rechner einloggen kann. Das Kommando ist `ssh -X -l username hostname`. Der Parameter `-X` ist optional. Er sorgt dafür, dass man Anwendungen (Editor ...) des entfernten Rechners aufrufen kann, die das X-Window System nutzen. SSH benutzt den Port 22.

Wenn aus Datensicherheitsgründen ein entfernter Rechner mit einer verschlüsselten Verbindung und gleichzeitig eine möglichst einfache Konfiguration gewählt werden soll, bietet sich für kleine Teams die zuletzt genannte Möglichkeit an, wobei das Filesystem des Servers verwendet wird. Dem entspricht der dick in Abbildung 19.5 eingezeichnete Weg. Er wird im Folgenden exemplarisch dargestellt. Etwas ansprechender in der Benutzung als die Kommandozeile sind die verschiedenen für Subversion erhältlichen GUI-Clients wie TortoiseSVN oder RapidSVN. Für manche Entwicklungsumgebungen gibt es Plug-ins. SVN ist die Kurzform von Subversion.



Hinweis

Aus Platzgründen wird in diesem Abschnitt nur kurz auf die wichtigsten Möglichkeiten von Subversion eingegangen. Mehr dazu können Sie im SVN-Buch [SFP] lesen, das sich auf der zu diesem Buch gehörenden DVD befindet. Auf der Internetseite [svn] gibt es Links zu weiteren hilfreichen Informationen.

Arbeitsweise aus Benutzersicht

Normalerweise sind an einem Projekt mehrere Personen beteiligt, denen bestimmte Bereiche des Projekts zugeordnet sind. Am Ende des Projekts könnte die Integration der Bereiche zum Beispiel ein neues Produkt ergeben. Nachdem das Projekt der Versionsverwaltung unterliegt, kann sich jeder Softwareentwickler die aktuelle Kopie herunterladen, die damit seine Arbeitskopie wird. Dieser Vorgang wird `checkout` genannt. Nach der Arbeit an einem Modul, einschließlich Integration und Test (sonst gibt es Schelte von den Kollegen ...), wird es mit `commit` in das Repository eingespeist. Jedem anderen steht dann das geänderte Modul nach einem `update` zur Verfügung.

19.2.1 Einrichtung des Servers

In diesem Abschnitt beschreibe ich die Einrichtung auf einem Linux-System, weil Linux-Rechner als Server verbreitet sind. Die Einrichtung als Windows Server bitte ich Sie, der Literatur zu entnehmen, zum Beispiel [SFP], Kapitel 6. Zunächst einmal müssen SSH und Subversion installiert sein. Dies ist bei den meisten Linux-Distributionen bereits der Fall oder mithilfe des Systemverwaltungstools leicht nachzuholen. Eine weitere Möglichkeit ist das Herunterladen der Quellen. Nach Entpacken wird Subversion wie üblich mit `./configure` und `make` übersetzt und mit `make install` installiert. Bitte achten Sie darauf, dass die auf dem Server installierte Version von Subversion mit der auf dem Clienten wenigstens in der Ziffer nach dem Punkt übereinstimmt, d.h. nicht auf dem einen Rechner Version 1.4 und auf dem anderen 1.5 verwenden.

Bevor Sie an die Einrichtung des Servers gehen, probieren Sie bitte aus, ob der Login auf dem Server mit `ssh` funktioniert. Dies kann direkt am Server geschehen, wenn als Hostname `localhost` oder `127.0.0.1` eingegeben wird. Falls die Verbindung nicht zustan-

de kommt, liegt es vermutlich daran, dass die Firewall eingeschaltet ist und den entsprechenden Port (siehe oben) sperrt. Gegebenenfalls muss der Port freigeschaltet werden. Bei Computern, die über einen Router mit dem Internet verbunden sind, kann die Firewall ganz abgeschaltet werden, wenn der Router die Firewall-Aufgabe übernimmt.

Wegen der von SSH verlangten Authentifizierung sollten auf dem Server spezielle Benutzerkonten nur für den SVN-Zugriff angelegt werden. Der Grund: Weil diese User Lese- und Schreibrechte im SVN-Bereich haben, können sie Informationen anderer SVN-User lesen. Danach kann der Server mit den folgenden Schritten eingerichtet werden:

- Als Root-User das Wurzelverzeichnis der Repositories anlegen, z. B. `/var/svn/repos`. Für das zu verwaltende Projekt mit dem Namen `projekt` (es kann hier ein anderer Name gewählt werden) wird das Repository mit

```
svnadmin create /var/svn/repos/projekt
```

angelegt. Svnadmin erzeugt dabei verschiedene Unterverzeichnisse.

- In `/var/svn/repos/projekt/conf` wird die Datei `passwd` modifiziert, sodass die Benutzer mit den Passwörtern unter der Gruppe `[users]` aufgeführt werden. Die Syntax ist `username = password`. Wegen der späteren Authentifizierung ist es am einfachsten, wenn hier das Login-Passwort verwendet wird. Dies ist nicht problematisch, wenn diese User nur für SVN verwendet werden. Inhalt zum Beispiel:

```
[users]
hansi = hansipasswort
leila = leilapasswort
# usw.
```

- In die Datei `svnserve.conf` im selben Verzeichnis wird unter `[general]` eingetragen:

```
# Anonyme duerfen nichts
anon-access = none
# Autorisierte duerfen lesen und schreiben
auth-access = write
# Verweis auf die Passwortdatei:
password-db = passwd
# realm ist ein Namensbereich fuer die Authentifizierung.
realm = Projekt Wunderbar
```

- Die Ausführung der SVN-Programme sollte durch einen speziellen User `svn` nur für diesen Zweck erfolgen, um andere Bereiche zu schützen. Ebenfalls wird eine Gruppe `svn` angelegt:

```
groupadd svn
useradd -G svn svn
```

- Den Usern auf dem System, die SVN benutzen, diese Gruppe zusätzlich zuordnen. Mit `usermod -A svn username` wird `svn` dem User hinzugefügt. Falls die Option `-A` nicht zur Verfügung steht (man `usermod` und `usermod -help` geben bei meinem System nicht dieselben Auskünfte), hilft der folgende Umweg: `groups username` gibt alle Gruppen aus. Mit `usermod -G svn, ... username` werden `svn` und alle anderen Gruppen dem Benutzerkonto `username` hinzugefügt. Die Ellipse `...` ist dabei durch die mit `groups username` ermittelten Gruppen zu ersetzen, falls außer `svn` weitere Gruppen gewünscht sind, für den Fall, dass es sich doch nicht nur um reine SVN-User handeln sollte. Die Gruppen sind ohne Leerzeichen nach dem Komma aufzuführen.

- Dann werden von Root die Rechte gesetzt, wobei die letzte Zeile den Zugriff von anderen verhindert (o = others).

```
cd /var
chown -R svn.svn svn/
chmod -R ug+rw svn/
chmod -R o-rwx svn/
```

- Den Netzwerkdienst mit einem Hilfsprogramm des Betriebssystems (yast2 bei SuSE-Linux) einstellen, damit bei einer SVN-Anfrage der Dienst `svnserve` gestartet wird. In der Datei `/etc/xinetd.d/svnserve` (SuSE-Linux) könnte danach Folgendes stehen (für Ihr System anpassen):

```
service svnserve
{
    socket_type    = stream
    protocol      = tcp
    wait          = no
    user          = svn
    group         = svn
    server        = /usr/bin/svnserve
    server_args   = svnserve -i -r /var/svn
}
```

`/usr/bin/svnserve` ist das aufzurufende Programm, der Ort könnte je nach Installation auch zum Beispiel `/usr/local/bin/svnserve` sein. Der Parameter `-i` steht für die Xinetd-Einbindung, der Wert nach dem Parameter `-r` gibt das Wurzelverzeichnis für SVN an.

- Um in jedem Fall Probleme mit Zugriffsrechten zu vermeiden, empfehlen die Autoren von [SFP], die `svn`-Programme umzubenennen und über ein Skript aufzurufen. Beispiel: `svnserve` wird in `svnserve-real` umbenannt. Dann wird ein Skript `svnserve` mit folgendem Inhalt angelegt:

```
#!/bin/sh
umask 002
/usr/bin/svnserve-real "$@"
```

Dieses Skript wird mit `chmod ug+x` ausführbar gemacht und tritt damit an die Stelle des ursprünglichen Programms. Der Sinn ist, mit `umask 002` (= user mask) die Schreibrechte anderer grundsätzlich zu verbieten, egal welche Rechte der ausführende User hat. `umask` verwendet dieselben Zahlenkombinationen wie der Befehl `chmod`, nur dass `umask` die angegebene Berechtigung ausblendet. Wer keine Lust hat, mit Oktalzahlen zu rechnen, kann sich des Links <http://javascriptkit.com/script/script2/chmodcal.shtml> bedienen.

- Zum Schluss kann mit `svn list svn+ssh://127.0.0.1/var/svn/repos/projekt` auf dem Server geprüft werden, ob `svn+ssh` funktioniert. Nach Passwort-Eingabe muss das Kommando ohne Fehlermeldung zurückkommen. Eine Ausgabe gibt es nicht – das Verzeichnis ist ja noch leer.

19.2.2 Exemplarische Benutzung

Vor dem ersten Import des Projektes sollten alle Dateien, die nicht der Versionsverwaltung unterliegen sollen, entfernt werden. Das Verzeichnis `projekt/trunk` enthalte den

Hauptentwicklungstamm (zu den Verzeichnissen *branches* und *tags* siehe [SFP]). Der erste Import geschieht mit:

```
svn import lokalesVerzeichnis svn+ssh://username@hostname/var/svn/repos/projekt \
-m "erster Import"
```

lokalesVerzeichnis ist das zu übertragende Projektverzeichnis, statt *username* muss der Name eines auf dem Server eingerichteten Benutzers verwendet werden, und *hostname* ist der Rechnername, der auch eine IP-Adresse sein kann, etwa 192.168.1.20 in einem Intranet. Der mit dem Parameter *-m* übergebene Kommentar ist zwingend erforderlich; wird er vergessen, wird automatisch der Standardeditor des Betriebssystems zwecks Eingabe gestartet. Natürlich wird auch das Passwort verlangt, bei manchen SVN-Kommandos auch zweimal (Login- und SVN-Passwort). Der Einfachheit halber werden Passworteingaben im Folgenden weggelassen.

```
svn -v list svn+ssh://username@hostname/var/svn/repos/projekt
```

zeigt die unter *projekt* existierenden Verzeichnisse an, zum Beispiel *trunk*. Die Option *-v* sorgt für eine ausführliche Ausgabe mit Revision, User und Datum, was bei späteren Aufrufen interessant sein kann. Das Versionsverwaltungssystem ist jetzt fertig eingerichtet.

Aus- und Einchecken

Jedes berechnete Projektmitglied kann sich jetzt seine aktuelle Arbeitskopie mit

```
svn co svn+ssh://username@hostname/var/svn/repos/projekt projekt
```

herunterladen. *co* ist die Abkürzung für *checkout*. Das Arbeitsverzeichnis wird am Ende der Zeile angegeben (*projekt*); es kann auch anders heißen. In jedem Verzeichnis der Arbeitskopie befinden sich Verzeichnisse mit dem Namen *.svn*. Sie dienen der SVN-Verwaltung und dürfen nicht geändert oder gelöscht werden. Nun kann im Verzeichnis *trunk* gearbeitet werden. Nach Abschluss der Arbeiten wird der neue Stand mit

```
svn commit projekt -m "erstes Commit"
```

gesichert. Es wird hier dabei ausgegangen, dass das zu sichernde Arbeitsverzeichnis *projekt* heißt, entsprechend dem *checkout* von oben. Dabei werden nur die geänderten Daten übertragen. Die Begründung für die Änderung muss wie oben nach *-m* angegeben werden. Wenn nun ein anderes Projektmitglied, das bereits einmal das Projekt ausgecheckt hat, *svn up projekt* aufruft, wird der aktualisierte Stand auf seinen Rechner übertragen. Das Wort *up* steht für *update*.

Wenn versehentlich etwas gelöscht wurde (oder aus anderen Gründen), kann es helfen, einen alten Stand des Projektes zu rekonstruieren.

```
svn co -r 44 svn+ssh://username@hostname/var/svn/repos/projekt rev44
```

legt zum Beispiel eine vollständige Kopie der Revision 44 im Verzeichnis *rev44* an. Es können auch einzelne Dateien rekonstruiert oder anstelle der Revision kann ein Datum gewählt werden ([SFP]).

Dateien hinzufügen und entfernen

Neue Dateien werden nicht automatisch der Versionsverwaltung hinzugefügt, um nicht temporäre oder andere weniger wichtige Dateien zu speichern. Neue Dateien müssen

explizit mit `add` angemeldet und zu löschende Dateien mit `rm` (= remove) abgemeldet werden:

```
svn add projekt/trunk/neuedatei
svn rm projekt/trunk/alteidei
```

Beide Aktionen wirken ab dem nächsten `commit`.

Schlüsselwortersetzung

Es ist sinnvoll, in einer Datei selbst den aktuellen Stand zu dokumentieren. Dazu bietet SVN einige Schlüsselwörter an: `Date` liefert Datum und Uhrzeit, `Author` den User-Namen, `HeadURL` die Adresse des Dokuments, und `Rev` ist die Revision. `Id` ist eine Kurzfassung von allem. Dazu werden die Schlüsselwörter, durch Dollarzeichen begrenzt, in den Kommentarbereich einer Datei eingetragen. Beispiel einer C++-Datei *main.cpp*:

```
// $Id$
// $Rev$
// $Date$
// $Author$
// $HeadURL$
// ...
{
    // Programmcode
}
```

Der Wunsch nach einer Schlüsselwortersetzung in allen `.cpp`-Dateien wird dem SVN-System einmalig mit

```
svn propset svn:keywords "Date Rev Author Id HeadURL" projekt/trunk/*.cpp
```

mitgeteilt. Die Eintragungen werden beim nächsten `commit` wirksam und beim nächsten `update` expandiert. Danach sieht der obige Programmauszug etwa so aus:

```
// $Id: main.cpp 6 2008-11-04 18:07:48Z username $
// $Rev: 6 $
// $Date: 2008-11-04 19:07:48 +0100 (Di, 04. Nov 2008) $
// $Author: username $
// $HeadURL: svn+ssh://username@hostname/var/svn/repos/projekt/trunk/main.cpp$
// ...
{
    // Programmcode
}
```

Die Zahl 6 nach `main.cpp` im `$Id$`-Bereich ist die Revision. Die Zusammenstellung zeigt nur die verschiedenen Möglichkeiten, in der Praxis wird man entweder `$Id$` oder die anderen Schlüsselwörter nehmen. Damit sind die wichtigsten Möglichkeiten von SVN beschrieben. Es gibt aber noch sehr viele weitere, die hier nicht behandelt werden können und die anderweitig gut dokumentiert sind.



Mehr zur Versionsverwaltung mit Subversion lesen Sie im SVN-Buch auf der DVD.

19.3 Projektverwaltung

Außer Versionen zu verwalten gibt es in einem Projekt noch andere Aufgaben: Organisation der Kommunikation im Projekt, Termine verfolgen, Kosten kontrollieren, Personaleinsatz planen. In diesem Abschnitt stelle ich Open Source-Werkzeuge vor, die dabei helfen können.

19.3.1 Projektmanagement

Seit mehr als fünf Jahren gibt es eine leistungsfähige Open Source-Konkurrenz zu den kommerziellen Werkzeugen für die Dokumentation und Steuerung von Zeitplänen und Finanzen eines Projekts: Open-Project (<http://openproj.org/openproj/>) und Project-Open (<http://www.project-open.com/>). Beide Software-Pakete laufen auf Windows-, Linux- und Mac-Betriebssystemen. Sie bieten die Möglichkeit, Arbeitspakete zu bilden und zu strukturieren. Die Arbeitspakete und ihre Abhängigkeiten werden terminlich eingeplant. Die Terminlage wird mit Balkendiagrammen (Gantt²-Diagrammen) visualisiert. Project-Open hat auf der Homepage allerdings einiges mehr an Dokumentation zu bieten und hat nach eigenen Angaben mehr als 1000 Firmen als Nutzer. Wie bei anderen Open Source-Projekten wird nicht an der Software, sondern der begleitenden Beratung verdient.

19.3.2 Wiki für Software-Entwicklungsprojekte

Ein Wiki ist eine Online-Plattform zur Kommunikation zwischen Menschen mit ähnlich gelagerten Interessen. Ein Wiki zeichnet sich dadurch aus, dass es als Hypertextsystem aufgebaut ist, in dem seine Benutzer nicht nur lesen, sondern auch schreiben können. Zur Textformatierung wird eine Sprache benutzt, die einfacher als HTML ist. In einem Wiki können Dokumente abgelegt und von mehreren Autoren bearbeitet werden. Es sind Diskussionsforen und vieles andere möglich. Das bekannteste Wiki ist vermutlich Wikipedia, eine beliebte Auskunftsource mit zig-tausenden Autoren. Ein Wiki ist eine ideale Plattform zur Kommunikation innerhalb eines Projekts aus folgenden Gründen:

- Die Mitarbeiter eines Projekts müssen sich nicht in räumlicher Nähe befinden, sodass eine weltweit verteilte Entwicklung beinahe so einfach ist, als würden alle am selben Ort arbeiten.
- Der Zugriff ist asynchron, das heißt, dass jeder dann über das Wiki kommuniziert, wenn es der Arbeitsablauf am besten erlaubt. Das soll nicht bedeuten, dass persönliche Treffen überflüssig sind – sie sind nur nicht so oft erforderlich.
- Ein Wiki ist ein Content Management System, in dem Dokumente archiviert werden. Gleichzeitig hat es eine eingebaute Versionsverwaltung für die im Wiki erzeugten Inhalte, sodass Vorgängerversionen eines Artikels gesehen und rekonstruiert werden können.
- Manche Wikis erlauben die leichte Integration einer Versionsverwaltung für Software auf eine Weise, dass das Repository mit dem Browser eingesehen werden kann.

Wikis sind gerade in der Open Source-Entwicklung sehr beliebt. Eine beeindruckende Zahl von Wikis der Apache Community ist unter <http://wiki.apache.org/general/FrontPage>

² Henry Gantt hatte diese Art der Darstellung vor etwa 100 Jahren bekannt gemacht.

aufgeführt. Wenn Sie ein Wiki in einem Projekt einsetzen, empfiehlt sich die MoinMoin Wiki Engine (<http://moinmo.in/>), mit dem die erwähnten Apache Wikis und viele andere realisiert wurden.

Eine ebenfalls empfehlenswerte Alternative ist Trac (<http://trac.edgewall.org/>). Trac ist bereits für Softwareprojekte zugeschnitten. Es stellt eine Schnittstelle für Subversion zur Verfügung, wobei sich das Repository allerdings auf derselben Maschine wie Trac befinden muss. Trac erlaubt mithilfe von sogenannten Tickets die Verfolgung von Aufgaben und Terminen. Die Aufgabenverfolgung mit Tickets erlaubt auch eine Fehlerverfolgung, sodass dafür spezialisierte Werkzeuge wie Mantis (<http://www.mantisbt.org/>) nicht notwendig sind. Trac wird von einer großen Anzahl Firmen eingesetzt. Die Abbildung 19.6 zeigt die Trac-Wiki-Titelseite eines studentischen Projekts, in dem auch die genannten Werkzeuge Doxygen und Subversion zum Tragen kamen.

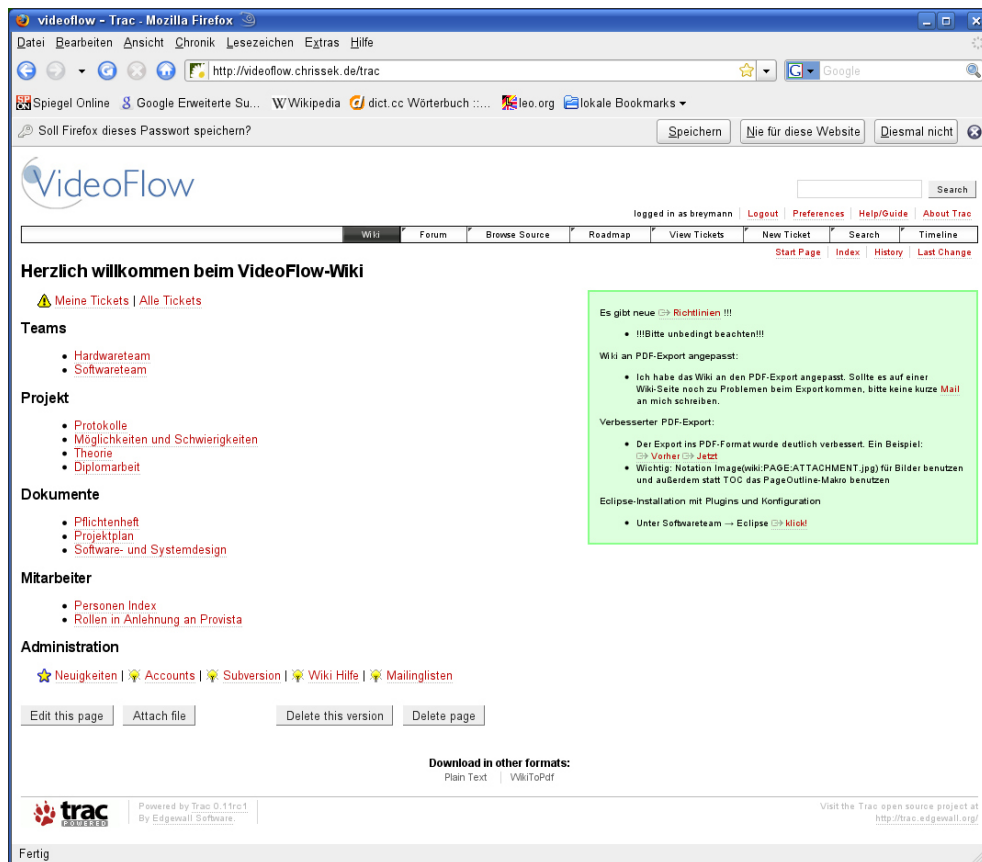


Abbildung 19.6: Trac-Hauptseite eines Projekts

IV.

Teil IV:

**Das C++-Rezeptbuch:
Tipps und Lösungen
für typische Aufgaben**

20

Sichere Programm- entwicklung

Dieses Kapitel behandelt die folgenden Themen:

- Regeln zum Design von Methoden
- Defensive Programmierung
- Exception-sichere Ressourcenbeschaffung
- Tipps zur Thread-Programmierung

Dieses Kapitel zeigt eine Auswahl von Tipps zu sicheren Programmentwicklung. Darunter sind einfache Tipps wie der zum korrekten Vergleich von `double`-Werten oder zur Schreibweise von `if`-Anweisungen, aber auch Faustregeln zur Konstruktion von Methoden und zur exception-sicheren Ressourcenbeschaffung.

20.1 Regeln zum Design von Methoden

Dieser Abschnitt gibt einige Empfehlungen zur Konstruktion der Prototypen von Methoden. Zur Erinnerung: Ein Prototyp ist die Deklaration einer Schnittstelle, über die von außen auf ein Objekt zugegriffen werden kann. Ein Prototyp besteht aus Rückgabe-

typ, Name und Parameterliste, wie schon aus Abschnitt 3.1.1 bekannt ist. Dazu kommt manchmal noch ein const-Qualifizierer. Unter anderem beziehen sich die Empfehlungen auf diese vier Bestandteile. Als Beispiel in einigen Fällen dient die Klasse `Geld`, deren Objekte aus einem Geldbetrag und der Währung als Attribute bestehen. Die Klasse soll in einer Anwendung Folgendes leisten:

```
// main()-Programm
Geld kaufpreis(100.00, "Euro");
cout << kaufpreis.betrag(); // 100
cout << kaufpreis.waehrung(); // Euro
// Ausgabe für Scheckvordrucke:
cout << kaufpreis.toString(); // eins-null-null
kaufpreis.neuerBetrag(90.0); // Betrag ändern
```

Aus der Anwendung ergeben sich Prototypen, die im `public`-Bereich der Klasse aufgeführt sind. Die Namen der privaten Attribute sind beliebig gewählt.

```
class Geld {
public:
    Geld(double einBetrag, const string& eineWaehrung);
    double betrag() const;
    void neuerBetrag(double);
    const string& waehrung() const;
    string toString() const;
private:
    double derBetrag;
    string dieWaehrung;
};
```

Die Methode `toString()` berechnet aus dem Betrag einen String entsprechend der Ziffernfolge des ganzzahligen Anteils des Betrags und gibt ihn zurück. Hier nun die sechs Regeln:

1. Die geplante Anwendung bestimmt die Methoden. Beim Entwurf einer Klasse ist es empfehlenswert, sich die Anwendung vorher zu überlegen und nur solche Prototypen vorzusehen, die dazu beitragen. Es ist nicht sinnvoll, Methodendeklarationen aufzuschreiben, die nur *vielleicht* brauchbar sind und deren Anwendung ungewiss ist. Der Grund dafür ist einfach: Alle Methoden müssen irgendwann auch definiert, dokumentiert und getestet werden. Diese Arbeit kann man sich für alle Methoden sparen, die nicht zum Einsatz kommen.
2. Der Name einer Methode soll beschreiben, was die Methode tut. Zum Beispiel ist der Name `betrag` aussagekräftiger als etwa `zahl`.
3. Eine Methode, die den Zustand eines Objekts nicht ändert, erhält den `const`-Qualifizierer (Methoden `betrag()`, `waehrung()` und `toString()`, nicht aber `neuerBetrag()`). Siehe auch die ausführliche Diskussion in Abschnitt 4.5.
4. Der Rückgabetypp ist natürlich `void`, wenn die Methode zwar etwas tut, aber nichts zurückgibt. Andernfalls bestimmt sich der Rückgabetypp aus der Antwort auf die Frage: Ist der zurückgegebene Wert ein Attribut der Klasse?
 - **Ja:** Nächste Frage: Ist der zurückgegebene Wert von einem Grunddatentyp? (d.h. `int`, `double`, `bool` usw. im Gegensatz zu einem Klassentyp)
 - *Ja:* Rückgabe per Wert. Beispiel: `betrag()`

- *Nein*: Rückgabe per `const&`. Beispiel: `waehrung()`
 - *Nein* (kein Attribut der Klasse): Rückgabe per Wert. Beispiel: `toString()`
5. Die Art der Übergabe eines Objekts in der Parameterliste kann ebenfalls durch die Beantwortung einiger Fragen bestimmt werden. Soll das Objekt beim Aufrufer der Methode verändert werden?
- *Ja*: Übergabe per Referenz (siehe Seite 111)
 - *Nein*: Nächste Frage: Gehört der übergebene Wert zu einem Grunddatentyp?
 - *Ja*: Übergabe per Wert. Beispiel: `neuerBetrag(double)`
 - *Nein*: Wird in der Funktion eine lokale Kopie des Parameters benötigt?
 - * *Ja*: Übergabe per Wert (vgl. Diskussionen auf den Seiten 168 und 590).
 - * *Nein*: Übergabe per `const&`, siehe Übergabe des Strings im Konstruktor.
6. Wenn Operationen verkettet werden sollen, ist das Objekt selbst per Referenz zurückzugeben. In C++ sind Anweisungen wie `a = b = c;` erlaubt und üblich, ebenso wie die schon bekannte Verkettung von Ausgaben mehrerer Variablen, etwa `cout << a << b << c << endl;`. Die Verkettung von Operatoren wird ausführlich in Kapitel 9 diskutiert. Operatoren sind Funktionen, für die dasselbe gilt. Nehmen wir zum Beispiel die Anwendung der Methode

```
// Methode der Klasse Rational
void add(const Rational& r);
```

von Seite 164 auf rationale Zahlen `a`, `b` und `c`. Die Operation `a.add(b);` ist damit möglich, nicht aber Verkettungen wie zum Beispiel

```
a.add(b).add(c); // a) entspricht: a += b; a += c;
a.add(b.add(c)); // b) entspricht: a += b += c;
```

Die Verkettungen könnten natürlich durch jeweils zwei einzelne Anweisungen ersetzt werden, aber hier geht es darum, zu zeigen, dass und wie Verkettungen mit der Rückgabe des Objekts selbst möglich gemacht werden. Betrachten wir beide Fälle:

- (a) Diese Anweisung wird von links abgearbeitet. `a.add(b)` muss dann ein Objekt zurückliefern, auf das dann `add(c)` angewendet wird. Auf einen Rückgabotyp `void` lässt sich keine Funktion anwenden, und der Compiler würde sich bitter beschweren. Dieses zurückgegebene Objekt ist sinnvollerweise nichts anderes als das durch die Addition veränderte Objekt `a`, hier `a` genannt. Aus `a.add(b)` ergibt sich also `a`, für das `add(c)` aufgerufen wird. In Einzelschritte zerlegt:

```
a.add(b).add(c);
  └─┬─┘
    a.add(c);
```

Damit ergibt sich eine veränderte Implementation (und Deklaration) für `add()`:

```
// veränderter Rückgabotyp:
Rational& Rational::add(const Rational& r) {
    zaehler = zaehler*r.nenner + r.zaehler*nenner;
    nenner = nenner*r.nenner;
    kuerzen();
    return *this;
}
```

Mit `*this` wird das Objekt, für das die Methode aufgerufen wird, zurückgegeben.

- (b) Ein Aufruf der Form `a.add(x)` verlangt, dass das Argument `x` vom Typ `Rational` ist. Da `x` identisch mit `b.add(c)` und gemeint ist, dass erst `c` auf `b` addiert wird, welches dann auf `a` addiert wird, folgt daraus, dass der Aufruf `b.add(c)` das veränderte `b` zurückgeben muss. Die unter a) angegebene Implementierung löst auch dieses Problem.

Natürlich kann von den Empfehlungen abgewichen werden – wenn es einen Grund dafür gibt. *Kleine* Klassenobjekte kosten nicht viel Zeit bei der Kopie und können daher per Wert übergeben werden statt per Referenz auf `const`.

20.2 Defensive Programmierung

Defensive Programmierung soll die Risiken falscher Algorithmen oder einer falschen Benutzung vermindern. So kann der »Buffer Overflow«, ein Fehler, der vielen Hackern Angriffsmöglichkeiten geboten hat und vermutlich noch bietet, durch defensive Programmierung verhindert werden. Einen Buffer Overflow zeigen die Beispiele 1 bis 3 auf Seite 194 ff. zur Eingabe einer Zeichenkette; Beispiel 4 zeigt, wie er verhindert wird. In diesem Abschnitt wird eine Auswahl weiterer Techniken genannt.

Allgemeine Empfehlungen

Programmcode kann schon durch vielerlei Maßnahmen verbessert werden. Manche davon können sehr aufwendig sein, wie etwa das Testen von Software, manche sind einfacherer Art. Eine kleine Auswahl:

- Programmcode verständlich und gut lesbar schreiben.
- Nicht das Rad neu erfinden, sondern erprobte Bibliotheken verwenden.
- Vier Augen sehen mehr als zwei. Ein Review des Codes durch eine *andere* Person als den Programmautor kann viele Fehler oder kritische Stellen finden.
- Eine damit verwandte Möglichkeit ist das »Pair Programming«, bei dem einer den Code schreibt und der andere über das zu lösende Problem nachdenkt, das Geschriebene kontrolliert und alles, was ihm dabei auffällt, gleich mit dem Autor bespricht. Beide sollen sich abwechseln. Pair Programming ist besonders durch das »Extreme Programming«-Buch von Kent Beck [Beck] bekannt geworden.
- Vor- und Nachbedingungen und Invarianten prüfen, sofern vertretbar (zur Diskussion des Aufwands siehe Seite 315).
- Exception-sicher programmieren (siehe Seiten 315 und 567).
- Nicht-reparierbare Fehler mit einer Exception melden (nicht mit Assertions).

Um nicht im Allgemeinen zu bleiben, gibt es im Folgenden eine Auswahl ganz konkreter Tipps, meistens ergänzt durch Programmbeispiele.

20.2.1 double- und float-Werte richtig vergleichen

Auf Seite 75 ist zu sehen, wie ein schlechter Vergleich zu undefiniertem Verhalten einer Schleife führen kann. Die dort gezogene Schlussfolgerung ist »Verwenden Sie nur integrale Datentypen wie `int`, `size_t` oder `char` als Laufvariable in Schleifen!«. Nur integrale Datentypen zu verwenden, hilft nicht, wenn tatsächlich `double`- oder `float`-Werte verglichen werden müssen, um in Abhängigkeit vom Vergleich Entscheidungen zu treffen. Dabei sind die Vergleiche `<`, `<=`, `>` und `>=` sicher. Problematisch sind `==` und `!=`, weil nur Bitmuster verglichen werden, ungeachtet der begrenzten Genauigkeit von Rechnungen und der Zahlendarstellung. Empfehlung: Vor der Rechnung den gewünschten Bereich festlegen, innerhalb dessen zwei Werte als gleich betrachtet werden sollen. Mathematiker verwenden oft »Epsilon« als Namen für einen solchen Bereich:

```
// x und y seien die zu vergleichenden Werte
double eps = 1.0e-8;    // Bereich
if (fabs(x-y) < eps) {  // fabs : Absolutbetrag
    // x und y werden als gleich betrachtet
}
else {
    // x und y sind ungleich
}
```

Ähnlich kann man vorgehen, wenn es um eine *relative* Abweichung gehen soll, zum Beispiel soll `y` nicht mehr als 0,1 Prozent von `x` abweichen:

```
double eps = 0.001;      // 0,1 %
if (fabs((y-x)/x) < eps) { // Risiko, siehe Text
    // x und y werden als gleich betrachtet
}
```

Der Bezugswert `x` darf nicht zu klein werden, um einen Divisions-Overflow zu vermeiden (bzw. einen Underflow bei Multiplikation beider Seiten mit `x`).

20.2.2 const verwenden

Um versehentliche Änderungen zu vermeiden, sollten alle unveränderliche Variablen und alle Methoden, die ein Objekt nicht verändern, `const` sein. Vorteil: Schon der Compiler prüft die Einhaltung der Absicht.

20.2.3 Anweisungen nach for/if/while einklammern

Nachträgliche Ergänzungen können ohne die Klammern leicht zu einem Fehler führen. Angenommen, dass in der Anweisung

```
if (x == 1)
    a = a + 3;
```

auch noch die Variable `counter` hochgezählt werden soll. Dies hat sich erst viel später herausgestellt, sodass nachträglich die `if`-Anweisung ergänzt wird:

```
if (x == 1)
    a = a + 3;
    ++counter;                // falsch
```

Tja, mal eben schnell ein Programm »verbessert«! Der Effekt: `++counter;` wird stets ausgeführt, egal ob `x` gleich 1 ist oder nicht! Hier fehlen die geschweiften Klammern. Daher gibt es die folgende *Empfehlung zur Schreibweise*. `if`-Anweisungen sollen stets mit geschweiften Klammern geschrieben werden, also zum Beispiel

```
if (x == 1) {
    a = a + 3;
}
```

Dies gilt auch, wenn der Inhalt des Blocks aus nur einer Zeile besteht. Einige der oben erwähnten Fehler entstehen dadurch gar nicht erst¹.

20.2.4 `int` und `unsigned/size_t` nicht mischen

Für die Anzahl der Objekte in einem Container wird im Allgemeinen der Typ `size_t` verwendet, weil die Anzahl nie negativ sein kann. Wenn so ein Wert unüberlegt mit einem `int`-Wert verglichen wird, kann es schwer zu entdeckende Fehler geben. Ein Beispiel finden Sie auf Seite 66.

20.2.5 `size_t` oder `auto` statt `unsigned int` verwenden

`size_t` ist groß genug, um die Größe eines jeden Objekts des aktuellen Systems aufzunehmen. Das ist bei `unsigned int` nicht garantiert. Das wirkt sich auch auf Konstante aus, zum Beispiel auf `string::npos`, das als Kriterium für eine erfolgreiche Suche verwendet wird. `npos` ist typischerweise die maximal mögliche `unsigned`-Zahl. Beispiel:

```
string zudurchsuchenderText("www.cppbuch.de");
unsigned int punktPosition = zudurchsuchenderText.find("."); // ??
if(punktPosition == string::npos) {
    cout << "'. nicht gefunden!" << endl;
}
```

Das mag auf einem 32-Bit-System funktionieren, auf dem der Typ von `npos` derselbe Typ wie `unsigned int` ist. Auf einem 64-Bit-System jedoch könnte `unsigned int` eine 32-Bit-Zahl sein und `npos` eine 64-Bit-Zahl. Das heißt, auf so einem 64-Bit-System wäre die Bedingung *niemals* wahr, ob der zu durchsuchende Text einen Punkt enthält oder nicht. Der tatsächliche Typ von `npos` ist `string::size_type`. `size_t` ist groß genug, und wenn der Typ von `size_t` nicht derselbe wie `string::size_type` sein sollte, gibt es auf jeden Fall eine definierte Typumwandlung. Noch besser ist es aber, den Compiler mit `auto` den richtigen Typ feststellen zu lassen:

```
auto punktPosition = zudurchsuchenderText.find("."); // !!
```

20.2.6 Postfix++ mit Präfix++ implementieren

`++i` und `i++` unterscheiden sich nur darin, dass im ersten Fall `i` erst hochgezählt und dann verwendet wird. Im zweiten Fall ist es umgekehrt. Um genau dieselbe Bedeutung des »Hochzählens«, das in einem binären Baum das Weiterschalten zum nächsten Schlüssel und bei einem Datum den nächsten Tag bedeuten kann, ohne Code-Replikation zu

¹ In diesem Buch wird aus Platzgründen gelegentlich davon abgewichen.

erreichen, ist die auf Seite 336 angegebene Methode gut geeignet. Der nachgestellte Operator `operator++(int)` wird mithilfe des Präfix-Operators implementiert:

```
X X::operator++(int) { // X hochzählen, Parameter wird nicht gebraucht
    X temp(*this);    // temporäres Objekt erzeugen
    ++*this;          // Präfix ++ aufrufen, *this wird modifiziert
    return temp;      // vorherigen Zustand zurückgeben
}
```

`X` kann jede Klasse sein, die einen Kopierkonstruktor und `operator++()` besitzt. Weil ein temporäres Objekt erzeugt wird, sollte man den nachgestellten `++`-Operator nicht als Ersatz für den vorangestellten `++`-Operator verwenden. Beispiel:

```
for(Iterator i = container.begin();
    i != container.end(); i++) { // ungünstig
    // ...
}
for(Iterator i = container.begin();
    i != container.end(); ++i) { // besser
    // ...
}
```

Der Compiler kann die Optimierung bei den Grunddatentypen selbst vornehmen; bei komplexeren Strukturen kann er das nicht ohne Weiteres: Ohne eine aufwendige Datenflussanalyse ist nicht feststellbar, ob die Bedeutung des Hochzählens in beiden Fällen exakt dieselbe ist.

20.2.7 Ein Destruktor darf keine Exception werfen

Die Einhaltung dieser Empfehlung garantiert, dass Destruktoren exception-sicher sind. Zur Erläuterung sei als Beispiel eine Klasse `Unsicher` mit zwei dynamischen Variablen gegeben:

```
class Unsicher {
public:
    Unsicher(int c, int d)
        : a(new X(c)), b(new X(d)) {
    }
    ~Unsicher() {
        delete a;    // siehe Text
        delete b;
    }
    // weggelassen: Methoden, die die Attribute ändern/verwenden
private:
    X* a;
    X* b;
};
```

`delete a;` ruft den Destruktor der Klasse `X` auf. Wenn dieser eine Exception wirft, wirft auch `~Unsicher()` eine Exception, und `delete b;` wird nie ausgeführt – ein Speicherleck. Eine `try/catch`-Konstruktion der Art

```
~Unsicher() {
    try {        // hilft nicht
```



```
        delete a;  
        delete b;  
    } catch(...) {  
        ; // nichts tun  
    }  
}
```

würde nicht helfen: Zwar könnte die Exception den Destruktor der Klasse `Unsicher` nicht verlassen, aber `delete b;` wird auch jetzt nicht ausgeführt, wenn `delete a;` eine Exception wirft. Also: Grundsätzlich jeden Destruktor so schreiben, dass er keine Exception wirft.

20.2.8 Typumwandlungsoperatoren vermeiden

Um die Typprüfung des Compilers nicht zu umgehen, sollte man Typumwandlungsoperatoren nur in begründeten Fällen einsetzen. Jede vom Compiler nicht gefundene Typverletzung ist eine verlorene Chance, einen möglichen Fehler zu finden. Wenn eine Typumwandlung gewünscht ist, ist es am besten, auf den Typumwandlungsoperator zu verzichten und stattdessen eine eigene Methode dafür zu schreiben. Eine implizite Typumwandlung ist dann unmöglich (siehe auch Übungsaufgabe von Seite 339).

20.2.9 explicit-Konstruktoren bevorzugen

Die Argumente des vorhergehenden Abschnitt gelten auch hier. Ein weiterer Vorteil ist die bessere Lesbarkeit des Programmcodes, weil sofort gesehen wird, dass eine Typumwandlung beabsichtigt ist. Ein Beispiel sehen Sie auf Seite 161.

20.2.10 Leere Standardkonstruktoren vermeiden

Objekte werden vom Konstruktor initialisiert. Manche IDEs schlagen gleich einen parameterlosen (Standard-)Konstruktor vor, oder er wird gedankenlos hingeschrieben (kann ja nicht schaden ...). Wenn er keinen Inhalt hat *und* ein weiterer Konstruktor existiert, können Flüchtigkeitsfehler produziert werden. Ein Beispiel: Ein Punkt soll stets definierte Koordinaten besitzen. Falls ein zusätzlicher leerer Standardkonstruktor existiert, kann die folgende Situation eintreten, wenn die Initialisierungsdaten vergessen werden:

```
Punkt p1(100, 200);    // ok, definierte Koordinaten  
Punkt p2;              // undefinierte Koordinaten, nicht beabsichtigt
```

Bei Nichtvorhandensein des leeren Standardkonstruktors würde der Compiler bei `p2` eine Fehlermeldung erzeugen.

20.2.11 Kopieren und Zuweisung verbieten

Klassen mit einem Zeiger als Attribut verwalten dynamisch angelegte Daten. Wie auf Seite 236 erläutert, benötigen diese in der Regel außer einem Destruktor einen eigenen Kopierkonstruktor und Zuweisungsoperator. Wenn versehentlich auf diese Elemente verzichtet wird, werden sie automatisch vom System erzeugt. Das kann katastrophale Folgen haben, weil automatisch erzeugte Kopierkonstruktoren und Zuweisungsoperatoren nur den Zeiger kopieren, nicht aber, auf was er zeigt. Die linke Seite der Abbildung 6.2 auf Seite 236 zeigt die Wirkung. Die folgende Klasse verhält sich wie in der Abbildung gezeigt:

```

class intArray { // fehlerhaft, siehe Text
public:
    intArray(int* p, int anz)
        : anzahl(anz), iptr(p) {
    }
    ~intArray() { delete[] iptr;}
    int& operator[](int i) { return *(iptr + i);}
private:
    size_t anzahl;
    int* iptr;
};

// mögliche Anwendung:
int main() {
    intArray iArr1(new int[10], 10);
    iArr1[0] = 100;
}

```

Wenn nun eine Kopie erzeugt wird, indem zu `main()` die Zeile

```
intArray iArr2 = iArr1;
```

hinzugefügt wird, ist das Programm fehlerhaft. Eine nachfolgende Änderung von `iArr1` wirkt sich bei `iArr2` aus und umgekehrt. Außerdem gibt es einen Crash am Ende des Programms, weil der Destruktor zweimal dasselbe Array zu löschen versucht.

Wenn nun gar keine Kopien gebraucht werden, müsste man also einen eigenen Kopierkonstruktor und Zuweisungsoperator schreiben, nur um deren automatische Generierung zu verhindern. Es gibt aber zwei bessere Lösungen. Die erste ist, Kopierkonstruktor und Zuweisungsoperator privat zu deklarieren; eine Implementierung ist nicht erforderlich:

```

class intArray {
public:
    // wie oben ...
private:
    intArray(intArray&); // verbietet Kopie
    void operator=(intArray&); // verbietet Zuweisung
    size_t anzahl;
    int* iptr;
};

```

Alternativ wird dem Compiler mit `= delete` mitgeteilt, dass er auf die automatische Erzeugung verzichten soll:

```

class intArray {
public:
    intArray(intArray&) = delete;
    void operator=(intArray&) = delete;
    intArray(int* p, int anz)
        : anzahl(anz), iptr(p) {
    }
    ~intArray() { delete[] iptr;}
    int& operator[](int i) { return *(iptr + i);}
private:

```

```
size_t anzahl;
int* iptr;
};
```

20.2.12 Vererbung verbieten

Manchmal möchte man die Ableitung von einer Klasse verhindern. Wenn zum Beispiel eine Klasse keinen virtuellen Destruktor hat, sind von ihr abgeleitete Klassen nicht polymorph ohne die Gefahr von Speicherlecks nutzbar, wie in Abschnitt 7.6.3 belegt, sodass man eine versehentliche polymorphe Nutzung ausschließen möchte. Ein anderer Grund mag sein, dass die Semantik von Funktionen nicht durch Redefinition in einer abgeleiteten Klasse verändert werden soll. Man kann nach [Str94] die Tatsache ausnutzen, dass bei virtuellen Basisklassen der Konstruktor der am meisten abgeleiteten Klasse für die Konstruktion des Basisklassensubobjekts verantwortlich ist (siehe Seite 291). Beispiel:

```
class Final {
    friend class NichtAbleitbar;
private:
    Final() {}
};

class NichtAbleitbar : public virtual Final {
public:
    NichtAbleitbar(int i): attr(i) {}
    int get() const { return attr; }
private:
    int attr;
};

class Versuch : public NichtAbleitbar {
public:
    Versuch(int i)
        : NichtAbleitbar(i) { // Fehlermeldung des Compilers
    }
};
```

Weil die Klasse `NichtAbleitbar` von `Final` erbt, muss sie auf deren Konstruktor zugreifen können. Da dieser privat ist, wird der Zugriff über die `friend`-Deklaration ermöglicht. Die Klasse `Versuch` ist zwar für die Konstruktion des Basisklassensubobjekts verantwortlich, hat aber keinen Zugriff – daher die Fehlermeldung des Compilers. Die virtuelle Vererbung ist entscheidend.

20.2.13 Defensiv Objekte löschen

Hier geht es um das Problem, dass ein Zeiger nach Löschen des Objekts weiterhin zugreifbar bleibt:

```
int *pa = new int[4];    // Array von int-Zahlen
// ... verwenden
delete [] pa;           // löschen
// .. mehr Programmcode
if(pa == 0) {            // Fehler
```

```
...
}
```

Der Fehler liegt in dem undefinierten Wert von `pa` nach der Löschoption. Falls ein Zeiger nach dem Löschen noch verwendet werden kann, sollte man ihn direkt nach dem `delete` mit `pa = 0;` auf `NULL` setzen. Dann kann er auf `0` geprüft werden oder es gibt eine definierte Fehlermeldung.

20.3 Exception-sichere Beschaffung von Ressourcen

Wenn eine Ressource beschafft werden soll, kann ein Problem auftreten. Das kann eine Datei sein, die nicht gefunden wird, oder ein Fehlschlag beim Beschaffen von Speicher. Weil die Probleme strukturell ähnlich sind, beschränke ich mich hier auf Probleme bei der dynamischen Beschaffung von Speicher. Das kann in einer Methode oder auch schon im Konstruktor auftreten. Ziel ist es, beim Auftreten von Exceptions kein Speicherleck zu erzeugen und die betroffenen Objekte in ihrem Zustand zu belassen.

20.3.1 Sichere Verwendung von `shared_ptr`

Bei der Konstruktion eines `shared_ptr` (Beschreibung Seite 851) soll die Erzeugung des Zeigers mit `new` stets innerhalb der Parameterliste geschehen.

```
Ressource pr = new Ressource(id);
// ... weiterer Code
shared_ptr<Ressource> sptr(pr);           // 1. falsch!
```

```
shared_ptr<Ressource> p(new Ressource(id)); // 2. richtig!
```

Begründung: Im Fall 1 kann es die folgenden Fehler geben:

- Es wäre möglich, `delete pr` aufzurufen. Bei der Zerstörung von `sptr` wird der Destruktor für `*pr` auch aufgerufen, dann also insgesamt *zweimal*.
- Es könnte sein, dass im Bereich »weiterer Code« eine Exception auftritt. Der resultierende Sprung des Programmablaufs aus dem aktuellen Kontext führt dazu, dass `delete` nicht mehr möglich ist. Das erzeugte Objekt bleibt unerreichbar im Speicher.

Im Fall 2 kann dies nicht geschehen: Wenn eine Exception geworfen wird, werden automatisch die Destrukturen aller auf dem Laufzeit-Stack befindlichen Objekte des verlassenen Gültigkeitsbereichs aufgerufen, also auch der Destruktor des `shared_ptr`-Objekts, der wiederum für das Löschen des übergebenen Objekts sorgt – eine Realisierung des Prinzips »Resource Acquisition Is Initialization« (RAII, siehe Glossar).

20.3.2 `shared_ptr` für Arrays korrekt verwenden

Der Destruktor eines `shared_ptr`-Objekts wendet `delete` auf den intern gespeicherten Zeiger an, wenn kein anderer `shared_ptr` auf die Ressource verweist (siehe Seite 851). Dies

kann zu einem Speicherleck führen, wenn der Zeiger mit `new []` erzeugt wurde, wie auf Seite 203 beschrieben. Hier ein Beispiel:

```
int* p = new int[10];
// p verwenden
// delete p; falsch!
delete [] p;    // richtig
```

Zwar kann es sein, dass im Fall der falschen Anweisung das Speicherleck nicht bemerkt wird, oder dass der Compiler aus dem Kontext den Fehler erkennt und korrigiert. Nach [ISO C++] ist das Verhalten jedoch undefiniert, das heißt, alle Möglichkeiten vom Weiterlaufen des Programms bis zum Absturz des Programms sind »legal«. Damit ist auch das Verhalten des folgenden Programms undefiniert:

```
void funktion() {
    shared_ptr<int> p(new int[10]);           // falsch
    // ... etliche Zeilen weggelassen
} // Memory-Leak möglich
```

Die Lösung des Problems besteht in der Übergabe eines `deleter`-Objekts an den `shared_ptr`. Wenn es so ein Funktionsobjekt gibt, wird dessen `operator()()` aufgerufen.

```
template<typename T>
struct ArrayDeleter {
    void operator()(T* ptr) {
        delete [] ptr;
    }
};

void funktion() {
    shared_ptr<int> p(new int[10], ArrayDeleter<int>()); // richtig
    // ... etliche Zeilen weggelassen
} // kein Memory-Leak, Array wird korrekt gelöscht
```

20.3.3 `unique_ptr` für Arrays korrekt verwenden

Bei `unique_ptr`-Objekten tritt dieselbe Problematik auf, wie oben im Abschnitt 20.3.1 beschrieben. Die Schnittstelle ist etwas anders:

```
template <class T, class D = default_delete<T>> class unique_ptr;
```

Der Typ des für die Löschung zuständigen Objekts gehört zur Schnittstelle. Wenn ein Arraytyp, gekennzeichnet durch `[]`, eingesetzt wird, kann der zweite Typ entfallen. Er wird dann durch den vorgegebenen (default) Typ für den Deleter ersetzt, der `delete []` aufruft. Die Funktion `f()` zeigt, wie es geht.

Listing 20.1: `unique_ptr` und Array

```
// cppbuch/k33/uniqueptr/array.cpp
#include<memory>

void f() {
    std::unique_ptr<int[]> arr(new int[10]); // int[] statt int
    // Benutzung des Arrays
```

```

    for(int i = 0; i < 10; ++i) {
        arr[i] = i;
        std::cout << arr[i] << std::endl;
    }
} // kein Memory-Leak, Array wird korrekt gelöscht

int main() {
    f();
}

```

Um den default-Template-Parameter sichtbar zu machen, könnte die erste Zeile in `f()` so geschrieben werden:

```
std::unique_ptr<int[], std::default_delete<int[]>> arr(new int[10]);
```

20.3.4 Exception-sichere Funktion

```

void func() {                // fehlerhaft, siehe Text
    Datum heute;             // Stack-Objekt
    Datum *pD = new Datum;   // Heap-Objekt beschaffen
    heute.aktuell();         // irgendeine Berechnung
    pD->aktuell();           // irgendeine Berechnung
    delete pD;              // Heap-Objekt freigeben
}

```

Wenn die Funktion `aktuell()` eine Ausnahme auswirft, wird der Destruktor von Objekt `heute` gerufen, und das Objekt wird vom Stack geräumt. Das Objekt, auf das `pD` zeigt, wird jedoch niemals freigegeben, weil `delete` nicht mehr erreicht wird und `pD` außerhalb des Blocks unbekannt ist:

```

int main() {
    try {
        func();
    }
    catch(...) {
        //... pD ist hier unbekannt
    }
}

```

Aus diesem Grund sollten ausschließlich Stack-Objekte (automatische Objekte) verwendet werden, wenn Exceptions auftreten. Dies ist immer möglich, wenn man Beschaffung und Freigabe eines dynamischen Objekts innerhalb eines Stack-Objekts versteckt. Das Hilfsmittel dazu kennen wir bereits, nämlich die »intelligenten« Zeiger aus Abschnitt 9.5:

```

void func() {
    // shared_ptr der Standardbibliothek, siehe Abschnitt 9.5.1. Header: <memory>
    std::shared_ptr<Datum> pDshared(new Datum);
    pDshared->aktuell();           // irgendeine Berechnung
}

```

Nun ist `pDshared` ein automatisches Objekt. Wenn jetzt eine Exception auftritt, gibt es kein Speicherleck, weil der Destruktor von `pDshared` den beschafften Speicher freigibt.

20.3.5 Exception-sicherer Konstruktor

Das Ziel, den Zustand eines Objekts bei Auftreten einer Exception unverändert zu lassen, ist in diesem Fall nicht erreichbar – das Objekt wird ja erst durch den Konstruktor erzeugt. Es geht also darum, dass

1. Ressourcen, die innerhalb des Konstruktors beschafft werden, freigegeben werden, und dass
2. Exceptions beim Aufrufer aufgefangen werden können.

In diesem Zusammenhang ist es wichtig zu wissen, wie sich C++ verhält, wenn in einem Konstruktor eine Exception auftritt.



Verhalten bei einer Exception im Konstruktor

- Für alle vollständig erzeugten (Sub-)Objekte wird der Destruktor aufgerufen. Unter »vollständig erzeugt« ist zu verstehen, dass der Konstruktor bis zum Ende durchlaufen wurde.
- Für *nicht* vollständig erzeugte (Sub-)Objekte wird *kein* Destruktor aufgerufen.

Das folgende Beispiel demonstriert dieses Verhalten. Ein Objekt der Klasse `Ganzes` enthält zwei Subobjekte der Typen `Teil1` und `Teil2`, die wie folgt definiert sind. Beachten Sie, dass der Konstruktor von `Teil2` zur Demonstration eine Exception wirft!

Listing 20.2: Klassen `Teil1` und `Teil2`

```
// cppbuch/k20/teil.h
#ifndef TEIL_H
#define TEIL_H
#include<iostream>

class Teil1 {
public:
    Teil1(int x)
        : attr(x) {
    }
    ~Teil1() {
        std::cout << "Teil1::Destruktor gerufen!" << std::endl;
    }
private:
    int attr;
};

class Teil2 {    // Konstruktor wirft zur Demonstration eine Exception
public:
    Teil2() {
        throw std::exception(); // auskommentieren:
        // dann wird der Destruktor gerufen, ansonsten NICHT!
    }
    ~Teil2() {
        std::cout << "Teil2::Destruktor gerufen!" << std::endl;
    }
};
#endif
```

Bei der Konstruktion des Objektes `ganzes` (siehe Beispiel unten) wird das Subobjekt `teil1` initialisiert. Die Konstruktion des Subobjekts vom Typ `Teil2`, die mit `new` versucht wird, schlägt jedoch fehl, weil der `Teil2`-Konstruktor eine Exception wirft.

Listing 20.3: Exception im Konstruktor

```
// cppbuch/k20/excImKonstruktor1.cpp
#include<iostream>
#include"teil.h"
using namespace std;

class Ganzes {
public:
    Ganzes() : teil1(99) {
        ptr = new Teil2;
    }
    ~Ganzes() {
        cout << "Ganzes::~Destruktor gerufen!" << endl;
        delete ptr;
    }
private:
    Teil1 teil1;
    Teil2* ptr;
    Ganzes(const Ganzes&);          // für Beispiel nicht erforderlich
    Ganzes& operator=(const Ganzes&); // für Beispiel nicht erforderlich
};

int main() {
    try {
        Ganzes ganzes;
    }
    catch(const exception& e) {
        cout << "Exception gefangen: " << e.what() << endl;
    }
}
```

Weil nur das Subobjekt `teil1` vollständig konstruiert wird, kommt auch nur dessen Destruktor zum Tragen. Die anderen Destrukturen werden nicht aufgerufen. Wird nun die Anweisung `throw exception();` auskommentiert oder gelöscht, werden alle Destrukturen aufgerufen.

Die am Anfang dieses Abschnitts genannten Punkte werden in diesem Beispiel bei Auftreten der Exception erreicht: Die »Ressource« `teil1` wird freigegeben, und die Exception wird in `main()` aufgefangen.

Ein Gegenbeispiel: Wenn die Klasse `Ganzes` *beide* Sub-Objekte mit `new` erzeugen würde, aber so, dass erst die Erzeugung des zweiten Sub-Objekts eine Exception wirft, wird der Destruktor für das erste Sub-Objekt *nicht* aufgerufen. Der Speicherplatz wird nicht wieder freigegeben.

Listing 20.4: Fehlerhafter Konstruktor

```
// Auszug aus cppbuch/k20/excImKonstruktor2.cpp
class GanzesMitFehler {
public:
    GanzesMitFehler() : ptr1(new Teil1(99)),
                      ptr2(new Teil2) {
    }
    ~GanzesMitFehler() {
        cout << "GanzesMitFehler::~Destruktor gerufen!" << endl;
        delete ptr1;
        delete ptr2;
    }
private:
    Teil1* ptr1;
    Teil2* ptr2;
    // Kopierkonstruktor und Zuweisungsoperator weggelassen
};
```

Mit `shared_ptr` wie in Abschnitt 20.3.1 geht man sicher. Wenn der `Teil2`-Konstruktor eine Exception wirft, wird der Destruktor von `Teil1` aufgerufen und gibt den Speicherplatz frei:

Listing 20.5: Konstruktor mit `shared_ptr`

```
// Auszug aus cppbuch/k20/excImKonstruktor3.cpp
class GanzesKorrigiert {
public:
    GanzesKorrigiert() : ptr1(new Teil1(99)),
                      ptr2(new Teil2) {
    }
    ~GanzesKorrigiert() {
        cout << "GanzesKorrigiert::~Destruktor gerufen!" << endl;
        // delete nicht notwendig wegen shared_ptr
    }
private:
    shared_ptr<Teil1> ptr1;
    shared_ptr<Teil2> ptr2;
};
```

20.3.6 Exception-sichere Zuweisung

Wenn bei einer Kopie Speicher beschafft werden muss, sollte man das zuerst tun! Der Grund: Falls es dabei eine Exception geben sollte, würden alle nachfolgenden, den Zustand des Objekts verändernden Anweisungen gar nicht erst ausgeführt. Die Problematik findet sich typischerweise beim Kopierkonstruktor und dem Zuweisungsoperator. Dazu gehören auch der Kurzformoperator `+=` und die Bildung temporärer Objekte. Sehen wir uns dazu eine mögliche Lösung der Aufgabe 9.6 von Seite 331 an:

```
// exception-sicher
MeinString& MeinString::operator=(const MeinString& m) { // Zuweisung
    char *p = new char[m.len+1]; // zuerst neuen Platz beschaffen
    strcpy(p, m.start);          // kopieren
```

```

delete [] start;          // alten Platz freigeben
len  = m.len;             // Verwaltungsinformation aktualisieren
start = p;
return *this;
}

```

Man könnte vordergründig daran denken, erst den alten Platz freizugeben, weil er ohnehin nicht mehr gebraucht wird, und dabei in der Summe sogar Speicher sparen, wenn nämlich bei `new` der alte Speicherplatz wiederverwendet werden sollte. Auch bräuchte man die Variable `p` nicht:

```

// NICHT exception-sicher!
MeinString& MeinString::operator=(const MeinString& m) { // Zuweisung
    delete [] start;          // weg damit, es wird schon gutgehen!
    start = new char[m.len+1]; // neuen Platz beschaffen
    strcpy(start, m.start);
    len  = m.len;             // Verwaltungsinformation aktualisieren
    return *this;
}

```

Wenn bei der Speicherplatzbeschaffung ein Problem auftreten sollte, wäre der Inhalt des Objekts durch das direkt vorangegangene `delete` zerstört! Von dem Problem, dass `m == this` sein könnte, will ich gar nicht erst reden.

Eine andere Möglichkeit, die Zuweisung exception-sicher zu gestalten, ist ein »swap-Trick«. Dazu wird eine temporäre Kopie erzeugt, und anschließend werden die Verwaltungsdaten vertauscht. Danach hat `*this` die richtigen Daten, und das temporäre Objekt wird korrekt vom Destruktor zerstört:

```

// exception-sicher
MeinString& MeinString::operator=(MeinString temp) { // Zuweisung
    // temporäre Kopie durch Übergabe per Wert
    // Nutzung der C++-Bibliothek, statt swap() selbst zu schreiben
    std::swap(temp.start, start);
    std::swap(temp.len, len);
    return *this;
}

```

Dieses Vorgehen ist bereits von Seite 218f. bekannt. Falls bei der Bildung von `temp` eine Exception vom Kopierkonstruktor geworfen würde, kämen die Folgezeilen nicht zur Ausführung, und das Objekt auf der linken Seite der Zuweisung bliebe unverändert. Dieses Muster lässt sich auf jede Klasse übertragen. Sie muss nur eine passende `swap()`-Methode besitzen:

```

// exception-sicherer Zuweisungsoperator
Klasse& Klasse::operator=(Klasse kopie) { // temporäre Kopie per Wert
    swap(kopie);          // wirft keine Exception
    return *this;
}

```

20.4 Aussagefähige Fehlermeldung ohne neuen String erzeugen

Exceptions sollen in bestimmten Fällen keine *neuen* Strings erzeugen, um Folge-Exceptions zu vermeiden – das könnte zum Beispiel bei nicht mehr ausreichendem Speicher geschehen. Man kann natürlich Fehlermeldungen statisch ablegen. Das ist jedoch manchmal nicht ausreichend, um eine genaue Information über die Fehlerursache zu erhalten. In solchen Fällen ist es zweckmäßig, einen Puffer vorzuhalten, der mit den Informationen über den Fehlerkontext gefüllt wird. Dies sei an einem praktischen Beispiel gezeigt. Dabei soll eine Exception den allgemeinen Text »Datei kann nicht geöffnet werden:« vorsehen, der um den Fehlerkontext, hier den Namen der betroffenen Datei, ergänzt wird. Das folgende Listing zeigt die Exception-Klasse:

Listing 20.6: Exception-Klasse mit kontextbezogener Fehlermeldung

```
// cppbuch/k25/binaer/IOException.h
#ifndef IOEXCEPTION_H
#define IOEXCEPTION_H
#include<string>
#include<cstring>
#include<exception>

namespace { // anonymer Namespace - nur in dieser Datei gültig
    char msg[100] = "Datei kann nicht geöffnet werden: ";
    size_t msglen = 0;
}

class IOException : public std::exception {
public:
    IOException(const std::string& filename) {
        if(msglen == 0) { // nur einmal berechnen!
            msglen = strlen(msg);
        }
        int maxRestlaenge = sizeof(msg) - msglen - 1;
        if(maxRestlaenge > 0) { // Puffer um Kontext ergänzen
            strncpy(msg+msglen, filename.c_str(), maxRestlaenge);
        }
    }
    const char* what() const noexcept {
        return msg;
    }
};
#endif
```

Wenn nun ein Programm nach einem IO-Fehler die Anweisung `throw IOException(dateiname);` ausführt, geschieht Folgendes:

- Die Endposition `msglen` des allgemeinen Textes wird gegebenenfalls ermittelt.
- Dann wird der restliche verfügbare Platz ermittelt, der 0 oder positiv sein kann. Die -1 berücksichtigt das abschließende Null-Byte.

- Der Puffer wird um den Namen der betroffenen Datei ergänzt. Die Funktion `strncpy()` (siehe Seite 880) sorgt dafür, dass die Puffergröße nicht überschritten wird.
- Das Anwendungsprogramm kann den Fehlertext mit der redefinierten Methode `what()` holen.

Dem Vorteil, dass keinerlei neuer Speicher angelegt wird und somit durch Speichermangel verursachte Folge-Exceptions unmöglich sind, stehen zwei Schwächen gegenüber, die bei der Anwendung zu berücksichtigen sind:

- Der Rückgabewert von `what()` bleibt nur bis zur nächsten `IOException` gültig, weil die statischen Daten in `msg` bei jeder `IOException` neu überschrieben werden.
- Die Klasse ist nicht thread-sicher, wie auch viele Klassen und Funktionen der Standardbibliothek. Wenn zwei verschiedene Threads gleichzeitig eine `IOException` werfen sollten, ist der Inhalt der Fehlermeldung undefiniert.

In den meisten Fällen spielen diese beiden Punkte keine Rolle.

20.5 Empfehlungen zur Thread-Programmierung

20.5.1 Warten auf die Freigabe von Ressourcen

Verwenden Sie die Konstruktionen

```
while(ressourcenNochNichtBereit) {
    cond.wait(lock);
}
```

mit einer Bedingungsvariablen `cond` und einem Lock-Objekt `lock`. Eine Begründung finden Sie auf Seite 433 und davor auch ein Beispiel. Die Alternative

```
while(ressourcenNochNichtBereit) {
    sleep(zeitdauer);
}
```

sollte nur genommen werden, wenn sich die Variante mit `wait()` als schwierig erweist; solche Fälle gibt es. Keinesfalls sollten Sie

```
while(ressourcenNochNichtBereit) {
}
```

schreiben – es würde sinnlos CPU-Zeit verbraten. Warum wird oben nicht

```
if(ressourcenNochNichtBereit) { // nicht empfehlenswert
    cond.wait(lock);
}
```

statt `while` gewählt, mögen Sie sich fragen. Der Grund liegt darin, dass *nach* der Rückkehr von `wait()`, aber noch *vor* der schließenden Klammer, ein konkurrierender Thread das Flag `ressourcenNochNichtBereit` wieder gesetzt haben könnte. Im Fall der `if`-Anweisung

würde dieses nicht mehr überprüft mit der möglichen Folge, dass zwei Threads gleichzeitig verändernd auf die Ressource zugreifen – ein Fehler!

20.5.2 Deadlock-Vermeidung

Das bekannteste Deadlock-Beispiel in der Informatik ist vermutlich das Problem der »Fünf Philosophen«, die mangels Koordination zu verhungern drohen. Falls Sie das Problem nicht kennen sollten, starten Sie am besten eine Internet-Suche mit dem Begriff »Fünf Philosophen«. Auf Seite 448 wird auf die Gefahr von Verklemmungen (Deadlocks) hingewiesen. Um die dort beschriebene Art von Deadlocks (es gibt leider noch andere) zu vermeiden, gibt es Empfehlungen für die Akquirierung und Freigabe von Ressourcen:

- Wenn mehr als eine Ressource angefordert wird, sollen alle beteiligten Threads sie in derselben Reihenfolge anfordern. Anders gesagt: Wenn ein Thread erst *A* und dann *B* benötigt, gilt diese Reihenfolge für alle Threads.
- Die zuletzt angeforderten Ressourcen sind zuerst wieder freizugeben. Wenn die Anforderung nach dem RAIL-Prinzip² geschieht, ist automatisch für die richtige Reihenfolge der Freigabe gesorgt.

Der erste Punkt ist nicht immer leicht zu bewerkstelligen. Sehen Sie sich dazu das einfache Beispiel der Überweisung von einem Konto auf das andere aus der Einführung an (Seite 30):

k1.überweisen(54688490, 1000.00)

Die Funktion muss während des Überweisungsvorgangs sicherstellen, dass die Zugriffe von anderen Threads auf beide beteiligte Konten gesperrt werden, etwa auf diese Art:

```
void ueberweisen(long zielkontoNr, double betrag) { // fehlerhaft!
    lock(this);           // eigenes Konto für andere sperren
    Konto* pk = getKonto(zielkontoNr);
    lock(pk);             // Ziel-Konto für andere sperren
    abheben(betrag);      // eigentliche Transaktion
    pk->einzahlen(betrag);
    release(pk);
    release(this);
}
```

Das Problem: Es könnte sein, dass *gleichzeitig* ein Betrag von dem anderen Konto *k2* auf das Konto *k1* überwiesen werden soll:

k2.überweisen(12573001, 20.95)

In diesem Fall könnten beide Threads gleichzeitig `lock(this)` ausführen – kein Problem, da es sich um verschiedene Objekte handelt. Aber dann bleiben sie aufeinander wartend in der Zeile `lock(pk)` hängen. In solchen Fällen kann eine Reihenfolge für alle vorgegeben werden. Im Beispiel oben wäre es zweckmäßig, das Konto mit der jeweils kleineren Kontonummer zuerst zu sperren.

20.5.3 notify_all oder notify_one?

`notify_one()` befreit *einen* auf die Ressource wartenden Thread, `notify_all()` *alle*. Im letzten Fall kommt nur einer zum Zuge, die anderen werden wieder in die Warteschlange

² Siehe Glossar

geschickt, sofern nur ein Thread weiterarbeiten darf. Das muss nicht immer so sein: Zum Beispiel muss ein verändernder Zugriff auf einen kritischen Bereich von nur einem Thread ausgeführt werden. Ein rein *lesender* Zugriff kann daher von vielen Threads gleichzeitig erfolgen, sobald der schreibende Thread die Änderungen vollendet hat. Dann ist ein `notify_all()` angebracht, wobei allerdings Lesen und Schreiben durch verschiedene Bedingungsvariablen gesteuert werden müssen, um die verschiedenen Freigabemöglichkeiten zu unterscheiden.

`notify_one()` kann Anwendung finden, wenn höchstens *ein* Thread die Ressource weiter nutzen soll *und* wenn jeder Thread die Ressource auf dieselbe Art nutzt *und* es nur eine Bedingung gibt, die ggf. zum Warten führt.

In allen anderen Fällen ist `notify_all()` angebracht.

Bei nicht ganz korrekter Programmierung ist `notify_all()` fehlertoleranter. Ein Beispiel:

1. Thread A blockiert einen Datenspeicher begrenzter Kapazität, um die Daten zu sortieren. Der Datenspeicher sei voll, es kann nichts mehr hinzugefügt werden.
2. Thread B möchte Daten entnehmen; damit würde Platz geschaffen. B muss aber warten, weil A noch nicht fertig ist.
3. Thread C möchte Daten in den Datenspeicher schreiben, kann es aber nur, wenn auch Platz vorhanden ist. Auch C wartet.
4. Thread A ist endlich fertig und sagt `notify_one()`. Vom Laufzeitsystem wird genau ein beliebiger Thread aus den Wartenden ausgewählt, in diesem Fall zufällig C.
5. Thread C kann nichts ausrichten und muss warten. Thread B könnte die Lage entschärfen, wartet aber immer noch.

Wenn Thread A in Punkt 4 `notify_all()` gesagt hätte, wäre B zum Zuge gekommen und hätte seinerseits mit `notify_all()` Thread C mitteilen können, dass der Schreibvorgang nunmehr möglich ist.



Tipp

Verwenden Sie im Zweifel lieber `notify_all()` statt `notify_one()`.

`notify_all()` ist bei vielen wartenden Threads natürlich aufwendiger. Andererseits ist der Aufwand, verglichen mit dem des Threads selbst, oft vernachlässigbar. `notify_all()` und `notify_one()` wirken nur im Moment des Aufrufs; er wird nicht gespeichert. Wenn aufgrund eines Programmfehlers ein `notify_all()` oder `notify_one()` ausgeführt wird, der Thread aber erst danach das zugehörige `wait()` erreicht, wird der Thread nie erlöst.

20.5.4 Performance mit Threads verbessern?

Generell kann die Performance einer Applikation mit Threads verbessert werden, wenn es mehr als einen Prozessor(kern) gibt. Dabei ist jedoch zu berücksichtigen, dass Threads Software komplexer und damit fehleranfälliger werden lassen, und dass die Thread-Verwaltung selbst einen bestimmten Aufwand erfordert. Dieser Aufwand soll den Performance-Gewinn nicht überschreiten.

**Tipp 1**

Parallelisieren Sie ein Programm mit Threads nur dann, wenn Sie wissen, dass es sich wirklich lohnt, nachgewiesen zum Beispiel mit Laufzeitmessungen an einem Prototypen.

**Tipp 2 (für Fortgeschrittene)**

Der Trend geht zu mehr Prozessorkernen. Wenn Sie zu schreibende Anwendungen davon profitieren lassen wollen und bereit sind, sich in die nicht immer einfache Materie einzuarbeiten, lohnt sich ein Blick auf OpenMP (<http://openmp.org>). OpenMP ist eine Multi-Plattform-Bibliothek zur Parallelisierung von Programmen und zur Kommunikation über shared Memory. Sie wird von vielen Compilern unterstützt.

21

Von der UML nach C++

Dieses Kapitel behandelt die folgenden Themen:

- Vererbung
- Interfaces
- Assoziationen
- Multiplizität
- Aggregation
- Komposition

Die Unified Modeling Language (UML) ist eine weit verbreitete grafische Beschreibungssprache für Klassen, Objekte, Zustände, Abläufe und noch mehr. Sie wird vornehmlich in der Phase der Analyse und des Softwareentwurfs eingesetzt. Auf die UML-Grundlagen wird hier nicht eingegangen; dafür gibt es gute Bücher wie [\[Oe\]](#). Hier geht es darum, die wichtigsten UML-Elemente aus Klassendiagrammen in C++-Konstruktionen, die der Bedeutung des Diagramms möglichst gut entsprechen, umzusetzen. Die vorgestellten C++-Konstruktionen sind Muster, die als Vorlage dienen können. Diese Muster sind nicht einzigartig, sondern nur Empfehlungen, wie man die Umsetzung gestalten kann. Im Einzelfall kann eine Variation sinnvoll sein.

21.1 Vererbung

Über Vererbung ist in diesem Buch schon einiges gesagt worden, das hier nicht wiederholt werden muss, siehe Kapitel 7 ab Seite 257 oder kurz im Glossar Seite 957. Die Abbildung 21.1 zeigt das zugehörige UML-Diagramm.



Abbildung 21.1: Vererbung

In vielen Darstellungen wird die Oberklasse oberhalb der abgeleiteten Unterklasse dargestellt; in der UML ist aber nur der Pfeil mit dem Dreieck entscheidend, nicht die relative Lage. In C++ wird Vererbung syntaktisch durch »: public« ausgedrückt:

```
class Unterklasse : public Oberklasse {
    // ... Rest weggelassen
};
```

21.2 Interface anbieten und nutzen

Interface anbieten

Die Abbildung 21.2 zeigt das zugehörige UML-Diagramm. Die Klasse Anbieter implementiert das Interface Schnittstelle-X. Bei der Vererbung stellt die abgeleitete Klasse die Schnittstellen der Oberklasse zur Verfügung. Insofern gibt es eine Ähnlichkeit, auch gekennzeichnet durch die gestrichelte Linie im Vergleich zum vorherigen Diagramm.

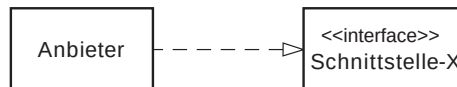


Abbildung 21.2: Interface-Anbieter

Die Ähnlichkeit wird in der Umsetzung nach C++ abgebildet: Anbieter wird von SchnittstelleX¹ abgeleitet. Um klarzustellen, dass um ein Interface geht, sollte SchnittstelleX abstrakt sein. Das Datenobjekt d wird nicht als const-Referenz übergeben, weil service() damit auch die Ergebnisse an den Aufrufer übermittelt.

¹ Die UML erlaubt Bindestriche in Namen, C++ nicht.

```

class SchnittstelleX {
public:
    virtual void service(Daten& d) = 0; // abstrakte Klasse, vgl. Seite 275 f.
};

class Anbieter : public SchnittstelleX {
public:
    void service(Daten& d) {
        // ... Implementation der Schnittstelle
    }
};

```

Interface nutzen

Bei der Nutzung des Interfaces bedient sich der Nutzer einer entsprechenden Methode des Anbieters. Die Abbildung 21.3 zeigt das zugehörige UML-Diagramm.



Abbildung 21.3: Interface-Nutzer

Ein Nutzer muss ein Anbieter-Objekt kennen, damit der Service genutzt werden kann. Aus diesem Grund wird in der folgenden Klasse bereits dem Konstruktor von Nutzer ein Anbieter-Objekt übergeben, und zwar per Referenz, nicht per Zeiger. Der Grund: Zeiger können NULL sein, aber undefinierte Referenzen gibt es nicht.

```

class Nutzer {
public:
    Nutzer(Anbieter& a)
    : anbieter(a) {
        daten = ...
    }
    void nutzen() {
        anbieter.service(daten);
    }
private:
    Daten daten;
    Anbieter& anbieter;
};

```

Nun kann man sich fragen, warum die Referenz oben nicht als `const` übergeben wird. Das kann je nach Anwendungsfall sinnvoll sein oder auch nicht. Es hängt davon ab, ob sich der Zustand des Anbieter-Objekts durch den Aufruf der Funktion `service(daten)` ändert. Wenn ja, zum Beispiel durch interne Protokollierung der Aufrufe, entfällt `const`.

21.3 Assoziation

Eine Assoziation sagt zunächst einmal nur aus, dass zwei Klassen in einer Beziehung (mit Ausnahme der Vererbung) stehen. Die Art der Beziehung und zu wie vielen Objekten sie aufgebaut wird, kann variieren. In der Regel gelten Assoziationen während der Lebensdauer der beteiligten Objekte. Nur kurzzeitige Verbindungen werden meistens nicht notiert. Ein Beispiel für eine kurzzeitige Verbindung ist der Aufruf `anbieter.service(daten)`; oben: `anbieter` kennt durch die Parameterübergabe das Objekt `daten`, wird aber vermutlich die Verbindung nach Ablauf der Funktion lösen.

Einfache gerichtete Assoziation

Das UML-Diagramm einer einfachen gerichteten Assoziation sehen Sie in Abbildung 21.4.

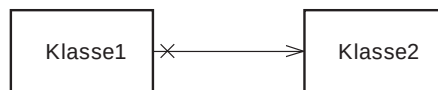


Abbildung 21.4: Gerichtete Assoziation

Mit »gerichtet« ist gemeint, dass die Umkehrung nicht gilt, wie zum Beispiel die Beziehung »ist Vater von«. Falls zwar Klasse1 die Klasse2 kennt, aber nicht umgekehrt, wird dies durch ein kleines Kreuz bei Klasse1 vermerkt. Es kann natürlich sein, dass eine Beziehung zwischen zwei Objekten *derselben* Klasse besteht. Im UML-Diagramm führt dann der von einer Klasse ausgehende Pfeil auf dieselbe Klasse zurück. In C++ wird eine einfache gerichtete Assoziation durch ein Attribut `zeigerAufKlasse2` realisiert:

```

class Klasse1 {
public:
    Klasse1() {
        : zeigerAufKlasse2(0) {
    }
    void setKlasse2(Klasse2* ptr2) {
        zeigerAufKlasse2 = ptr2;
    }
private:
    Klasse2* zeigerAufKlasse2;
};
  
```

Ein Zeiger ist hier besser als eine Referenz geeignet, weil es sein kann, dass das Kennenlernen erst nach dem Konstruktoraufruf geschieht.

Gerichtete Assoziation mit Multiplizität

Die Multiplizität, auch Kardinalität genannt, gibt an, zu wie vielen Objekten eine Verbindung aufgebaut werden kann. In Abbildung 21.5 bedeutet die 1, dass jedes Objekt der Klasse2 zu genau einem Objekt der Klasse1 gehört. Das Sternchen * bei Klasse2 besagt,

dass einem Objekt der Klasse1 beliebig viele (auch 0) Objekte der Klasse2 zugeordnet sind.

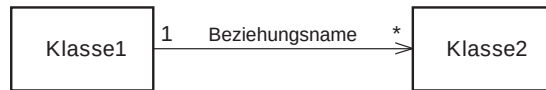


Abbildung 21.5: Gerichtete Assoziation mit Multiplizitäten

Im folgenden C++-Beispiel entspricht `Zeitung` der Klasse1 und `Abonnent` der Klasse2. Eine Zeitung kennt N Abonnenten. Ein Abonnent hat diese Zeitung abonniert oder nicht. Im letzteren Fall ist er entweder Abonnent einer anderen Zeitung oder kein Abonnent, allenfalls ein potentieller. Um die Multiplizität auszudrücken, bietet sich ein `vector` an:

```

class Abonnent; // Vorwärtsdeklaration

class Zeitung {
public:
    void addAbonnent(Abonnent* a) {
        abonnenten.push_back(a);
    }
private:
    std::vector<Abonnent*> abonnenten;
};

class Abonnent {
public:
    void abonniere(const Zeitung& z) {
        meineZeitung = &z;
    }
private:
    Zeitung* meineZeitung;
};
  
```

Im Beispiel sind beide Klassen voneinander abhängig. Die Vorwärtsdeklaration ist notwendig, damit der Compiler beim Compilieren der Klasse `Zeitung` die dort angesprochene Klasse `Abonnent` kennt.



Mehr zur Auflösung von gegenseitigen Abhängigkeiten lesen Sie in Abschnitt 4.8.

Einfache ungerichtete Assoziation

Eine ungerichtete Assoziation wirkt in beiden Richtungen und heißt deswegen auch bidirektionale Assoziation. Die Abbildung 21.6 zeigt das UML-Diagramm.

Wenn zwei sich kennenlernen, kann das mit einer ungerichteten Assoziation modelliert werden. Zur Abwechslung sei die Umsetzung in C++ nicht mit zwei, sondern nur mit einer Klasse (namens `Person`) gezeigt. Das heißt, die Klasse hat eine Beziehung zu sich selbst, siehe Abbildung 21.7. Solche Assoziationen werden auch rekursiv genannt und dienen zur Darstellung der Beziehung verschiedener Objekte derselben Klasse.



Abbildung 21.6: Ungerichtete Assoziation

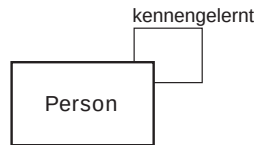


Abbildung 21.7: Rekursive Assoziation

Die Umsetzung in C++ wird am Beispiel von Personen gezeigt, die sich gegenseitig kennenlernen. Ein Aufruf `A.lerntkennen(B)`; impliziert, dass B auch A kennenlernen.

Listing 21.1: Assoziation: Personen lernen sich kennen

```
// cppbuch/k21/main.cpp
#include "Person.h"

int main() {
    Person mabuse("Dr. Mabuse");
    Person klicko("Witwe Klicko");
    Person holle("Frau Holle");
    mabuse.lerntkennen(klicko);
    holle.lerntkennen(klicko);
    // ...
}
```

Die entscheidende Methode der Klasse `Person` ist `lerntkennen(Person& p)` (siehe unten). Beim Eintrag in die Menge der Bekannten wird festgestellt, ob der Eintrag vorher schon vorhanden war. Wenn nicht, wird er auch auf der Gegenseite vorgenommen.

Listing 21.2: Klasse Person

```
// cppbuch/k21/Person.h
#ifndef PERSON_H_
#define PERSON_H_
#include <iostream>
#include <set>
#include <string>

class Person {
public:
    Person(const std::string& name) :
        name_(name) {
    }
}
```

```

virtual ~Person() {
}

const std::string& getName() const {
    return name_;
}

void lerntkennen(Person& p) {
    bool nichtvorhanden = bekannte.insert(p.getName()).second;
    if (nichtvorhanden) { // falls unbekannt, auch bei p eintragen
        p.lerntkennen(*this);
    }
}

void bekannteZeigen() const {
    std::cout << "Die Bekannten von " << getName() << " sind:" << std::endl;
    for (auto iter = bekannte.begin(); iter != bekannte.end(); ++iter) {
        std::cout << *iter << std::endl;
    }
}
private:
    std::string name_;
    std::set<std::string> bekannte;
};
#endif

```

21.3.1 Aggregation

Die »Teil-Ganzes«-Beziehung (englisch *part of*) wird auch *Aggregation* genannt. Sie besagt, dass ein Objekt aus mehreren Teilen besteht (die wiederum aus Teilen bestehen können). Die Abbildung 21.8 zeigt das UML-Diagramm. Die Struktur entspricht der gerichteten Assoziation, sodass deren Umsetzung in C++ hier Anwendung finden kann. Ein Teil kann für sich allein bestehen, also auch vom Ganzen gelöst werden. Letzteres geschieht in C++ durch Nullsetzen des entsprechenden Zeigers.

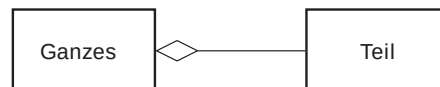


Abbildung 21.8: Aggregation

21.3.2 Komposition

Die Komposition ist eine spezielle Art der Aggregation, bei der die Existenz der Teile vom Ganzen abhängt. Damit ist gemeint, dass die Teile zusammen mit dem Ganzen erzeugt und auch wieder vernichtet werden. Ein Teil ist somit stets genau einem Ganzen zugeordnet; die Multiplizität kann also nur 1 sein. Die Abbildung 21.9 zeigt das UML-Diagramm.



Abbildung 21.9: Komposition

Es empfiehlt sich, bei der Umsetzung in C++ Werte statt Zeiger zu nehmen. Dann ist gewährleistet, dass die Lebensdauer der Teile an das Ganze gebunden ist:

```
class Ganzes {
public:
    Ganzes(int datenFuerTeil1, int datenFuerTeil2)
        : erstesTeil(datenFuerTeil1), zweitesTeil(datenFuerTeil2) {
        // ...
    }
    // ...
private:
    Teil erstesTeil;
    Teil zweitesTeil;
};
```

22

Performance, Wert- und Referenzsemantik

Dieses Kapitel behandelt die folgenden Themen:

- Performanceproblem Wertsemantik
- Optimierung durch Referenzsemantik für R-Werte
- Vermeidung temporärer Objekte

Von Wertsemantik spricht man, wenn auf Objekte direkt zugegriffen wird. Dem gegenüber steht die Referenzsemantik, die einen Zugriff auf Objekte nur über Verweise oder Referenzen zulässt. Java verwendet bei Klassenobjekten Referenzsemantik, C++ baut auf der Wertsemantik auf. In C++ lässt sich eine Referenzsemantik gut mit Zeigern oder Referenzen realisieren. Die Konsequenzen seien anhand einiger Beispiele gezeigt, wobei die Existenz der Klasse A gegeben sein soll:

```
A a1;  
A a2 = a1;           // Wertsemantik: Eine echte Kopie wird erzeugt  
A* aptr1 = new A;  
A* aptr2 = aptr1;     // Referenzsemantik: nicht das Objekt, sondern  
                     // nur der Verweis/Zeiger wird kopiert.  
if (a1 == a2) ...     // Wertsemantik: Prüfung auf Gleichheit  
if (aptr1 == aptr2) ... // Referenzsemantik: Prüfung auf Identität
```


Wenn zwei Zeiger gleich, aber nicht Null sind, verweisen sie auf dasselbe Objekt. Wenn zwei Zeiger ungleich sind, verweisen sie auf verschiedene Objekte, die untereinander gleich oder ungleich sein können. Die Tabelle 22.1 zeigt eine Übersicht, wobei mit Referenzen auch Zeiger gemeint sind.

Tabelle 22.1: Wert- und Referenzsemantik

Unterschied bezüglich	Wertsemantik	Referenzsemantik
Zugriff	direkt	über Referenz
==	prüft Gleichheit	vergleicht Identität
=	kopiert Objekt	kopiert Referenz
Sichtbarkeitsregeln	für Objekte	für Referenzen
Lebensdauer	Objekt existiert nur innerhalb des Gültigkeitsbereichs	Objekt existiert unabhängig von der Referenz
Speicherort für Objekte	Stack	Heap
Parameterübergabe	f (A a)	f (const A& a) f (const A* ptr)
Wertrückgabe	A g()	A& g(), A* g() (semantisch nicht dasselbe)

Besonders beim Einsatz von Containern sollte man sich genau überlegen, ob echte Kopien (Wertsemantik) gewünscht sind oder nicht:

```
// Vektor mit (möglicherweise aufwendigen) Kopien
vector<A> v1(10000);

// Initialisierung aller Elemente (weggelassen)

vector<A> v2 = v1;    // 10000 Aufrufe von A(const A&).
v2[1].setAttribut(100); // wirkt sich nicht auf v1[1] aus.

// Vektor mit Zeigern
vector<A*> vp1(10000);

// Initialisierung aller Elemente (weggelassen)

vector<A*> vp2 = vp1;  // Kein Aufruf von A(const A&).
vp2[1]->setAttribut(100); // wirkt sich auf v1[1] aus.
```

Wenn keine Kopien gewünscht sind, empfiehlt es sich, mit Zeigern zu arbeiten. Das kann bei großen Containern einen erheblichen Performance-Gewinn bedeuten. Dabei ist natürlich zu beachten, dass mit `new` angelegte Objekte nach Verwendung irgendwann mit `delete` zerstört werden müssen. Wenn man sich darum nicht kümmern möchte, sollte man im Vektor die Zeiger als `shared_ptr` ablegen:

```
for(int i=0; i < vp1.size(); ++i) {
    vp1[i] = std::shared_ptr<A>(new A(i));
}
```



Informationen zu `shared_ptr` finden Sie auf den Seiten 344 und 851.

22.1 Performanceproblem Wertsemantik

Die Wertsemantik führt manchmal zu überflüssigen Kopien. Sehen wir uns dazu einen `++`-Operator an, der zwei Strings verkettet und dabei den als vorhanden angenommenen `++`-Operator nutzt:

```
StringTyp operator+(const StringTyp& a, const StringTyp& b) {
    return StringTyp(a) += b;
}
```

Bei der Parameterübergabe wird der Aufruf des Kopierkonstruktors durch die Übergabe von Referenzen vermieden. Ohne weitere Optimierung wird er aber für `StringTyp(a)` aufgerufen und noch einmal bei der Rückgabe des Ergebnisses. Der Compiler kann entsprechend der Bemerkung auf Seite 159 den zweiten Aufruf eliminieren (englisch *named return value optimization (NRVO oder kurz RVO)*). Dies kann leicht gezeigt werden, wenn der Kopierkonstruktor mit einer protokollierenden Ausgabe wie etwa

```
std::cout << "Kopierkonstruktor gerufen" << std::endl;
```

versehen wird. Die zwei Fälle (mit und ohne Optimierung) können mit dem GNU C++-Compiler verglichen werden, indem die Übersetzung ohne beziehungsweise mit dem Parameter `-fno-elide-constructors` durchgeführt wird.

22.1.1 Auslassen der Kopie

Das Auslassen der Kopie (englisch *copy elision*) durch den Compiler steigert die Performance. Das führt auf eine weitere Optimierungsmöglichkeit, wenn temporäre Objekte beteiligt sind. Sehen Sie sich dazu die folgenden beiden Varianten an (nach [Abr]):

```
// Auszug aus cppbuch/k22/RVO/main.cpp
StringTyp str1("Erster");
StringTyp str2("Zweiter");
StringTyp erg1(str1 + str2);           // 1) siehe unten
StringTyp erg2(StringTyp("temporaer") + str2); // 2) siehe unten
```

`str1` und `str2` sind benannte Variablen, also eindeutig L-Werte. `StringTyp("temporaer")` hingegen ist ein R-Wert – er kann nicht auf der linken Seite einer Zuweisung stehen. Betrachten Sie nun die Konsequenzen bei Aufruf des obigen `++`-Operators:

1. `StringTyp erg1(str1 + str2);`
ohne RVO: Der obige `++`-Operator hat `a` als ersten Parameter. Davon erzeugt er eine Kopie (1. Aufruf des Kopierkonstruktors mit "Erster"). An diese Kopie wird `b` angehängt. Das Ergebnis "ErsterZweiter" wird per `return` zurückgegeben (2. Aufruf des Kopierkonstruktors mit "ErsterZweiter"). Die Bildung von `erg1` erfordert schließlich den 3. Aufruf des Kopierkonstruktors.
mit RVO: Der Aufruf des Kopierkonstruktors bei der Rückgabe wird eingespart.
2. `StringTyp erg2(StringTyp("temporaer") + str2);`
ohne RVO: Verhalten wie unter 1. beschrieben.

mit RVO: Verhalten wie unter 1. beschrieben. Aber: Eine genaue Betrachtung zeigt, dass das temporäre Objekt `StringTyp("temporaer")` nach Aufruf der Operatorfunktion nicht weiter benötigt wird. Der erste Kopierkonstruktor wäre überflüssig, wenn dieses temporäre Objekt direkt an die Stelle von `a` treten könnte. Man müsste nur dem Compiler die Möglichkeit geben, RVO einzusetzen.

Dazu wird der erste Parameter einfach *per Wert* übergeben, und auf die explizite Kopie wird verzichtet.

```
StringTyp operator+(StringTyp a, const StringTyp& b) {
    return a += b;
}
```

Der Compiler hat nun die Möglichkeit, das temporäre Objekt direkt an der Stelle von `a` zu verwenden. Ergebnis: Es ist nur *ein einziger* Aufruf des Kopierkonstruktors notwendig, nämlich der zur Erzeugung von `erg2`!

Wir haben hier das auf den ersten Blick paradox erscheinende Verhalten, dass die Übergabe *per Wert* performanter als die Übergabe *per Referenz* sein kann. Falls der erste Parameter ein nicht temporäres Objekt ist, wird der Kopierkonstruktor vom Compiler ganz normal aufgerufen. Daraus ergibt sich die folgende



Empfehlung

Falls in einer Funktion die lokale Kopie eines Arguments benötigt wird (und nur dann!) sollte das Argument *per Wert* übergeben werden.

Dann hat der Compiler die Möglichkeit, mit RVO den Kopierkonstruktor für temporäre Objekte einzusparen. Im Fall des obigen `++`-Operators gilt dies nur für das Argument `a`, nicht für `b`.

22.1.2 Temporäre Objekte bei der Zuweisung

Die oben beschriebene Problematik gilt auch in anderen Zusammenhängen. Beim Aufruf

```
ergstr = stra + strb; // Alle Variablen sind vom Typ StringTyp.
```

wird ohne RVO auf der rechten Seite der Kopierkonstruktor einmal bei der Bildung des temporären Objekts aufgerufen, dann ein weiteres Mal bei der Rückgabe und ein drittes Mal beim Aufruf des Zuweisungsoperators, der dem bekannten Muster aus Abschnitt 9.2.2 folgt:

```
// Auszug aus cppbuch/k22/RVO/einfacherstring.h
StringTyp& operator=(StringTyp s) { // Zuweisungsoperator
    swap(s);
    return *this;
}
```

Der `StringTyp`-Zuweisungsoperator übernimmt den Parameter *per Wert* (= Aufruf des Kopierkonstruktors) und vertauscht die Kopie mit `*this`. Danach gibt der Destruktor den alten Wert der linken Seite der Zuweisung, der nun in `s` steckt, frei. Das Überladen von Operatoren hat einen Preis, wie man sieht. *Mit* RVO wird wie oben ein Aufruf weniger

gebraucht. Der C++-Standard schreibt die Optimierung nicht vor; sie wird dem Compiler überlassen. Ein semantisch äquivalenter Funktionsaufruf etwa der Art

```
ergstr.verketten(stra, strb);
```

benötigt zwar eine `new`-Operation, wenn der aufnehmende String keine Platzreserve hat, würde aber gänzlich ohne temporäre Objekte und Aufruf des Kopierkonstruktors auskommen. Aus diesem Grund hat sich das C++-Standard-Komitee eine Spracherweiterung einfallen lassen, die im nächsten Abschnitt beschrieben wird.

22.2 Optimierung durch Referenzsemantik für R-Werte

Das Problem ist klar zu identifizieren. Es ist die Wertsemantik, die zur Kopie von Werten auch in den Fällen führt, in denen die Weiterleitung einer Referenz genügen würde.

- Die Funktion `operator+(StringTyp, const StringTyp&)` gibt ein Ergebnis zurück, das temporär ist und nach Ende der Funktion ohnehin verschwindet. Mit anderen Worten, es könnte direkt auf der rechten Seite einer Zuweisung eingesetzt werden, anstatt den Kopierkonstruktor aufzurufen.
- Ähnliches gilt für die Zuweisung selbst. Statt die rechte Seite der Zuweisung zu kopieren, könnte die rechte Seite direkt an die Stelle von `ergstr` treten. Das geht natürlich nur, wenn rechts wirklich ein temporäres Objekt auftritt, das nach Vollendung der Zuweisung verschwindet. Eine Zeile wie `ergstr = stra;` erfordert jedoch eine echte Kopie, weil beide Variablen danach bestehen bleiben.

Es genügt also in bestimmten Fällen, wenn Objekte *bewegt* anstatt *kopiert* werden (englisch *move semantics* bzw. *copy semantics* oder *value semantics*). Die Frage ist, wie ein Compiler klar erkennen kann, ob bei einer Zuweisung eine echte Kopie notwendig ist oder nicht. Das entscheidende Kriterium zeigen die folgenden Zeilen:

```
ergstr = stra;           // Die rechte Seite ist ein L-Wert (siehe Seite 953).
ergstr = stra + strb;    // Die rechte Seite ist ein R-Wert.
```

Um per Programm R-Werte explizit berücksichtigen zu können, wurde im neuen C++-Standard die Bindung temporärer Objekte an Referenzen durch die syntaktische Erweiterung `&&` für Referenzen auf R-Werte eingeführt:

```
void func(const Typ& t); // bindet an L-Wert
void func(Typ&& t);      // bindet an R-Wert
```

Der `const`-Qualifizierer fehlt typischerweise, weil Änderungen an den temporären Objekten möglich sein sollen (siehe unten). Beide Referenztypen werden über den normalen Mechanismus des Überladens von Funktionen eingebunden.

Um im Folgenden die Effekte am Beispiel zu zeigen, wird als Basis eine einfache String-Klasse `StringTyp`, die nur die wichtigsten Operationen enthält, vorausgesetzt. Diese Klasse wird anschließend um Referenzen auf R-Werte erweitert.

Listing 22.1: Klasse StringTyp

```

// cppbuch/k22/move/einfacherstring.h
#ifndef EINFACHERSTRING_H
#define EINFACHERSTRING_H
#include<cstddef>           // size_t
#include<cstring>           // strlen()
#include<iostream>         // ostream

class StringTyp {
public:
    StringTyp(const char* s)           // Konstruktor
        : len(strlen(s)), start(new char[len+1]) {
        strcpy(start, s);
    }

    StringTyp(const StringTyp& s)       // Kopierkonstruktor
        : len(s.len), start(new char[len+1]) {
        strcpy(start, s.start);
    }

    ~StringTyp() {                   // Destruktor
        delete [] start;
    }

    StringTyp& operator=(StringTyp s) { // Zuweisungsoperator
        swap(s);
        return *this;
    }

    StringTyp& operator+=(const StringTyp& s) { // Verkettungsoperator
        char* temp = new char[len + s.len + 1]; // neuen Platz beschaffen
        strcpy(temp, start);                    // Teil 1 kopieren
        strcpy(temp + len, s.start);            // Teil 2 kopieren
        delete [] start;                        // alten Platz freigeben
        len += s.len;                          // Verwaltungsinformation aktualisieren
        start = temp;
        return *this;
    }

    const char* c_str() const { return start; } // C-String zurückgeben
private:
    size_t len;                             // Länge
    char *start;                             // Zeiger auf den Anfang
    void swap(StringTyp& m) {                // vertauschen
        std::swap(start, m.start);
        std::swap(len, m.len);
    }
};

// Ausgabeoperator
inline std::ostream& operator<<(std::ostream &os, const StringTyp& s) {
    os << s.c_str();
}

```

```

    return os;
}

// binärer Verkettungsoperator
inline StringTyp operator+(StringTyp a, const StringTyp& b) {
    return a += b;
}
#endif

```



Hinweise

Um das Beispiel auf den Effekt der Referenzen auf R-Werte zu beschränken, wurden optimierende Maßnahmen wie die Wiederbenutzung von Speicher oder Vorhalten von Speicher, wie in Abschnitt 6.1.1, beschrieben, weggelassen.

In der Datei *cppbuch/k22/move/einfacherstring.h* auf der DVD sind zusätzlich Testausgaben eingebaut, die die Aufrufe und die Anzahl der new-Operationen dokumentieren. Die Testausgaben sind durch auskommentieren der Zeile `#define TEST` abschaltbar.

Um die Wirkung mit oder ohne Referenzen auf R-Werte auf dem Computer nachvollziehen können, gibt es ein *main*-Programm mit einigen temporären Objekten:

Listing 22.2: main-Programm

```

// cppbuch/k22/move/main.cpp
#define EINFACH // siehe unten
#ifdef EINFACH
#include "einfacherstring.h"
#else
#include "movingstring.h"
#endif
#include <iostream>
using namespace std;

int main() {
    StringTyp a("einA");
    StringTyp b("einB");
    StringTyp c("einC");
    cout << " Fall 1 : a = \"Hallo\" + b;" << endl;
    a = "Hallo" + b;

    cout << "\n Fall 2 : a = b + \" hallo\";" << endl;
    a = b + " hallo";

    cout << "\n Fall 3 : a = StringTyp(\"guten\") + \" Tag\";" << endl;
    a = StringTyp("guten") + " Tag";

    cout << "\n Fall 4 : a = b + c;" << endl;
    a = b + c;

    cout << "\n Fall 5 : StringTyp neu = b + c;" << endl;
    StringTyp neu = b + c;
}

```

```

cout << "\n Fall 6 : neu = a + \"eins\" + \"zwei\" + \"drei\";" << endl;
neu = a + "eins" + "zwei" + "drei";

cout << "na = " << a << endl;
cout << "b = " << b << endl;
cout << "c = " << c << endl;
cout << "neu= " << neu << endl;

cout << "\n Fall 7 : neu = a + b + c + neu;" << endl;
neu = a + b + c + neu;
}

```

Performancekriterium

`new` und `delete`-Operationen sind zeitraubend. Im Folgenden wird die Anzahl eingesparten der `new`-Operationen als Maßstab für die Verbesserung der Performance genommen. Das `main`-Programm verursacht 50 `new`-Operationen, wenn `einfacherstring.h` inkludiert wird. Diese Zahl wird durch Referenzen auf R-Werte deutlich reduziert, wie unten zu sehen.

Um diese Referenzen im konkreten Fall der Klasse `StringTyp` nutzen zu können, werden unter anderem die Deklarationen

```

StringTyp(StringTyp&&);           // bewegender (nicht kopierender) Konstruktor
StringTyp& operator=(StringTyp&&); // bewegendes Zuweisung

```

benötigt. Die Änderungen der Klasse `StringTyp` sind in der Datei `cppbuch/k22/move/movingstring.h` berücksichtigt. Im `main`-Programm oben wird mit dem Makro `EINFACH` entschieden, welche der beiden Dateien inkludiert wird. Um den Effekt der Referenzen auf R-Werte ungestört sehen zu können, wird die Wegoptimierung der Konstruktoraufrufe abgeschaltet. Das heißt, `main.cpp` wird mit der Option `-fno-elide-constructors` übersetzt, wenn `make` aufgerufen wird (siehe `cppbuch/k22/move/makefile`).

22.2.1 Bewegender Konstruktor

Der bewegende Konstruktor (englisch *moving constructor*) heißt so, weil er das Objekt anscheinend *bewegt*, nicht kopiert. Tatsächlich übernimmt er nur die Daten des Parameter-Objekts:

```

// Auszug aus cppbuch/k22/move/movingstring.h
StringTyp(StringTyp&& s)           // bewegender Konstruktor
: len(s.len), start(s.start) { // Speicher übernehmen
    s.start = 0;                 // nicht vergessen! (siehe Text)
}

```

Der Konstruktor beschafft keinerlei Speicherplatz auf dem Heap und ist daher sehr schnell. Das Nullsetzen des Zeigers `s.start` ist unbedingt notwendig, damit der `delete []`-Aufruf des Destruktors von `s` unschädlich bleibt. Bei der Erzeugung des neuen Objekts werdem dem temporären Objekt, das danach nicht mehr gebraucht wird, die Ressourcen (d.h. der zugewiesene Speicher) »gestohlen« (englisch *resource stealing*).

Wenn der Parameter `s` ein L-Wert wäre, müsste er für eine weitere Verwendung erhalten bleiben, und man dürfte ihm nicht die Ressourcen entziehen. In diesem Fall ruft der Compiler den normalen Konstruktor `StringTyp(const StringTyp& s)` auf.

22.2.2 Bewegender Zuweisungsoperator

Der bewegende Zuweisungsoperator unterscheidet sich vom normalen Zuweisungsoperator dadurch, dass keine Kopie angelegt wird. Wie bei dem Kopierkonstruktor wird die Ressource des Parameters in Besitz genommen. Damit kein Speicherleck entsteht, muss der vorher belegte Speicher freigegeben werden.

```
// Auszug aus cppbuch/k22/move/movingstring.h
StringTyp& operator=(StringTyp&& s) { // bewegender Zuweisungsoperator
    delete [] start;                // alten Speicher freigeben
    start = s.start;                // Ressource aneignen
    s.start = 0;                    // wegen Destruktor von s
    len = s.len;
    return *this;
}
```

Die Variante mit `swap()` ist natürlich auch möglich. Die Funktion `swap()` vertauscht die Daten des Objekts `*this` mit denen von `s`. Damit wird erreicht, dass `*this` die gewünschten Daten erhält und der Destruktor von `s` die alten Daten von `*this`, die sich nunmehr in `s` befinden, korrekt löscht. Die Parameterübergabe per `&&` kann hier als zerstörender Lesevorgang aufgefasst werden. Da es sich um ein temporäres Objekt handelt, ist dies kein Problem.

```
// Variante
StringTyp& operator=(StringTyp&& s) { // bewegender Zuweisungsoperator
    swap(s);                          // Ressourcen und Verwaltungsinformationen tauschen
    return *this;
}
```

Der bisher verwendete Zuweisungsoperator (`operator=(StringTyp)` von Seite 592) übergibt den Parameter per Wert, was für temporäre *und* andere Objekte möglich ist. Wenn der bewegende Zuweisungsoperator hinzukommt, kann der Compiler nicht entscheiden, welcher Zuweisungsoperator verwendet werden soll. Aus diesem Grund muss der Zuweisungsoperator `operator=(StringTyp)` geändert werden:

```
// Auszug aus cppbuch/k22/move/movingstring.h
// geändert! Jetzt Übergabe per const& statt per Wert
StringTyp& operator=(const StringTyp& s) {
    StringTyp tmp(s);
    swap(tmp);
    return *this;
}
```

Dieser Zuweisungsoperator wird nunmehr für L-Werte aufgerufen, der andere Zuweisungsoperator (`operator=(StringTyp&&)`) für R-Werte.

22.3 Ein effizienter binärer Plusoperator

Die im vorangehenden Abschnitt eingeführten Elemente »bewegender Konstruktor« und »bewegender Zuweisungsoperator« reduzieren die Zahl der `new`-Operationen von 50 auf 43. Damit ist noch nicht das Problem aller überflüssigen temporären Objekte gelöst. Es gibt vier Möglichkeiten der Verkettung zweier Strings, die L-Werte oder R-Werte sein können (a, b und c sind String-Objekte):

```
a = "Hallo" + b;           // R-Wert L-Wert
a = b + "hallo";          // L-Wert R-Wert
a = StringTyp("guten") + " Tag"; // R-Wert R-Wert
a = b + c;                 // L-Wert L-Wert
```

Hier sind Objekte mit Namen (L-Werte), aber auch temporäre Objekte beteiligt.

Der binäre `++`-Operator kann bei der Rückgabe mit einer Typumwandlung vom bewegenden Konstruktor profitieren:

```
StringTyp operator+(const StringTyp& x, const StringTyp& y) { // Verkettung
    return static_cast<StringTyp&&>(StringTyp(x) += y);
}
```

Da die Wegoptimierung des Kopierkonstruktors vom C++-Standard nicht vorgeschrieben wird, wird hier nachgeholfen, indem durch die Typumwandlung definitiv der bewegende Kopierkonstruktor aufgerufen wird. Achtung: Der Rückgabebetyp ist `StringTyp`, nicht `StringTyp&&`! Der Unterschied: Im ersten Fall tritt das Objekt an die Stelle des Funktionsaufrufs, im zweiten Fall eine Referenz auf das (lokale!) Objekt, das nach Ende der Funktion dann nicht mehr existiert. Mit dieser Typumwandlung reduziert sich die Anzahl der `new`-Operationen für das obige `main`-Programm auf nur noch 32!

Es würde in allen vier Fällen der (normale) Kopierkonstruktor jeweils nur noch einmal aufgerufen, sowie jeweils einmal der Moving-Konstruktor und der Moving-Zuweisungsoperator.

`std::move()`

An dieser Stelle sei die Standardfunktion `move()` erwähnt, die dieselbe Typumwandlung bewirkt:

```
StringTyp operator+(const StringTyp& x, const StringTyp& y) { // Operator 1
    return std::move(StringTyp(x) += y);
}
```

Die Funktion `move(par)` kann dann vorteilhaft eingesetzt werden, wenn ihr Parameter `par` danach definitiv nicht mehr benötigt wird. Falls der Parameter ein L-Wert sein sollte, wäre eine weitere Verwendung kritisch – schließlich sind ihm möglicherweise die Ressourcen entzogen worden!



Mehr zu `move` lesen Sie in Abschnitt 27.2.

22.3.1 Kopien temporärer Objekte eliminieren

Der Operator `operator+(const StringTyp&, const StringTyp&)` berücksichtigt keine temporären Objekte in der Parameterliste. Wenn die Eigenschaft R-Wert der temporären Objekte ausgenutzt wird, ergeben sich drei weitere Varianten des `++`-Operators:

```
inline StringTyp operator+(StringTyp&& x, const StringTyp& y) { // Operator 2
    return std::move(x += y);
}

inline StringTyp operator+(const StringTyp& x, StringTyp&& y) { // Operator 3
    // return StringTyp(x) += y; besser, weil ein new weniger:
    return std::move(y.frontinsert(x));
}

inline StringTyp operator+(StringTyp&& x, StringTyp&& y) { // Operator 4
    return std::move(x += y);
}
```

Zu Operator 3: Der Aufruf `y.frontinsert(x)` fügt den String `x` *am Anfang* von `y` ein. Dafür ist nur eine `new`-Operation notwendig. `StringTyp(x) += y`; hingegen benötigt ein `new` zur Konstruktion der temporären Kopie von `x` und ein weiteres für den `+=`-Operator. Die Funktion `frontinsert()` finden Sie in der Datei `cppbuch/k22/move/movingstring.h`.

Den Operatoren 2 bis 4 wird jeweils mindestens ein temporäres Objekt übergeben. Wenn mit diesem in der Operatorfunktion gearbeitet und die Referenz per `move()` darauf zurückgegeben wird, erübrigt sich der normale Kopierkonstruktor vollends. Es wird der bewegende Kopierkonstruktor aufgerufen. Es bleibt nur noch der bewegende Zuweisungsoperator – eine erhebliche Verbesserung! Sie finden das vollständige Programm im Verzeichnis `cppbuch/k22/move`.

Mit diesen Operatoren reduziert sich die Anzahl der `new`-Operation auf 24, also weniger als die Hälfte der anfänglichen 50!

22.3.2 Verbesserung durch verzögerte Auswertung

Nur im Fall des Operators 1 von oben ist einmal der Aufruf des Kopierkonstruktors notwendig, weil keins der beiden Argumente ein R-Wert ist. Das erste Argument durch eine Übergabe per Wert zu ersetzen, würde nichts bringen, weil es einen Compilations-Konflikt mit Operator 2 geben würde. Am besten wäre es, wenn dieses funktionslokale Objekt gar nicht erst erzeugt werden müsste und das Ergebnis direkt an den Zielort geschrieben werden könnte. Dann wäre keinerlei Kopierkonstruktoraufruf notwendig. In der Klasse `StringTyp` ist diese effiziente Variante etwa mit einer Member-Funktion realisierbar, in der die Parameter per Referenz übergeben werden:

```
void StringTyp::verketten(const StringTyp& x, const StringTyp& y) {
    char* temp = new char [x.len + y.len + 1]; // Platz beschaffen
    strcpy(temp, x.start); // Teil 1 kopieren
    strcpy(temp + x.len, y.start); // Teil 2 kopieren
    delete [] start; // alten Speicherplatz freigeben
    start = temp; // Verwaltungsdaten aktualisieren
    len = a.len + b.len;
}
```

Der Aufruf `ergstr = stra + strb`; würde einfach durch den Funktionsaufruf `ergstr.verketten(stra, strb)`; ersetzt werden. Die Funktion vermeidet jeden Kopierkonstruktoraufruf und ist daher die schnellere Lösung. Aber – ist es denn wirklich notwendig, aus Performance-Gründen auf das C++-Markenzeichen »Überladen von Operatoren« zu verzichten? Die Antwort ist nein, wenn wir in die C++-Trickkiste greifen.

Ein binärer Operator weiß nicht, was mit seinem Ergebnis geschieht: Deswegen muss er ein temporäres Ergebnisobjekt zur Verfügung stellen, könnte man annehmen. Tatsächlich aber zeigt der Einsatz nur zwei typische Benutzungsweisen:

```
StringTyp neu = stra + strb; // Konstruktion eines neuen Objekts
ergstr = stra + strb;       // Zuweisung an ein vorhandenes Objekt
```

Der Aufruf des Kopierkonstruktors wäre vermeidbar, wenn es gelänge, die Beschaffung von Speicher und das Kopieren der Daten in den Kopierkonstruktor zu verlagern, die Auswertung der +-Operation also zu verzögern. Das kann mit dem Trick nach [Str], Kap. 22, erreicht werden, in `operator+()` nur die Referenzen der beteiligten Objekte zu übertragen und sonst nichts zu tun. Die eigentliche Arbeit muss dann im Kopierkonstruktor bzw. im Zuweisungsoperator erledigt werden. Die Übertragung wird mit einem Hilfstyp `Plus` vorgenommen, der nur ein Paar Referenzen auf die beteiligten Objekte in sich trägt. (Sie finden das vollständige Programm im Verzeichnis *cppbuch/k22/plus/*.)

```
// Auszug aus cppbuch/k22/plus/movingstring.h
class StringTyp;
typedef std::pair<const StringTyp&, const StringTyp&> Plus;
```

`pair<U, V>` ist ein Standard-Template (Header `<utility>`, siehe Seite 750), das ein Wertepaar kapselt. Das Paar kann auch aus Elementen verschiedenen Typs bestehen. Für die Template-Parameter wird hier `StringTyp&` eingesetzt. Der Trick besteht darin, den Operator 1 oben so umzubauen, dass er den String nicht verkettet, sondern nur die Adressen der Parameter weiterleitet:

```
Plus operator+(const StringTyp& x, const StringTyp& y) { // Operator 1a
    return Plus(x, y);
}
```

Das Ergebnis der Verkettung führt entweder zu einem neuen Objekt, oder es wird einem anderen Objekt zugewiesen. Es werden also ein entsprechender Konstruktor und ein Zuweisungsoperator benötigt. Beide erledigen die Verkettung, wobei der Zuweisungsoperator auf den Konstruktor zugreift. Nur im Konstruktor gibt es ein `new`:

```
StringTyp(const Plus& pa) // Konstruktor für Plus
: len(pa.first.len + pa.second.len),
  start(new char [len + 1]) { // Platz beschaffen
    strcpy(start, pa.first.start); // Teil 1 kopieren
    strcpy(start + pa.first.len, pa.second.start); // Teil 2 kopieren
}

StringTyp& operator=(const Plus& pa) { // Plus auswerten
    StringTyp temp(pa); // Konstruktor für Plus-Argument
    swap(temp);         // *this mit temp vertauschen
    return *this;
}
```

Damit wird im `main`-Beispiel die Anzahl der `new`-Operationen auf nunmehr 21 verringert. Auf der Basis dieser Lösung wird der Ablauf der Anweisung

```
ergstr = stra + strb + strc;
```

analysiert. In den folgenden Zeilen sehen Sie die einzelnen Schritte der Abarbeitung.

```
ergstr = operator+(stra, strb) + strc; // Operator 1a von oben  
ergstr = Plus(stra, strb) + strc;    // Ergebnis
```

Weil der `++`-Operator ein `StringTyp`-Objekt erwartet, wird `Plus(stra, strb)` umgewandelt:

```
ergstr = StringTyp(Plus(stra, strb)) + strc;
```

Das Ergebnis der Umwandlung ist ein temporäres Objekt, hier `TEMP1` genannt. Dazu passend kommt der Operator 2 von oben (`operator+(StringTyp&&, const StringTyp&)`) zur Geltung:

```
ergstr = operator+(TEMP1, strc);
```

Dessen Ergebnis ist wieder ein temporäres Objekt, hier `TEMP2` genannt. `TEMP2` wird Parameter des bewegenden Zuweisungsoperators:

```
ergstr.operator=(TEMP2);
```

Das Ergebnis dieser Operation ist die veränderte Variable `ergstr`. Im Vergleich zu dem fiktiven Funktionsaufruf

```
ergstr.verketten(stra, strb);
```

von Seite 591 ist die Performance nur um den Aufruf des Konstruktors und Kopierkonstruktors der Klasse `Plus` schlechter. Da nur zwei Referenzen erzeugt bzw. kopiert werden, ist der zusätzliche Aufwand im Vergleich zur Speicherbeschaffung mit `new` und der Kopie der Strings unerheblich, und der gewünschte Performance-Gewinn ist erreicht. Der zusätzliche Konstruktor `StringTyp(const Plus&)` käme bei der Erzeugung eines `StringTyp`-Objekts aus zwei anderen zum Tragen, zum Beispiel

```
StringTyp verkettet = einString + zweiterString;
```

Ausblick

Für mehr als zwei rechte Operanden lässt sich eine Lösung gänzlich ohne temporäre Objekte finden, wenn Expression Templates [Ve95a] verwendet werden. Expression Templates dienen dazu, einen Ausdruck zur Compilationszeit mithilfe der Template-Metaprogrammierung, die in Abschnitt 6.4 (Seite 251) kurz beschrieben wird, zu optimieren. Eine andere Möglichkeit ist, anstelle von `Plus` ein Tupel (Klasse `tuple` von Seite 752) einzusetzen. Im Gegensatz zu `Plus` kann ein Tupel-Objekt eine beliebige Anzahl von Parametern speichern. Abschnitt 24.10.1 zeigt diesen Ansatz für eine 2-dimensionale Matrix.

23

Effektive Programm- erzeugung

Dieses Kapitel behandelt die folgenden Themen:

- Automatische Ermittlung von Abhängigkeiten
- Makefile für Verzeichnisbäume
- Automatische Erzeugung von Makefiles
- Erzeugen von Bibliotheken
- Code Bloat bei der Instanziierung von Templates vermeiden
- GNU Autotools
- Alternative CMake

Dieses Kapitel beschreibt ergänzende und mächtigere Techniken für *make* als in der Einführung vorgestellt. Wie dort wird auch für die Beispiele dieses Kapitels die Klasse `Rational` von Seite 164 ff. benutzt. Es gibt die Dateien *main.cpp*, *rational.h* und *rational.cpp*.



Die Grundlagen zu Make finden Sie in Kapitel 17.

23.1 Automatische Ermittlung von Abhängigkeiten

Von welchen Header-Dateien eine cpp-Datei abhängig ist, lässt sich durch einen Blick auf die `#include`-Anweisungen der cpp-Datei feststellen. Die `#include`-Anweisungen werden ohnehin bei der Compilation durch den C++-Präprozessor gelesen – der hat die benötigten Informationen! Die Entwickler des GNU-C++-Compilers haben daran gedacht, diese Informationen zur Verfügung zu stellen. Dazu wird der Compiler mit der Option `-M` aufgerufen, die alle Abhängigkeiten in einer Form ausgibt, die für ein Makefile geeignet ist. Um die System-Header-Dateien auszublenden, bietet sich die Option `-MM` an. So ergibt der Aufruf `g++ -MM rational.cpp > rational.d` die Datei *rational.d* mit dem Inhalt

```
rational.o: rational.cpp rational.h
```

Die Optionen `-M` und `-MM` implizieren die Option `-E`, die dafür sorgt, dass der Compiler nach Aufruf des Präprozessors nichts mehr tut, also weder compilieren noch linken. Es ist üblich, die Abhängigkeiten in Dateien mit der Endung `.d` (für dependency) zu speichern, hier also *rational.d*, und diese Dateien mit `include` im Makefile einzuschließen. Das Minuszeichen vor `include` unterdrückt Meldungen bei fehlenden d-Dateien – ein normaler Zustand, schließlich sollen sie ja bei Bedarf erzeugt werden. In der oben gezeigten Datei *rational.d* wird berücksichtigt, dass eine Änderung in der Datei *rational.h* zur Neubildung von *rational.o* führen muss, nicht aber, dass auch *rational.d* vielleicht neu berechnet werden müsste. Es könnte ja sein, dass zum Beispiel eine `#include "neu.h"`-Anweisung in *rational.h* aufgenommen wurde. Eine Änderung von *neu.h* müsste dann auch zur Neucompilation führen. Aus diesem Grund sollte *rational.d* selbst auch von *rational.h* abhängen, das heißt, *rational.d* sollte besser so aussehen:

```
rational.d rational.o: rational.cpp rational.h
```

Man könnte nun die Zeichenkette »rational.d«, gefolgt von einem Leerzeichen, in die gleichnamige Datei schreiben, etwa

```
%.d: %.cpp
→ @echo -n $@ " " > $@
→ $(CXX) -MM >> $@ $<
```

und die Ausgabe von `$(CXX) -MM` anhängen. Das Symbol $\rightarrow$ steht für das Tabulatorzeichen. Zur Erinnerung: `$@` gibt das Ziel und `$<` die erste Abhängigkeit an. `-n` unterdrückt den Zeilensprung. Die zwei Aufrufe der Shell könnten zusammengefasst werden:

```
→ @echo -n $@ " " > $@ && $(CXX) -MM >> $@ $<
```

`&&` ist die Verkettung von Befehlen in der Shell. Es gibt aber auch die Option `-MT`, mit der der Teil vor dem Doppelpunkt definiert werden kann:

```
→ $(CXX) -MM -MT "$@ $(patsubst %.d,%.o,$@)" -MF $@ $<
```

`-MT` schreibt in unserem Beispiel erst »rational.d«, dann »rational.o« als Ergebnis der Ersetzung mit `patsubst`. Die Anführungszeichen sorgen dafür, dass alles dazwischen zur

Option `-MT` gehört. `-MF` gibt den Namen der zu erzeugenden Datei an, der identisch mit dem Ziel ist. Am Ende der Aktion folgt der Name der zu analysierenden Datei `$<`, also der entsprechende `cpp`-Dateiname. Vermutlich ist diese Lösung effizienter als die vorherigen, weil die `d`-Datei nur einmal angefasst wird. Manche wählen noch standardmäßig die Option `-MG` in der Aktion für `d`-Dateien. Sie bewirkt, dass fehlende Header-Dateien nicht als Fehler gelten. Dies ist dann wichtig, wenn Header-Dateien noch in einem anderen Schritt generiert werden sollen, etwa bei der Verarbeitung von Grammatiken mit den Programmen *yacc* oder *bison*. Ich empfehle, die Option `-MG` nicht einzusetzen, um Fehler schneller erkennen zu können – es sei denn, man braucht sie aus den genannten Gründen wirklich. Die Datei *m4.mak* auf der Basis von *m3.mak* von Seite 522 zeigt, wie es geht:

Listing 23.1: Makefile mit automatischer Generierung der Abhängigkeiten

```
# cppbuch/k23/abhaengigkeiten/m4.mak
.PHONY: all clean
CXX := g++
CXXFLAGS := -c -g -Wall
LDFLAGS := -g
objs := $(patsubst %.cpp,%.o,$(wildcard *.cpp))
deps := $(objs:.o=.d)
all: projekt.exe
    -include $(deps)

projekt.exe: $(objs)
    -> $(CXX) $(LDLAGS) -o $@ $^

%.d: %.cpp
    -> $(CXX) -MM -MT "$@ $(patsubst %.d,%.o,$@)" -MF $@ $<

%.o: %.cpp
    -> $(CXX) $(CXXFLAGS) $<

clean:
    -> rm -f $(objs) $(deps)
```

Die Definition der Variablen `deps` zeigt eine abkürzende Art der Musterersetzung, die Langform ist `deps := $(patsubst %.o,%.d,$(objs))`. Eine Besonderheit gibt es beim Löschen der Dateien zu beobachten: Wenn `make -f m4.mak clean` *zweimal* direkt nacheinander aufgerufen wird, werden die `d`-Dateien vor dem zweiten Aufruf erst neu erzeugt, dann gelöscht. Der Grund liegt in der `include`-Anweisung, die unabhängig von jedem Ziel ausgewertet wird, d.h. auch bei `clean`.

23.1.1 Getrennte Verzeichnisse: *src*, *obj*, *bin*

Bei größeren Projekten ist es üblich, Quellen- und Objektverzeichnisse zu trennen. Ein Vorteil ist, dass das Quellenverzeichnis nicht durch generierte Dateien »verunreinigt« wird und ohne Platzvergeudung oder Löschooperationen gesichert werden kann. Es wird oft *src* (für das engl. *source*) genannt. Die Verzeichnisnamen sind natürlich frei wählbar, aber es gibt Konventionen. Daran orientiert, nenne ich das Verzeichnis, in dem die Objektdateien liegen, *obj* und das Verzeichnis mit der ausführbaren Datei *bin*. Die Abhängigkeitsdateien (Endung `.d`) sind im Verzeichnis *obj*, weil auch sie generierte Dateien sind und jederzeit

neu erzeugt werden können. Damit ergibt sich nach Ablauf des *make*-Kommandos das folgende, mithilfe des Unix-Programms *tree* erzeugte Abbild der Verzeichnisstruktur:

```

.                               Projektebene
|-- bin
|   '-- projekt.exe             Executable
|-- makefile
|-- obj
|   |-- main.d
|   |-- main.o
|   |-- rational.d
|   '-- rational.o
'-- src
    |-- main.cpp
    |-- rational.cpp
    '-- rational.h

```

Die Verzeichnisse *obj* und *bin* werden bei Bedarf neu angelegt. Das folgende *makefile* leistet das Gewünschte, ohne dass die *cpp*-Dateien und ihre Abhängigkeiten spezifiziert werden müssten. Es müssen nur die Verzeichnisnamen definiert werden:

Listing 23.2: Makefile mit Generierung der Abhängigkeiten und getrennten Verzeichnissen

```

# cppbuch/k23/verzTrennung/makefile
CXX := g++
CXXFLAGS := -c -g -Wall
LDFLAGS := -g
# Festlegen der Verzeichnisnamen
SRCDIR := src
OBJDIR := obj
BINDIR := bin

EXEFILE := projekt.exe

cppfiles := $(wildcard $(SRCDIR)/*.cpp)
objects := $(subst $(SRCDIR)/, $(OBJDIR)/, $(cppfiles:.cpp=.o))
deps := $(objects:.o=.d)

.PHONY: all clean

all: $(BINDIR)/$(EXEFILE)
    -include $(deps)

$(OBJDIR)/%.d: $(SRCDIR)/%.cpp
    ↪ mkdir -p $(@D)
    ↪ $(CXX) -MM -MT "$@ $(patsubst %.d,%.o,$@)" -MF $@ $<

$(OBJDIR)/%.o: $(SRCDIR)/%.cpp
    ↪ @echo compiling $< ...
    ↪ $(CXX) $(CXXFLAGS) $(INCLUDE) $< -o $@

$(BINDIR)/$(EXEFILE): $(objects)
    ↪ mkdir -p $(BINDIR)
    ↪ $(CXX) -o $@ $^ $(LDFLAGS)

```

```
clean:
→ $(RM) -r -f $(OBJDIR)
→ $(RM) -r -f $(BINDIR)
```

Was geschieht im Einzelnen? Zunächst einmal werden die cpp-Dateien mit dem von oben bekannten `wildcard`-Kommando ermittelt. Das Quellenverzeichnis `$(SRCDIR)/` muss angegeben werden.

Im nächsten Schritt wird zur Ermittlung der Objektdateien `subst` eingesetzt. `subst` (für Substitution oder Ersatz) hat drei Argumente: zu ersetzender String, Ersatzstring, zu modifizierender Text. Der Text besteht aus der Liste der Namen der cpp-Dateien, deren Endung `.cpp` durch `.o` ersetzt wird. In dieser Liste wird die Zeichenkette `src/` durch `obj/` ersetzt. Das Ergebnis ist die Liste der zu erzeugenden Objektdateien. Daraus wird in der nächsten Zeile die Liste der `.d`-Dateien erzeugt, indem die Endung `.o` durch `.d` ersetzt wird.

`mkdir -p` erzeugt das Verzeichnis, falls es noch nicht vorhanden ist. Dabei ist `@D` der Verzeichnisanteil (`D` = directory) des Ziels. Das Ziel `clean` löscht die Verzeichnisse mit den generierten Dateien mit dem Programm, das in der Variablen `$(RM)` definiert wurde. Die Voreinstellung ist `rm`, der Unix-Befehl zum Löschen. Der Rest ist wie auf Seite 603.

23.2 Makefile für Verzeichnisbäume

Ergänzend zur oben beschriebenen Trennung von Verzeichnissen ist es üblich, dass es Unterverzeichnisse entsprechend den einzelnen Modulen gibt. Jedem Modul kann ein Namespace zugeordnet werden, sodass die Verzeichnis-Hierarchie die Namespace-Hierarchie abbilden kann. Das Verzeichnis der Objektdateien spiegelt die Struktur des Quellenverzeichnisses wieder. Bezogen auf das bisher betrachtete einfache Beispiel, aufgeteilt in zwei Module `appl` und `rational`, könnte der Verzeichnisbaum wie folgt aussehen (ohne Makefiles):

```
Projekt
|-- bin                      // enthält Executable
|   '-- projekt.exe
|-- dist
|   '-- gezippt.zip          // Distribution
|-- obj                      // automatisch generiertes Verzeichnis
|   |-- appl
|       |-- main.d
|       '-- main.o
|   '-- rational
|       |-- rational.d
|       '-- rational.o
'-- src                      // Quellverzeichnis
    |-- appl                 // enthält Modul appl, die Anwendung
    |   '-- main.cpp
    '-- rational             // enthält Modul rational
```

```
|-- rational.cpp
'-- rational.h
```

Den Zusammenhang mit Namespaces zeigen die folgenden Code-Fragmente, wobei auch die Schachtelung von Namespaces gezeigt wird (obwohl es hier nur das zu schachtelnde Modul `rational` gibt):

```
// cppbuch/k23/dirTree/src/rational/rational.h
#ifndef PROJEKT_RATIONAL_H
#define PROJEKT_RATIONAL_H
namespace projekt {
namespace rational {
    class Rational {
    public: // Rest weggelassen
    }; // Klasse Rational
} // Namespaces
#endif
```

```
// cppbuch/k23/dirTree/src/rational/rational.cpp
#include "rational.h"
namespace projekt {
namespace rational {
    // hier folgen alle Funktionsdefinitionen
} // Namespaces
```

```
// cppbuch/k23/dirTree/src/appl/main.cpp
#include "../rational/rational.h"
using projekt::rational::Rational;
int main() {
    Rational a,b; // usw. (Rest von main() weggelassen)
}
```

Oft enthalten Verzeichnisse unterhalb der obersten Ebene des Quellenverzeichnis selbst wieder Makefiles, die von der jeweils oberen Ebene aufgerufen werden, rekursiver Aufruf genannt. Diese Makefiles müssen natürlich angelegt werden. Wenn viele ausführbare Programme zu erzeugen sind, ist das ein sinnvoller Weg. Wenn es aber wie hier um nur ein ausführbares Programm *projekt.exe* geht, genügt ein einziges Makefile auf Projektebene. Makefiles in Unterverzeichnissen werden nicht gebraucht. Beide Möglichkeiten werden in den folgenden Abschnitten demonstriert.

23.2.1 Rekursive Make-Aufrufe

Der Einfachheit halber wird hier auf die automatische Bestimmung der Abhängigkeiten und die Spiegelung der Struktur des Quellenverzeichnis im Verzeichnisbaum der Objektdateien verzichtet. Wie in [Mill] gezeigt, können rekursive Make-Aufrufe Probleme bei gegenseitigen Abhängigkeiten verursachen. Weil es hier nur um die Darstellung der Wirkungsweise rekursiver Make-Aufrufe geht, sie aber nicht immer empfehlenswert sind, erscheint die Vereinfachung gerechtfertigt. Im einfachen Fall könnte das Makefile auf Projektebene wie folgt aussehen:

```
export CXX := g++
export CXXFLAGS := -g -Wall -c
```

```
export LDFLAGS := -g
all:
→ $(MAKE) -C src/rational
→ $(MAKE) -C src/appl
clean:
→ $(MAKE) -C src/appl clean
```

Die export-Anweisung sorgt dafür, dass die Variablen in den Unterverzeichnissen bekannt sind. Die Aktionen sind nichts als der Aufruf von *make* in den jeweiligen Unterverzeichnissen. `$(MAKE)` ist das auf Projektebene aufgerufene Make-Programm – diese Art des Aufrufs garantiert, dass kein anderes Make-Programm (sofern vorhanden) in den Unterverzeichnissen aufgerufen wird. Alternativ wäre auch das Kommando `cd src/rational && $(MAKE)` usw. möglich gewesen. Weil jede $\rightarrow$ -Zeile an eine eigene Shell-Instanz weitergegeben wird, wirkt sich der Wechsel in ein anderes Verzeichnis nicht auf andere Aktionen aus. Das Ziel `clean` wird nur an das Verzeichnis weitergereicht, in dem gelinkt wird (siehe unten), weil dazu alle benötigten Objektdateien referenziert werden müssen. Diese Information kann für das Löschen verwendet werden. Es folgen die Makefiles für die Unterverzeichnisse:

Listing 23.3: Makefile für Unterverzeichnis rational

```
# cppbuch/k23/dirTreeRec/src/rational/makefile
objs := rational.o
all: $(objs)
%.o : %.cpp
→ @echo compiling $< .....
→ -$(CXX) $(CXXFLAGS) $< -o $@
```

Listing 23.4: Makefile für Unterverzeichnis appl

```
# cppbuch/k23/dirTreeRec/src/appl/makefile
objs := main.o ../rational/rational.o
exe := ../../bin/projekt.exe
all: $(exe)
%.o : %.cpp
→ @echo compiling $< .....
→ -$(CXX) $(CXXFLAGS) $< -o $@
$(exe): $(objs)
→ mkdir -p $(@D)
→ @echo linking $^
→ $(CXX) -o $@ $(objs) $(LDFLAGS)
clean:
→ $(RM) $(objs)
```

In der vorletzten Regel wird `$(@D)` verwendet, der Verzeichnisanteil (D = directory) des Ziels. Dementsprechend bezeichnet die automatische Variable `@F` den Dateinamen des Ziels ohne das Verzeichnis. Wenn die zu erzeugenden Dateien voneinander unabhängig sind, sind rekursive Aufrufe problemlos.



Ein Anwendungsbeispiel dafür finden Sie in Abschnitt [23.3.1](#).

23.2.2 Ein Makefile für alles

Wenn es, wie oben erwähnt, um nur ein ausführbares Programm, zum Beispiel *projekt.exe*, geht, genügt ein einziges Makefile auf Projektebene. Makefiles in Unterverzeichnissen sind dann überflüssig. Hier wird wieder von der auf Seite 605 abgebildeten Verzeichnisstruktur ausgegangen. Das folgende Makefile geht vom gezeigten Verzeichnisbaum aus:

Listing 23.5: Makefile mit automatischer Ermittlung der cpp-Dateien

```
# cppbuch/k23/dirTree/makefiledep.mak  noch nicht optimal, siehe unten!
SRCDIR := src
verz := $(foreach dir,$(SRCDIR),$(wildcard $(dir)/*))
cppfiles := $(foreach dir,$(verz),$(wildcard $(dir)/*.cpp))
include include.mak
```

Mit Hilfe der Funktion `foreach` werden der Variablen `verz` die unterhalb `src` liegenden Verzeichnisse zugewiesen. Mit derselben Funktion werden alle `cpp`-Dateien dieser Verzeichnisse bestimmt. Zum Schluss wird die nachfolgende Datei *include.mak* eingelesen, die die restliche Verarbeitung übernimmt:

Listing 23.6: Include-Makefile

```
# cppbuch/k23/dirTree/include.mak
CXX := g++
CXXFLAGS := -g -Wall -c
INCLUDE := -I.
LDFLAGS := -g
OBJDIR := obj
BINDIR := bin
DISTDIR := dist
EXEFILE := projekt.exe
objects := $(subst $(SRCDIR), $(OBJDIR), $(cppfiles:.cpp=.o))
deps := $(objects:.o=.d)
.PHONY: all clean dist
all: $(BINDIR)/$(EXEFILE)
    -include $(deps)
$(OBJDIR)/%.d: $(SRCDIR)/%.cpp
    -> mkdir -p $(@D)
    -> $(CXX) -MM -MT "$@ $(patsubst %.d,%.o,$@)" -MF $@ $<

$(OBJDIR)/%.o: $(SRCDIR)/%.cpp
    -> $(CXX) $(CXXFLAGS) $(INCLUDE) $< -o $@

$(BINDIR)/$(EXEFILE): $(objects)
    -> mkdir -p $(BINDIR)
    -> $(CXX) -o $@ $^ $(LDFLAGS)

$(DISTDIR)/gezippt.zip: $(BINDIR)/$(EXEFILE)
    -> mkdir -p $(DISTDIR)
    -> zip $(DISTDIR)/gezippt.zip $(BINDIR)/$(EXEFILE)

dist: $(DISTDIR)/gezippt.zip
```

```
clean:
→ $(RM) -r -f $(OBJDIR)
→ $(RM) -r -f $(BINDIR)
ultraclean:
→ $(RM) -r -f $(OBJDIR)
→ $(RM) -r -f $(DISTDIR)
→ $(RM) -r -f $(BINDIR)
```

include.mak enthält alle notwendigen Regeln. Im Vergleich zu den vorherigen Makefiles wurden die Ziele *dist* und *ultraclean* hinzugefügt. *make dist* sorgt dafür, dass die ausführbare Datei gezippt und im Verzeichnis *dist* abgelegt wird. Das letzte Ziel löscht sämtliche Verzeichnisse mit generierten Dateien. Einen kleinen »Schönheitsfehler« hat diese Lösung noch: Sie funktioniert, wenn es unterhalb von *\$(SRCDIR)* genau noch eine Ebene von Verzeichnissen mit *cpp*-Dateien gibt, nicht aber, wenn es noch tiefer verschachtelte Verzeichnisse gibt. Dazu gibt es eine Lösung im nächsten Abschnitt.

23.3 Automatische Erzeugung von Makefiles

Leider ist es nicht möglich, mit *make* eine beliebig verschachtelte Struktur unterhalb *\$(SRCDIR)* zu analysieren. Um das zu erreichen, hilft ein kleines Shell-Skript *makemakefile*, sodass nun auch dieses Problem gelöst ist:

Listing 23.7: Shell-Skript zur automatischen Makefile-Erzeugung

```
# cppbuch/k23/dirTree/makemakefile
SRCDIR=src
echo SRCDIR := $SRCDIR > makefile
echo cppfiles=\ >> makefile
find $SRCDIR -name "*.cpp" \
  | sed "s%\.cpp%\\.cpp\\\\\\\\%g" >> makefile
# Leerzeile
echo "" >> makefile
echo include include.mak >> makefile
```

Unter Unix muss das Skript mit `chmod u+x makemakefile` ausführbar gemacht werden.



Hinweis für Windows-Benutzer

Der Windows-Kommandointerpreter versteht das Skript nicht, weswegen ein kleiner Umweg gegangen werden muss. Es wird davon ausgegangen, dass MSYS und MinGW (siehe Seite 518) installiert und im Pfad sind. Zum Aufruf von *makemakefile* gehen Sie in das Verzeichnis, das *makemakefile* enthält, und tippen dort `sh makemakefile` ein. Das Shell-Programm *sh* kann mit dem Skript etwas anfangen, und das Makefile wird erzeugt.

Wenn *makemakefile* aufgerufen wird, geschieht Folgendes:

1. `SRC DIR := src` wird in die Datei *makefile* geschrieben.
2. `cppfiles=\` wird in die nächste Zeile geschrieben. Im Skript muss für den Backslash `\\` geschrieben werden.
3. Der `find`-Befehl ermittelt alle `cpp`-Dateien in allen Unterverzeichnissen, ausgehend vom Quellverzeichnis `SRC DIR`. Das Ergebnis wird dem Streameditor *sed* übergeben, der an jedes `».cpp«` (das sich am Zeilenende befindet) einen Backslash anfügt. *sed* erfordert dazu `\\`. Die Wirkung eines Backslashes am Zeilenende ist, dass die Folgezeile als dazugehörig interpretiert wird, das Ergebnis also wie eine einzige Zeile wirkt.
4. Nach Anhängen einer Leerzeile wird die Zeile `include include.mak` in das Makefile geschrieben. Das Ergebnis ist die folgende Datei *makefile*.

```
SRC DIR := src
cppfiles=\
src/rational/rational.cpp\
src/appl/main.cpp\

include include.mak
```

Die Datei *include.mak* ist dieselbe wie oben. Der Aufruf `make` würde dann alle Abhängigkeiten aller `cpp`-Dateien in einem beliebig verschachtelten Verzeichnisbaum unterhalb `$(SRC DIR)` bestimmen, alle `cpp`-Dateien übersetzen und die Ergebnisse in einem entsprechenden Verzeichnisbaum unterhalb von `$(OBJ DIR)` ablegen. Anschließend würde das ausführbare Programm *projekt.exe* im `$(BIND IR)`-Verzeichnis erzeugt werden. Ein abschließender Hinweis zu `$(SRC DIR)`: Durch den Ersetzungsmechanismus in *include.mak* bedingt darf `$(SRC DIR)` nicht leer sein oder nur auf das aktuelle Verzeichnis `(.)` verweisen.

23.3.1 Makefile für rekursive Aufrufe erzeugen

Im Verzeichnis *cppbuch* der Beispiele von der DVD finden Sie ein anderes kurzes Skript mit dem Namen *makemakefile*. Es erzeugt ein Makefile, in dem alle Makefiles der untergeordneten Verzeichnisse aufgerufen werden:

Listing 23.8: Shell-Skript zur Makefile-Erzeugung für rekursive `make`-Aufrufe

```
# cppbuch/makemakefile
rm -f makefile
echo all:> temporaer.txt
find . -name makefile\
  | sed "s%\./%\→\$(MAKE) -C %g" \
  | sed "s%/makefile%g" >> temporaer.txt
echo clean:>> temporaer.txt
find . -name makefile\
  | sed "s%\./%\→\$(MAKE) -C %g" \
  | sed "s%/makefile% clean%g" >> temporaer.txt
mv temporaer.txt makefile
echo makefile erzeugt! Aufruf: make oder make clean
```

Der `find`-Befehl ermittelt alle Dateien mit dem Namen *makefile* in allen Unterverzeichnissen. Das Ergebnis wird dem Streameditor *sed* übergeben, der die Zeichenfolge `»./%«` am Anfang jeder Zeile durch ein Tabulatorzeichen, gefolgt von `»$(MAKE) -C «`, ersetzt.

Die Zeichenfolge »`makefile`« wird gelöscht, sodass nur der Verzeichnisname bleibt. Mit dem erzeugten Makefile können mit nur einem Befehl sämtliche Beispieldateien übersetzt werden. Dabei werden die Makefiles in den jeweiligen Verzeichnissen aufgerufen. Fast alle dieser Makefiles bestehen jeweils nur aus einer Zeile, die das passende Makefile aus dem Verzeichnis `cppbuch/make` einschließt. In den beiden Makefiles in `cppbuch/make` befindet sich vor `$(CXX)` ein Minuszeichen, damit der Vorgang bei Fehlermeldungen nicht abgebrochen wird. Im Übrigen kann man `make` beschleunigen, wenn es mit der Option `-jX` aufgerufen wird (X = Grad der Parallelität, z.B. 2 oder 4). Sie bewirkt, dass die Übersetzungen *parallel* ablaufen. Um tatsächlich eine signifikante Zeiteinsparung erreichen zu können, müssen natürlich Betriebssystem und Hardware Parallelität unterstützen und die Übersetzungsvorgänge unabhängig voneinander sein. Ein Nachteil der parallelen Abarbeitung ist jedoch, dass sich die Reihenfolge von Fehler- und anderen Meldungen ändern kann.

23.4 Erzeugen von Bibliotheken

Aus Abschnitt 3.3.2 wissen Sie, dass zusammengehörige Klassen und Funktionen zu Bibliotheksmodulen zusammengefasst werden können. In diesem Abschnitt geht es darum, wie man das macht und wie ein Bibliotheksmodul, auch kurz Bibliothek oder Library genannt, benutzt wird. Um die Darstellung möglichst einfach zu halten, soll in den folgenden Beispielen nur die Klasse `Rational` in ein Bibliotheksmodul transformiert werden. Es gibt zwei Projekte oder Sichtweisen:

1. Entwicklungsprojekt zum Erzeugen der Bibliothek. Das Ergebnis ist kein ausführbares Programm, sondern ein Bibliotheksmodul, das anschließend eingebunden werden kann (siehe 2.). Es ist typisch, dass Bibliotheksmodule nach ihrer Entwicklung nur selten geändert, aber oft benutzt werden. Die Verzeichnisstruktur ist, bezogen auf unser Beispiel:

```
libprojekt          // Projektverzeichnis
|-- lib             // Verzeichnis für das Bibliotheksmodul
|-- libinclude.mak // wird von makefile inkludiert, s.u.
|-- makefile
|-- makemakefile   // von Seite 609 bekannt
'-- src
    '-- rational
        |-- rational.cpp
        '-- rational.h
```

2. Softwareprojekt, das ein oder mehrere Bibliotheksmodule nutzt. Verzeichnisstruktur, bezogen auf unser Beispiel:

```
anwendung          // Projektverzeichnis
|-- main.cpp
|-- makefile
'-- rational.h
```


rational.cpp fehlt. In *main.cpp* wird nur die Schnittstelle *rational.h* benutzt; die Implementierung der Funktionen findet sich im einzubindenden Bibliotheksmodul.

Es werden statische und dynamische Bibliotheksmodule unterschieden. Was damit gemeint ist und welche Vor- oder Nachteile damit verbunden sind, ist Inhalt der nächsten Abschnitte.

23.4.1 Statische Bibliotheksmodule

Statische Bibliotheksmodule werden, wie in Abbildung 3.9 (Seite 124) gezeigt, zu dem ausführbaren Programm gebunden (statisches Linken). Die so erzeugte Datei ist dementsprechend größer. *Vorteil:* Sie kann auf einen anderen Computer derselben Bauart und mit demselben Betriebssystemtyp kopiert werden und funktioniert dort wie auf dem Originalsystem. *Nachteil:* Wenn N Programme dieselbe statische Bibliothek benötigen, wird N mal der zugehörige Speicherplatz gebraucht, wenn die Programme gleichzeitig laufen.

Erzeugen eines statischen Bibliotheksmoduls

Ein statisches Bibliotheksmodul unterliegt in der GNU-Welt einer Namenskonvention. Der Name beginnt mit *lib*, danach folgt ein passender Name, und die Dateiergung ist *.a* (für Archiv). Das GNU-Programm *ar* erzeugt und modifiziert statische Bibliotheksmodule. Der entscheidende Auszug des für unser Beispiel geeigneten Makefiles sieht so aus:

Listing 23.9: Makefile zur Erzeugung einer statischen Bibliothek (Auszug)

```
# cppbuch/k23/staticLib/libprojekt/libinclude.mak (Auszug)
LIB := lib/librational.a

all: $(LIB)

$(LIB): $(objects)
    mkdir -p $(@D)
    ar cru $(LIB) $(objects)
```

Der Rest des Makefiles entspricht der Datei *include.mak* von Seite 608. Die Parameter *cru* bedeuten:

- c : Archiv ggf. ohne Warnung neu erzeugen (create).
- r : Ggf. vorhandene Dateien ersetzen (replace).
- u : Wie r, aber nur, wenn die Dateien neuer als die zu vorhandenen sind (update).

Einbinden eines statischen Bibliotheksmoduls

Die oben erzeugte Datei *lib/librational.a* wird eingebunden, indem sie dem Compiler beim Linken übergeben wird. Das Makefile zum Erzeugen unser schlichten Applikation:

Listing 23.10: Einbindung des Bibliotheksmoduls

```
# cppbuch/k23/staticLib/anwendung/makefile
.PHONY: clean

CXX := g++
CXXFLAGS := -c -g -Wall
LIBDIR := ../libprojekt/lib
```

```
LDFLAGS := -g -static -L$(LIBDIR) -lrational

projekt.exe: main.o
→ $(CXX) -o projekt.exe main.o $(LDFLAGS)

main.o: main.cpp rational.h
→ $(CXX) $(CXXFLAGS) main.cpp

clean:
→ rm -f projekt.exe main.o
```

Auch hier ist eine Konvention zu beachten, wie an der Variablen `$LDFLAGS` zu sehen ist: Der Name des Archivs wird mit dem Schalter `-l` übergeben, aber der Anfang des Dateinamens *lib* und die Endung *.a* werden weggelassen. Der Schalter `-L` teilt dem Linker das Verzeichnis, in dem sich die Bibliotheksdatei befindet, mit. `-static` sorgt für die statische Einbindung.

23.4.2 Dynamische Bibliotheksmodule

Dynamische Bibliotheksmodule sind *nicht* in der ausführbaren Datei enthalten, sondern werden erst beim Start des Programms dazugebunden (dynamisches Linken). *Vorteil:* Wenn beliebig viele Programme dieselbe dynamische Bibliothek benötigen, wird der zugehörige Speicherplatz nur einmal gebraucht. *Nachteil:* Die ausführbare Datei kann auf einen anderen Computer derselben Bauart und mit demselben Betriebssystemtyp kopiert werden, funktioniert dort aber nur, wenn auch die dynamische Bibliothek mitgeliefert und an einer passenden Stelle installiert wird – andernfalls wird sie beim Start des Programms nicht gefunden.

Erzeugen eines dynamischen Bibliotheksmoduls

Auch für dynamische Bibliotheksmodule gilt die Konvention, dass ein Dateiname mit *lib* beginnt und die Endung weggelassen wird. Im Unterschied zu den statischen Bibliotheksmodulen ist die Endung *.so* (shared objects) unter Linux und *.dll* (dynamic link library) unter Windows. Das für unser Beispiel geeignete Makefile berücksichtigt in der Definition der Variablen `DYNLIB` das Betriebssystem, indem die Umgebungsvariable `OS` (operating system) daraufhin geprüft wird, ob sie die Zeichenkette »Windows« enthält. Die Funktion `findstring` gibt die gesuchte Zeichenkette zurück, sofern sie vorhanden ist, andernfalls wird der leere String zurückgegeben. Bei dieser Gelegenheit lernen Sie die gleich einen Bedingungsausdruck für *make* kennen. Mit `ifeq` wird das Ergebnis von `findstring` mit »Windows« verglichen. Die Variable `DYNLIB` wird in Abhängigkeit vom Ergebnis des Vergleichs definiert. Hier folgt nun das Makefile:

Listing 23.11: Makefile mit include zur Bibliothekserzeugung

```
# cppbuch/k23/dynLib/libprojekt/makefile
SRCDIR := src
cppfiles := src/rational/rational.cpp
include libinclude.mak
```

Die mit `include` eingebundene Datei enthält die Abfrage des Betriebssystems. Die dynamische Bibliothek wird mit der Option `-shared` erzeugt (Zeile vor dem `clean`-Target).

Listing 23.12: Include-Makefile mit Erzeugung einer dynamischen Bibliothek

```
# cppbuch/k23/dynLib/libprojekt/libinclude.mak
CXX := g++
CXXFLAGS := -g -Wall -c
INCLUDE := -I.
LD_FLAGS := -g
OBJDIR := obj

ifeq "$(findstring Windows,$(OS))" "Windows"
DYNLIB := lib/librational.dll
else # Unix/Linux
DYNLIB := lib/librational.so
endif

objects := $(subst $(SRCDIR), $(OBJDIR), $(cppfiles:.cpp=.o))
deps := $(objects:.o=.d)

.PHONY: all clean dist

all: $(DYNLIB)

-include $(deps)

$(OBJDIR)/%.d: $(SRCDIR)/%.cpp
-✂ mkdir -p $(@D)
-✂ $(CXX) -MM -MG -MT "$@ $(patsubst %.d,%.o,$@)" -MF $@ $<

$(OBJDIR)/%.o: $(SRCDIR)/%.cpp
-✂ $(CXX) $(CXXFLAGS) $(INCLUDE) $< -o $@

$(DYNLIB): $(objects)
-✂ mkdir -p $(@D)
-✂ $(CXX) -shared -o $(DYNLIB) $(objects)

clean:
-✂ rm -r -f $(OBJDIR)
-✂ rm -r -f $(DYNLIB)
```

Einbinden eines dynamischen Bibliotheksmoduls

Im Gegensatz zum statischen Binden wird in das ausführbare Programm beim Linken nur ein Verweis auf die Bibliothek integriert. Das Kommando zum Linken bleibt bis auf `-static` dasselbe wie zur Einbindung eines statischen Bibliotheksmoduls, zum Beispiel:

```
g++ -o projekt.exe main.o -Llibverzeichnis -lrational
```

Der Linker erkennt an der Endung der gefundenen Datei `.so` oder `.dll`, dass nur ein Verweis einzutragen ist. Das Linux-Programm `ldd` zeigt die eingetragenen dynamischen Bibliotheken. So ergibt der Aufruf `ldd projekt.exe`:

```
linux-gate.so.1 => (0xffffe000)
librational.so => not found
libstdc++.so.6 => /usr/local/lib/libstdc++.so.6 (0xb7eb0000)
```

```
libm.so.6 => /lib/libm.so.6 (0xb7e8a000)
libgcc_s.so.1 => /usr/local/lib/libgcc_s.so.1 (0xb7e7c000)
libc.so.6 => /lib/libc.so.6 (0xb7d4d000)
/lib/ld-linux.so.2 (0xb7fc3000)
```

Man sieht, dass der Verweis `librational.so` existiert, die Bibliothek aber nicht an den voreingestellten Plätzen zu finden ist. Spätestens bei Aufruf des Programms muss der Ort der Bibliothek bekannt sein.

Programmaufruf unter Linux

Unter Linux gibt es die Konvention, dass der Pfad zu Libraries, die nicht zum System gehören, in der Shell-Variablen `LD_LIBRARY_PATH` abgelegt wird. Dazu wird, bevor `projekt.exe` aufgerufen wird, in der Shell das Kommando

```
export LD_LIBRARY_PATH=../libprojekt/lib
```

einggegeben. Andernfalls würde der Aufruf von `projekt.exe` eine Fehlermeldung ergeben. Nachdem der Pfad zur Library angegeben worden ist, kann auch `ldd` die Bibliothek lokalisieren. Oben wird angenommen, dass sich das Verzeichnis *libprojekt* auf derselben Ebene wie das Verzeichnis, in dem `projekt.exe` ausgeführt wird, befindet. Im Allgemeinen ist das jedoch nicht der Fall, weswegen die zugehörigen Bibliotheken bei der Installation eines Programms in die entsprechenden Systemverzeichnisse für Libraries, zum Beispiel `/usr/local/lib`, kopiert werden. Und wenn nicht, also `LD_LIBRARY_PATH` zur Geltung kommt, sollte nicht der relative, sondern der absolute Pfad eingetragen werden.

Programmaufruf unter Windows

Ein Aufruf des Programms `projekt.exe` unter Windows würde ohne weitere Maßnahmen auch zu einer Fehlermeldung führen. Windows sucht erst im aktuellen Verzeichnis nach der Bibliothek und danach im durch die Umgebungsvariable `PATH` definierten Pfad. Um eine DLL (dynamic link library) auffindbar zu machen, ergänzt man den Pfad durch Eingabe des Kommandos

```
PATH=%PATH%;..\libprojekt\lib
```

Oder man kopiert die DLL in eins der Verzeichnisse, die sich bereits im Pfad befinden, zum Beispiel `C:\windows\system`. Das folgende Makefile hat ein Ziel `run`, das abfragt, ob das Bibliotheksverzeichnis im Pfad liegt (Windows) bzw. den `LD_LIBRARY_PATH` entsprechend einstellt (Linux).

Listing 23.13: Makefile mit Ziel `run`

```
# cppbuch/k23/dynLib/anwendung/makefile
.PHONY: run zeigeLibs clean

CXX := g++
CXXFLAGS := -c -g -Wall
LIBDIR := ../libprojekt/lib
LDFLAGS := -g -L$(LIBDIR) -lrational
EXE := projekt.exe
```

```

$(EXE): main.o
→ $(CXX) -o $(EXE) main.o $(LDFLAGS)

main.o: main.cpp rational.h
→ $(CXX) $(CXXFLAGS) main.cpp

zeigeLibs: $(EXE)
→ ldd $(EXE)

run: $(EXE)
ifeq "$$(findstring Windows,$(OS))" "Windows"
ifeq "$$(findstring $(subst /,\, $(LIBDIR)),$(PATH))" ""
→ @echo Keine Ausfuehrung! $(LIBDIR) muss im Pfad liegen!
else
→ $(EXE)
endif
else # Unix/Linux
→ export LD_LIBRARY_PATH=$(LIBDIR); ./$(EXE)
endif
clean:
→ rm -f $(EXE) main.o

```

23.5 GNU Autotools

Es gibt große Projekte, deren ausführbare Programme auf vielen verschiedenen Computern und Betriebssystemen laufen sollen. Ein prominentes Beispiel dafür ist die GNU Compiler Collection (GCC). Die oben gezeigten Mechanismen wären für so eine komplexe Aufgabe nicht ausreichend. Es gibt dafür einen Satz von Werkzeugen, GNU Autotools genannt, die dabei helfen sollen. Besonders sind die Werkzeuge Autoconf und Automake zu nennen [GNU]. Hier wird aus folgenden Gründen nur kurz auf die Benutzung der Autotools eingegangen:

1. Die Autotools sind Standard bei GNU- und manchen anderen Open Source-Programmen. Die Kommandofolge

```
./configure
make install
```

zum Erzeugen eines Makefiles und anschließendem Übersetzen und Installieren eines Programms kennt fast jeder Linux-Benutzer. Punkt und Schrägstrich vor `configure` sind erforderlich, wenn das aktuelle Verzeichnis nicht im Pfad liegt.
2. Der Umfang der Möglichkeiten sprengt den Rahmen dieses Buchs.
3. Für viele Programme ist der in den vorherigen Abschnitten beschriebene Build-Prozess mit `make` ausreichend und komfortabel.
4. Der Einarbeitungsaufwand ist beträchtlich. Es muss die Syntax mehrerer Programme, die alle zusammenwirken, gelernt werden. Die Dokumentation ist für unerfahrene Entwickler kaum verständlich.

5. Die gewünschte Portabilität wird oft nicht oder nur mit erheblichem Anpassungsaufwand geleistet, besonders die Portierung von Linux auf Windows.

Die letzten beiden Punkte führten dazu, dass sich etliche Entwickler von den Autotools abgewendet haben und weiter abwenden. So hat sich unter anderem das Entwicklungsteam der Linux-Desktop-Oberfläche KDE für den Umstieg auf CMake entschieden [Neun] (<http://www.cmake.org/>). CMake wird im folgenden Abschnitt 23.6 kurz beschrieben. Für die Demonstration der Autotools sei die folgende Verzeichnisstruktur gegeben:

```

.                                Projektverzeichnis
|-- anwendung
|  |-- main.cpp
|-- libprojekt
|   |-- src
|       |-- rational
|           |-- rational.cpp
|           '-- rational.h

```

Ziel ist es, auf jeder Ebene geeignete Makefiles zu erzeugen:

- *Projektverzeichnis*: Das Makefile soll die Makefiles der Unterverzeichnisse aufrufen.
- *libprojekt*: Das Makefile soll eine statische Bibliothek *librational.a* erzeugen.
- *anwendung*: Das Makefile soll *main.cpp* übersetzen und dabei die Header-Datei *../libprojekt/src/rational/rational.h* einbinden. Die Objektdatei *main.o* soll mit der Bibliothek *librational.a* zur ausführbaren Datei *appl* zusammengebunden werden.

Dazu verlangen die Autotools in jedem der drei Verzeichnisse die Dateien *configure.ac*, *Makefile.am* zur Steuerung des Prozesses sowie die Dateien *AUTHORS*, *NEWS*, *Change-Log* und *README*. Diese sechs Dateien müssen geschrieben werden, wobei der Inhalt der letzten vier beliebig ist – die Dateien müssen nur existieren. Weitere verlangte Dateien wie *COPYING*, *INSTALL* und einige Skripte können mit dem Befehl `automake -a` erzeugt werden. Um nun die *configure*-Dateien zu erzeugen, müssen *zuerst* in den Unterverzeichnissen und dann im Hauptverzeichnis die Kommandos

```
autoreconf
automake -a
```

aufgerufen werden. Anschließend können im Projektverzeichnis (oder selektiv in den Unterverzeichnissen) `./configure` und `make` aufgerufen werden. Im Verzeichnis *anwendung* findet sich anschließend die ausführbare Datei *appl*. Die Steuerungsdateien *configure.ac* und *Makefile.am* sind:

```
# cppbuch/k23/autotools/Makefile.am
SUBDIRS = libprojekt anwendung

# cppbuch/k23/autotools/configure.ac
AC_INIT(anw, 1.0, info@fehler.de)
AC_CONFIG_AUX_DIR([.])
AM_INIT_AUTOMAKE
AC_PROG_CXX
AC_CONFIG_FILES([Makefile])
AC_CONFIG_SUBDIRS([libprojekt anwendung])
AC_OUTPUT
```

Dabei steht AC für autoconf und AM für automake. Es gibt außer diesen beiden noch mehr Programme bzw. Skripte, die aber indirekt aufgerufen werden und hier nicht direkt sichtbar werden. Dem Makro AC_INIT wird der Name des Projekts, die Versionsnummer und eine E-Mail-Adresse für Fehlermeldungen mitgegeben. AC_PROG_CXX ermittelt den C++-Compiler. Zu den weiteren Makros bitte ich, [\[GNU\]](#) zu konsultieren. Im Projektverzeichnis wird letztlich auf die Unterverzeichnisse verwiesen, deren entsprechende Dateien folgen.

```
# cppbuch/k23/autotools/libprojekt/Makefile.am
noinst_LIBRARIES = librational.a
librational_a_SOURCES = src/rational/rational.cpp \
                        src/rational/rational.h
```

noinst heißt, dass die zu erzeugende Bibliothek nicht installiert, also in das entsprechende Systemverzeichnis (Voreinstellung */usr/local/lib*) kopiert werden soll. Die Quellen müssen angegeben und können nicht automatisch ermittelt werden.

```
# cppbuch/k23/autotools/libprojekt/configure.ac
AC_INIT([libprojekt], [1.1])
AC_CONFIG_AUX_DIR([.])
AM_INIT_AUTOMAKE
AC_PROG_CXX
AC_PROG_RANLIB
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

AC_PROG_RANLIB sorgt für die Generierung einer Bibliotheksdatei.

```
# cppbuch/k23/autotools/anwendung/Makefile.am
AM_CXXFLAGS = -I../libprojekt/src/rational
bin_PROGRAMS = appl
appl_SOURCES = main.cpp
appl_LDADD = ../libprojekt/librational.a
```

Der Inhalt des Makros AC_CXXFLAGS wird dem Compiler übergeben, damit er *rational.h* findet. Der Name des zu erzeugenden Programms wird mit bin_PROGRAMS angegeben. Dieser Name wird dann als Präfix für die Angabe der Quelldateien und der hinzuzufügenden Library benutzt.

```
# cppbuch/k23/autotools/anwendung/configure.ac
AC_INIT(appl, 1.0, info@fehler.de)
AC_CONFIG_AUX_DIR([.])
AM_INIT_AUTOMAKE
AC_PROG_CXX
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

23.6 CMake

Wie die Autotools erzeugt CMake Makefiles, die von *make* weiterverarbeitet werden. Nach meiner eigenen Erfahrung ist der Einarbeitungsaufwand im Vergleich zu den Autotools erheblich geringer. Es muss auch nur ein Bruchteil an Dateien produziert werden, nämlich nur eine kurze Datei namens *CMakeLists.txt* für jedes Verzeichnis! Zum Vergleich seien die Dateien für dieselbe kleine Aufgabe, die für die Autotools verwendet wurde, angegeben.

```
# cppbuch/k23/cmake/CMakeLists.txt
project (anw)
add_subdirectory (libprojekt)
add_subdirectory (anwendung)
```

```
# cppbuch/k23/cmake/libprojekt/CMakeLists.txt
add_library (rational STATIC src/rational/rational.cpp)
```

```
# cppbuch/k23/cmake/anwendung/CMakeLists.txt
include_directories(..libprojekt/src/rational)
link_directories(..libprojekt)
add_executable(appl main.cpp)
target_link_libraries(appl rational)
```

Das ist schon alles. Die Makefiles werden durch Aufruf im Projektverzeichnis (oder selektiv in den Unterverzeichnissen) erzeugt, indem nur

```
cmake .
```

eingegeben wird (Punkt nicht vergessen). Das ist alles! Damit hat *cmake* gute Chancen, die umständlichen Autotools abzulösen. Als Nachteil mag gesehen werden, dass die Installation von CMake vorausgesetzt wird, wenn auf einem System das Makefile neu konfiguriert werden soll, während eine von den Autotools erzeugte Distribution, die üblicherweise *configure* enthält, auf einem anderen System die Autotools nicht benötigt. Dieser minimale Nachteil wird verschwinden, wenn CMake es schafft, die Autotools abzulösen.

23.7 Code Bloat bei der Instanziierung von Templates vermeiden

In Projekten mit einer großen Anzahl von Dateien gibt es bei der Instanziierung von Templates das Phänomen der Aufblähung der Objektdateien (englisch *code bloat*). Zur Erklärung wird die Existenz der folgenden Template-Klasse angenommen:

Listing 23.14: Beispiel-Template

```

#ifndef MEINTYP_T
#define MEINTYP_T

template<typename T>
class MeinTyp {
public:
    MeinTyp(T par) {
        // umfangreicher Code für den Konstruktor
    }

    T get() {
        // umfangreicher Code für get()
    }
private:
    // Attribute weggelassen
};
#endif

```

Das Anwendungsprogramm bestehe aus 501 Dateien *main.cpp*, *prog001.cpp*, *prog002.cpp* usw. bis *prog500.cpp*. Es sei ferner angenommen, dass in jeder der *prog*-Dateien Objekte der Klassen *MeinTyp<string>* und *MeinTyp<double>* verwendet werden und deswegen jedesmal die Datei *meintyp.t* mit `#include` eingeschlossen wird. Dieses Compilationsmodell heißt deswegen auch Inklusions-Modell. Die Übersetzung aller *cpp*-Dateien liefert *main.o*, *prog001.o*, *prog002.o* usw. bis *prog500.o*. Diese Dateien enthalten alle den vollständigen Objektcode der Templateklasse, obwohl er in allen Dateien identisch ist (abgesehen vom Unterschied *string/double*)! Damit dauert erstens die Compilation relativ lange, weil der Objektcode jedesmal neu erzeugt wird, und zweitens wird eine Menge an Massenspeicherplatz verschwendet. Bei großen Projekten kann eine vollständige Compilation durchaus mehrere Stunden dauern. Der Linker hat die Aufgabe, die 498 überflüssigen Duplikate zu entfernen. Mit wenig Aufwand lässt sich das Verhalten entscheidend verbessern. Dazu gibt es einige Wege, die ein ungefähr gleich gutes Ergebnis liefern:

1. Verhindern der überflüssigen Objektcode-Erzeugung für Templates mit der Anweisung `extern template`.
2. Aufspaltung des Templates in Schnittstelle und Implementation

Die Methoden werden in den folgenden Abschnitten am Beispiel gezeigt. Die zweite Methode hat den Vorteil, dass der Compiler nicht jedesmal die ganze Implementation des Templates lesen muss, sondern nur genau einmal. Dafür hat sie den Nachteil, dass sie nur für selbst zu schreibende Templates anwendbar ist – die vorgegebenen Templates etwa der C++-Standardbibliothek können nicht verändert werden.

23.7.1 extern-Template

Eine Deklaration mit dem Schlüsselwort `extern` hat in Abschnitt 3.3.3 die Bedeutung, dass etwas benutzt werden kann, das anderweitig definiert ist. Hier wird es in einem ähnlichen Sinne verwendet. Zunächst sei das Hauptprogramm vorgestellt:

Listing 23.15: Hauptprogramm

```
// cppbuch/k23/templateInst/extern/main.cpp
#include "func01.h"
#include "func02.h"

int main() {
    func01(); // benutzt Template
    func02(); // benutzt Template
}
```

Im diesem Beispiel wird nicht von 500, sondern nur von zwei Dateien ausgegangen, die jeweils verschiedene Instanziierungen des Templates nutzen:

Listing 23.16: Erste Funktion

```
// cppbuch/k23/templateInst/extern/func01.cpp
#include "func01.h"
#include "meintyp.t"
#include <iostream>
#include <string>

// verhindert Instanziierung:
extern template class Meintyp<std::string>;

void func01() {
    Meintyp<std::string> mt("func01");
    std::cout << mt.get() << std::endl;
}
```

Listing 23.17: Zweite Funktion

```
// cppbuch/k23/templateInst/extern/func02.cpp
#include "func02.h"
#include "meintyp.t"
#include <iostream>
// verhindert Instanziierung:
extern template class Meintyp<double>;

void func02() {
    Meintyp<double> mt(3.1415926);
    std::cout << mt.get() << std::endl;
}
```

Auf die Darstellung der Header-Dateien wird verzichtet. Natürlich müssen die Templates wenigstens an einer Stelle instanziiert werden. Man fügt dem Projekt eine Datei, in der die Templates explizit instanziiert werden, hinzu:

Listing 23.18: Datei zur Instanziierung von Templates für zwei Datentypen

```
// cppbuch/k23/templateInst/extern/instanzen.cpp
#include "meintyp.t"
#include <string>
template class Meintyp<std::string>; // explizite Instanziierung
template class Meintyp<double>; // explizite Instanziierung
```

23.7.2 Aufspaltung in Schnittstelle und Implementation

Der Weg führt über die Aufspaltung der Datei *meintyp.t* in eine gleichnamige Datei, die ausschließlich die Prototypen enthält, und eine Datei *meintypImpl.t* mit der Implementierung:

Listing 23.19: Template-Prototypen

```
// cppbuch/k23/templateInst/trennIFimpl/meintyp.t
#ifndef MEINTYP_T
#define MEINTYP_T

template<typename T>
class MeinTyp {
public:
    MeinTyp(T par); // Prototyp
    T get();        // Prototyp
private:
    // Attribute weggelassen
};
#endif
```

Listing 23.20: Template-Implementierung

```
// Implementierungsdatei cppbuch/k23/templateInst/trennIFimpl/meintypImpl.t
#include "meintyp.t"

template<typename T>
MeinTyp<T>::MeinTyp(T par) {
    // umfangreicher Code
}

template<typename T>
T MeinTyp<T>::get() {
    // umfangreicher Code
}
```

In den Dateien *func01.cpp* und *func02.cpp* kann die `extern`-Anweisung entfallen. Es fehlt nur die Datei, die die zu instantizierenden Typen enthält. Sie heie ebenfalls *instanzen.cpp*, und *nur sie* bentigt die Implementierungsdatei. Die Datei *instanzen.cpp* wird dem Projekt hinzugefgt. Da sich nur in ihr der eigentliche Programmcode fr das Template befindet, werden alle anderen `cpp`-Dateien erheblich schneller compiliert, der Platzbedarf fr die Objektdateien schrumpft, und das Linken geht schneller.

Listing 23.21: Datei zur Template-Instanziierung

```
// cppbuch/k23/templateInst/trennIFimpl/instanzen.cpp
#include "meintypImpl.t" // Implementierung einlesen (nicht meintyp.t!)
#include <string>
template class MeinTyp<std::string>; // explizite Instanziierung
template class MeinTyp<double>;    // explizite Instanziierung
```

Wer mehr ber die Template-Instanziierung (und berhaupt mehr ber Templates) wissen mchte, dem sei [\[VaJo\]](#) empfohlen.

24

Algorithmen für verschiedene Aufgaben

Dieses Kapitel behandelt die folgenden Themen:

- Algorithmen für verschiedene Zwecke
- Auf Folgen arbeitende Algorithmen
- Komplexe Zahlen in C++
- Zufallszahlen
- Grenzwerte von Zahltypen ermitteln
- 2-dimensionale Matrix mit zusammenhängendem Speicher

Für viele Probleme gibt es bereits fertige Lösungen. Dieses Kapitel hat seinen Schwerpunkt in den Lösungen, die von der C++-Standardbibliothek angeboten werden. Um die Anwendung zu erleichtern, gibt es zu fast jedem Rezept ein konkretes Beispiel.



Hinweis

Wenn nicht anders angegeben, gilt für die Algorithmen dieses Kapitels der Header `<algorithm>`.

Das Verständnis mancher Tipps setzt die Kenntnis der Wirkungsweise der STL voraus (Kapitel 11, 28, 29 und 30). Manche Algorithmen verwenden möglicherweise Ihnen unbekannte Funktionen. Beschreibungen dazu können Sie mithilfe des Registers finden.

24.1 Algorithmen mit Strings



Hinweis

Eine Menge Algorithmen, die mit Objekten oder Parametern der Klasse `string` arbeiten, finden Sie in Kapitel 32. Abschnitt 31.2 verhilft bei Bedarf zu besserem Verständnis der sprachabhängigen Funktionen.

24.1.1 String splitten

Auf Seite 643 wird ein C-String zerlegt. Der C-String wird dabei modifiziert. In diesem Abschnitt wird etwas Ähnliches unternommen, aber für echte `string`-Objekte und ohne deren Zerstörung. In Java gibt es eine nützliche Funktion `split(regex)`, die ein `String`-Array zurückgibt. `regex` ist ein regulärer Ausdruck mit den Trennzeichen. An dieser Funktion habe ich mich orientiert, wobei an die Stelle von `regex` ein `String` mit den Trennzeichen tritt. Die C++-Entsprechung des Java-Aufrufs `String[] arr = text.split("[ ,.])`; ist `split(text, " ,.", v)`, wobei `v` der Vektor ist, in dem das Ergebnis abgelegt werden soll. Falls das Trennzeichen nicht vorkommt, enthält der Vektor im einzigen Element den ganzen `String`.

Listing 24.1: String splitten

```
// cppbuch/k24/strings/split.cpp
#include<iostream>
#include<string>
#include<vector>
#include<showSequence.h>

void split(const std::string& s, const std::string& trenn,
           std::vector<std::string>& v) {
    size_t start = 0;
    size_t pos;
    do {
        pos = s.find_first_of(trenn, start);
        v.push_back(s.substr(start, pos-start));
        start = pos+1;
    } while(pos != std::string::npos);
}

using namespace std;

int main() {
    string text("The quick brown fox jumps over the lazy dog's back.");
    vector<string> v;
    split(text, " ,.", v);
    showSequence(v, "", " *\n");
}
```

Einfacher geht es, wenn die Boost-Library installiert ist:

Listing 24.2: String splitten mit Boost

```
// cppbuch/k24/strings/boostsplit.cpp
#include<iostream>
#include<string>
#include<vector>
#include<boost/algorithm/string/split.hpp>
#include<boost/algorithm/string/classification.hpp> // is_any_of
#include<showSequence.h>

using namespace std;
using namespace boost::algorithm;

int main() {
    string text("The quick brown fox jumps over the lazy dog's back.");
    vector<string> v;
    split(v, text, is_any_of(".,"));
    showSequence(v, "", " *\n");
}
```

24.1.2 String in Zahl umwandeln

Für die häufige Aufgabe, einen String in eine Zahl umwandeln zu müssen, gibt es verschiedene Lösungsansätze, die im Folgenden beschrieben werden.

- Funktionen der Standardbibliothek, die mit string-Parametern arbeiten.
- Den String in einen Stream schreiben und als Zahl wieder auslesen.
- Funktion `lexical_cast` der Boost-Library.
- Funktionen der C-Library.
- Eigene Umwandlungsfunktion.

string-Funktionen der Standardbibliothek

Die Funktionsnamen fangen mit `stoi` (für *string to*) an, gefolgt von einem oder zwei Buchstaben für den Typ (`i` für `int`, `ul` für `unsigned long` usw.). Eine Aufzählung finden Sie auf Seite 847; hier seien exemplarisch nur die Umwandlungen in eine `double`- und eine `int`-Zahl gezeigt. Die Schnittstelle für `stoi()`:

```
int stoi(const string& str, // der umzuwandelnde String
        size_t* endeptr = 0, // dort wird die erste nicht mehr ausgewertete Position
                                // hinterlegt
        int base = 10);    // Basis der Umwandlung
```

Die Funktionen liefern eine Exception, wenn eine Konversion nicht möglich oder die Zahl außerhalb des Bereichs für den Typ ist. So ist zum Beispiel die Zeichenkette `FF` nur als Hexadezimal interpretierbar, und `FFFFFFFF` überschreitet den Bereich einer 32-Bit-`int`-Zahl. Das folgende Programm zeigt die Möglichkeiten:

Listing 24.3: String mit std-Funktionen in Zahl umwandeln

```
// cppbuch/k24/strings/string2zahl/std.cpp
#include<iostream>
#include<string>
```

```

#include<stdexcept>
using namespace std;

int main() {
    cout << "Bitte Zeichenkette eingeben (z.B. 3.14 12345 0x78 FF 0123): ";
    string zeile;
    getline(cin, zeile);
    size_t endpos; // erste nicht mehr ausgewertete Position
    int modus[] = {-1, 0, 2, 8, 10, 16};
    string modusText[] = {"double",           // -1
                          "Standard (0 okt/Ox hex/dez)", // 0
                          "Binärzahl",        // 2
                          "Oktalzahl",        // 8
                          "Dezimalzahl",      // 10
                          "Hexadezimalzahl"};  // 16

    size_t anzahl = sizeof(modus)/sizeof(modus[0]);
    for(size_t i = 0; i < anzahl; ++i) {
        try {
            cout << "Interpretation als " << modusText[i] << ": ";
            if(-1 == modus[i]) { // double
                cout << stod(zeile, &endpos);
            }
            else { // int
                cout << stoi(zeile, &endpos, modus[i]);
            }
            if(endpos < zeile.length()) {
                cout << " nicht ausgewertet: " << (zeile.c_str() + endpos);
            }
            cout << endl;
        } catch(const invalid_argument&) {
            cerr << "Konversion ist nicht möglich!" << endl;
        } catch(const out_of_range&) {
            cerr << "Zahl ist außerhalb des Bereichs für diesen Typ!" << endl;
        }
    }
}

```

Umwandlung mit stringstream

Diese Variante nutzt aus, dass in einen stringstream wie nach cout geschrieben werden kann. Anschließend wird er wie ein Eingabe-Stream (wie cin) verwendet. Bei nicht konvertierbaren Strings gibt es keine Exception.

Listing 24.4: String mit stringstream in Zahl umwandeln

```

// cppbuch/k24/strings/string2zahl/sstream.cpp
#include<iostream>
#include<string>
#include<sstream>
using namespace std;

int main() {
    cout << "Bitte Zeichenkette eingeben (z.B. 3.14 12345) : ";

```

```

string zeile;
getline(cin, zeile);
stringstream wandler;
wandler << zeile; // String schreiben
double d;
wandler >> d;      // als double-Zahl lesen
cout << "Zahl =" << d << endl;
}

```

Umwandlung mit Boost-Funktion

Wenn Sie Boost installiert haben, empfehle ich die Funktion `lexical_cast<T>(arg)`. Sie wandelt den Parameter `arg` in den gewünschten Typ `T` um. Bei Konvertierungsfehlern wird eine Exception geworfen. Auch wird erwartet, dass ein String vollständig umwandelbar ist – einen Zeiger auf den nicht konvertierten Rest gibt es nicht.

Listing 24.5: String mit `lexical_cast` in Zahl umwandeln

```

// cppbuch/k24/strings/string2zahl/boost.cpp
#include<iostream>
#include<string>
#include<boost/lexical_cast.hpp>
using namespace std;

int main() {
    cout << "Bitte Zeichenkette eingeben (z.B. 3.14 12345) : ";
    string zeile;
    getline(cin, zeile);
    try {
        double d = boost::lexical_cast<double>(zeile);
        cout << "Zahl =" << d << endl;
    }
    catch(boost::bad_lexical_cast & blc) {
        cerr << blc.what() << endl;
    }
}

```

C-Funktionen

Die folgenden Funktionen setzen das Einbinden des Headers `<cstdlib>` voraus. Sie stammen aus der Sprache C und sind Grundlage der Implementierung der oben beschriebenen Funktionen, die mit `string`-Objekten arbeiten (`stoi()` usw.).

- `long strtol(const char* s, char* rest, int basis)` wandelt den C-String `s` in eine `long`-Zahl um und gibt diese zurück. Falls `s` nach der Zahl noch weitere, nicht konvertierbare Zeichen enthält, werden sie in `rest` abgelegt, sofern `rest` ungleich `NULL` ist. `basis` ist die Basis.
- `unsigned long strtoul(const char* s, char* rest, int basis)` macht dasselbe für `unsigned long`-Zahlen.
- `double strtod(const char* s, char* rest)` leistet Entsprechendes für `double`-Zahlen.

- `atoi(const char* s)` entspricht `(int) strol(s, NULL, 10)`.
- `atol(const char* s)` entspricht `strol(s, NULL, 10)`.
- `atof(const char* s)` entspricht `strtod(s, NULL)`.

Die Funktionen können für ein `string`-Objekt `einString` mit Hilfe der Elementfunktion `c_str()` genutzt werden, also etwa `strol(einString.c_str(), NULL, 10)`. Im Fehlerfall wird 0 zurückgegeben. Für die Angabe der Basis gibt es einige Regeln:

- **Basis 0:** Die Basis wird entsprechend den Konventionen der Programmiersprache ermittelt: Die Zahl 0777 wird als Oktalzahl, die Zahl 0x1abc als Hexadezimalzahl interpretiert. Alle anderen Zahlen werden als Dezimalzahlen betrachtet.
- **Basis 8 :** Die Zahl wird in jedem Fall als Oktalzahl interpretiert. 0777 und 777 ergeben dasselbe.
- **Basis 16 :** Die Zahl wird in jedem Fall als Hexadezimalzahl interpretiert. 0x1abc und 1abc ergeben dasselbe.

Für die Basis 8 wird 1abc daher 1 ergeben, mit abc als Rest. Natürlich sind auch andere Basen möglich. Die Eingabe 1000 mit der Basis 2 führt zum Ergebnis 8. Das folgende Programm zeigt die verschiedenen Interpretationen für einen eingegebenen String an:

Listing 24.6: String mit C-Funktionen in Zahl umwandeln

```
// cppbuch/k24/strings/string2zahl/cstdlib.cpp
#include<iostream>
#include<cstdlib>
using namespace std;

int main() {
    cout << "Bitte Zeichenkette eingeben (z.B. 3.14 12345 0x78 FF 0123): ";
    char eingabe[80];
    cin.get(eingabe, sizeof(eingabe));
    char* rest;
    int modus[] = {-1, 0, 2, 8, 10, 16};
    string modusText[] = {"double", "Standard (0 okt/0x hex/dez)",
                          "Binärzahl", "Oktalzahl", "Dezimalzahl",
                          "Hexadezimalzahl"};
    for(size_t i = 0; i < sizeof(modus)/sizeof(modus[0]); ++i) {
        cout << "Interpretation als " << modusText[i] << ": ";
        if(-1 == modus[i]) { // double
            cout << strtod(eingabe, &rest);
        }
        else { // long
            cout << strtol(eingabe, &rest, modus[i]);
        }
        if(*rest) {
            cout << " nicht ausgewertet:" << rest;
        }
        cout << endl;
    }
}
```

Eigene Funktion

Da es die anderen Funktionen gibt, ist eine eigene Funktion nicht notwendig. Die Funktion `s2i()` soll daher nur zeigen, wie etwa `stoi()` intern funktionieren könnte, insbesondere die Kontrolle des Überlaufs. Zur Vereinfachung wird von einer Dezimalzahl ausgegangen und die Information über den nicht ausgewerteten Rest wird nicht übergeben. In der Datei `cppbuch/k24/strings/string2zahl/s2i.cpp` finden Sie die Funktion mit einem Anwendungsbeispiel.

24.1.3 Zahl in String umwandeln

Ähnlich wie oben gibt es einige Varianten. Zwar ist die Richtung der Umwandlung umgekehrt, weil oben Strings in Zahlen konvertiert werden, aber es gibt strukturelle Ähnlichkeiten. Aus diesem Grund fällt die Darstellung etwas kürzer aus. Beispiele finden Sie im Verzeichnis `cppbuch/k24/strings/zahl2string/`.

- Funktionen `to_string(zahl)` der Standardbibliothek. `zahl` steht dabei für einen Parameter eines beliebigen Zahlentyps.
- Die Zahl in einen Stream schreiben und als String wieder auslesen, ähnlich wie oben (Seite 626). Das Listing auf Seite 393 zeigt ein ausführliches Beispiel, wie Zahlen formatiert in einem String abgelegt werden können.
- Funktion `lexical_cast` der Boost-Library, zum Beispiel

```
string ergebnis = boost::lexical_cast<string>(zahl);
```

- Eigene Umwandlungsfunktion. Wenn auf eine Formatierung verzichtet werden soll, kann für `int`-Zahlen die einfache Funktion `string i2string(int)` genommen werden, die Sie in `cppbuch/k24/strings/zahl2string/i2s.cpp` finden und die auf Seite 498 abgedruckt ist.

24.1.4 Strings sprachlich richtig sortieren

Mit der Standard-Sprachumgebung »C« kommen entsprechend der ASCII-Tabelle erst die Großbuchstaben, dann die Kleinbuchstaben, dann alle Werte danach bei 8-Bit-Zeichen. Bei einer ISO-8859-1-Codierung würden Worte, die mit Umlauten beginnen, bei dieser Sortierung erst nach den Kleinbuchstaben kommen. Im Deutschen gibt es für die Sortierung andere Regeln:

- Sie erfolgt unabhängig von der Groß- bzw. Kleinschreibung eines Worts.
- Bei sonst gleichen Wörtern kommen Kleinbuchstaben vor den Großbuchstaben.
- Umlaute wie ä, ö, ü und die zugehörigen Vokale wie a, o, u sind gleichrangig.
- Der Buchstabe ß wird wie ss einsortiert.
- Falls es zwei Wörter gibt, die sich nur dadurch unterscheiden, dass der Buchstabe ß durch ss ersetzt wurde, wird das Wort mit ss zuerst eingeordnet.

Ein Vergleich zweier Strings mithilfe der ASCII-Tabelle würde fehlerhafte Ergebnisse liefern, weil zum Beispiel »ü« im ASCII nicht definiert ist. In C++ kann die Sortierung korrekt durchgeführt werden, wenn die passende Sprachumgebung mitgegeben wird. Das Beispiel zeigt, wie ein `locale`-Objekt zur sprachlich korrekten Sortierung eingesetzt wird:

Listing 24.7: Sprachlich richtige Sortierung bei ISO8859-Codierung

```
// cppbuch/k24/strings/richtigsortierenISO8859.cpp
// Dieses Datei ist ISO 8859-1 codiert.
#include<algorithm>
#include<iostream>
#include<vector>
#include<showSequence.h> // siehe Seite 648
using namespace std;

int main() {
    vector<string> v;
    v.push_back("Maße");
    v.push_back("Masse");
    v.push_back("Ähnlich");
    v.push_back("Alphabet");
    v.push_back("aal");
    v.push_back("ähnlich");
    v.push_back("alphabet");
    v.push_back("Aal");
    cout << "vorher:";
    showSequence(v);
    sort(v.begin(), v.end());
    cout << "nach Sortieren ohne Compare-Objekt:\n";
    showSequence(v);
    locale deutsch("de_DE");
    sort(v.begin(), v.end(), deutsch);
    cout << "nach Sortieren (locale de_DE):\n";
    showSequence(v);
}
```

Im Falle von Multi-Byte-Sequenzen wie bei der UTF-8-Codierung ist `string` nicht geeignet und man muss mit `wstring` arbeiten:

Listing 24.8: Sprachlich richtige Sortierung bei UTF-8-Codierung

```
// cppbuch/k24/strings/richtigsortieren.cpp
// Diese Datei ist UTF-8 codiert
#include<algorithm>
#include<iostream>
#include<vector>
#include<printWstringVector.h> // zeigt einen wstring-Vektor an
#include<string>

using namespace std;

int main() {
    locale::global(locale("de_DE.utf-8")); // deDE global setzen
    vector<wstring> v;
    v.push_back(L"Maße");
    v.push_back(L"Masse");
```

```

v.push_back(L"Ähnlich");
v.push_back(L"Alphabet");
v.push_back(L"aal");
v.push_back(L"ähnlich");
v.push_back(L"alphabet");
v.push_back(L"Aal");
wcout << "vorher:";
printWstringVector(v);
sort(v.begin(), v.end());
wcout << "nach Sortieren ohne Compare-Objekt:\n";
printWstringVector(v);
locale deutsch("de_DE");
sort(v.begin(), v.end(), deutsch);
wcout << "nach Sortieren (locale de_DE):\n";
printWstringVector(v);
}

```

24.1.5 Umwandlung in Klein- bzw. Großschreibung

Wie im vorherigen Abschnitt gibt es das Problem, dass die C-Standardfunktionen `toupper()` und `tolower()` Umlaute nicht korrekt umwandeln – nämlich dann, wenn die Sprachumgebung nicht richtig eingestellt ist. Für eine ISO-8859-1-Umgebung bieten sich zwei einfache Funktionen an, die mitsamt ihrer Anwendung gezeigt werden:

Listing 24.9: Umwandlung von Strings in Klein- bzw. Großschreibung (ISO8859-1-Codierung)

```

// cppbuch/k24/strings/kleingrosseinfachISO8859.cpp
#include<iostream>
#include<cstring>
#include<locale>
#include<string>

void kleinEinfach(std::string& str) {
    for(size_t i = 0; i < str.length(); ++i) {
        str[i] = tolower(str[i]); // benutzt eingestellte Locale
    }
}

void grossEinfach(std::string& str) {
    for(size_t i = 0; i < str.length(); ++i) {
        str[i] = toupper(str[i]); // benutzt eingestellte Locale
    }
}

using namespace std;

int main() {
    locale::global(locale("de_DE")); // de_DE global setzen
    string text("Falsches Üben von Xylophonmusik quält jeden größeren Zwerg.");
    cout << "anfangs:" << text << endl;
    grossEinfach(text);
}

```

```

    cout << "grossEinfach: " << text << endl;
    kleinEinfach(text);
    cout << "kleinEinfach: " << text << endl;
}

```

In einer UTF-8-Umgebung funktioniert das Programm nicht richtig, erkennbar an der falschen Ausgabe der Umlaute. Die Lösung dieses Problems folgt sofort: Falls sowohl string als auch wstring-Objekte bearbeitet werden sollen, sind Template-Funktionen sinnvoll. Um nicht Spezialisierungen mit toupper(), tolower(), towupper() towlower() schreiben zu müssen, wird auf die Funktionen von Seite 831 zurückgegriffen. Die Funktionen klein() und gross() werden in einer Template-Datei gekapselt:

Listing 24.10: Templates zu String-Umwandlung

```

// Auszug aus cppbuch/k24/strings/localeutils.t
#include<string>
#include<locale>

template<typename charT>
void klein(std::basic_string<charT>& s,
          const std::locale& loc = std::locale()) {
    for(size_t i = 0; i < s.length(); ++i) {
        s[i] = std::use_facet<std::ctype<charT> >(loc).tolower(s[i]);
    }
}

template<typename charT>
void gross(std::basic_string<charT>& s,
           const std::locale& loc = std::locale()) {
    for(size_t i = 0; i < s.length(); ++i) {
        s[i] = std::use_facet<std::ctype<charT> >(loc).toupper(s[i]);
    }
}

```

Wenn kein locale-Objekt angegeben wird, ist die global eingestellte Sprachumgebung zuständig. Die Anwendung für 1-Byte-Zeichen ist ähnlich einfach wie oben:

Listing 24.11: de_DE-String-Umwandlung (1-Byte-Zeichen)

```

// cppbuch/k24/strings/kleingrossISO8859.cpp
#include<iostream>
#include"localeutils.t"
using namespace std;
int main() {
    locale::global(locale("de_DE")); // de_DE global setzen
    string text("Falsches Üben von Xylophonmusik quält jeden größeren Zwerg.");
    cout << "anfangs:" << text << endl;
    gross(text);
    cout << "gross: " << text << endl;
    klein(text);
    cout << "klein: " << text << endl;
}

```

Im folgenden Beispiel wird eine UTF-8-Locale in Verbindung mit einem `wchar_t`-String (für Multibyte-Zeichen) verwendet:

Listing 24.12: wstring-Umwandlung

```
// cppbuch/k24/strings/kleingrossUTF8.cpp
// Diese Datei ist UTF-8 codiert!
#include<iostream>
#include"localeutils.t"
using namespace std;
// ACHTUNG: nur wcout verwenden!
// Bei GNU C++ 4.5 ändert die Benutzung von cout die Einstellungen für wcout
int main() {
    locale::global(locale("de_DE.utf-8")); // deDE global setzen
    wstring text(L"Falsches Üben von Xylophonmusik quält jeden größeren Zwerg.");
    wcout << "anfangs:" << text << endl;
    gross(text);
    wcout << "gross: " << text << endl;
    klein(text);
    wcout << "klein: " << text << endl;
}
```



Mehr über Zeichensätze und -codierung lesen Sie in Abschnitt 31.2.

24.1.6 Strings sprachlich richtig vergleichen

Das Problem der Sortierung, die ja einen Vergleich beinhaltet, wird schon in Abschnitt 24.1.4 auf Seite 629 gelöst. Hier bleibt deshalb nur, den sprachlich richtigen Vergleich zweier Strings zu gestalten. Dazu baut man sich einen Funktor, der die `locale`-Einstellung auswertet:

Listing 24.13: Funktor zum String-Vergleich

```
// Auszug aus cppbuch/k24/strings/localeutils.t
template<typename charT>
struct Stringvergleich {
    Stringvergleich(const std::locale& lo = std::locale())
        : loc(lo) {
    }
    bool operator()(const std::basic_string<charT>& s1,
                    const std::basic_string<charT>& s2) {
        return std::use_facet<std::collate<charT>>(loc)
            .compare(s1.c_str(), s1.c_str() + s1.length(),
                    s2.c_str(), s2.c_str() + s2.length()) < 0;
    }
    const std::locale loc;
};
```

Das kleine Beispielprogramm zeigt, wie der Funktor zum Vergleich zweier `char`-Strings verwendet wird:

Listing 24.14: String-Vergleich

```
// cppbuch/k24/strings/stringvergleich.cpp
#include<iostream>
#include"localeutils.t"
using namespace std;

int main() {
    locale::global(locale("de_DE")); // de_DE global setzen
    Stringvergleich<char> sv;
    string s1("ähnlich");
    string s2("bildschön");
    if(sv(s1, s2)) {
        cout << s1 << " kommt vor " << s2 << endl;
    }
    else {
        cout << s2 << " kommt vor " << s1 << endl;
    }
}
```

Zum Vergleich zweier `wchar_t`-Strings (`wstrings`) kann ein `Stringvergleich<wchar_t>`-Funktioner genommen werden, siehe `cppbuch/k24/strings/wstringvergleich.cpp`

24.1.7 Von der Groß-/Kleinschreibung unabhängiger Zeichenvergleich

In Abschnitt 24.1.5 wird die Umwandlung von Strings in Klein- bzw. Großschreibung behandelt. Hier geht es nur noch um den Vergleich einzelner Zeichen. Für Nicht-ASCII-Zeichen ist auch hier die Sprachumgebung entscheidend. Um für `char` und `wchar_t`-Zeichen gewappnet zu sein, wird eine Template-Funktion genommen. Das `locale`-Objekt kann übergeben werden; wenn nicht, wird die Voreinstellung angewendet:

Listing 24.15: Zeichenvergleich ohne Berücksichtigung der Groß-/Kleinschreibung

```
// Auszug aus cppbuch/k24/strings/localeutils.t
template<typename charT>
bool equalIgnoreCase(charT c1, charT c2,
                    const std::locale& loc = std::locale()) {
    return std::use_facet<std::ctype<charT>> >(loc).toupper(c1)
        == std::use_facet<std::ctype<charT>> >(loc).toupper(c2);
}
```

Die Benutzung der Funktion zeigt dieses kleine Beispiel:

Listing 24.16: `equalIgnoreCase()` in Aktion

```
// cppbuch/k24/strings/zeichenvergleich.cpp
#include<iostream>
#include"localeutils.t"
using namespace std;

int main() {
    locale::global(locale("de_DE")); // de_DE global setzen
    string s1("ähnlich_x"); // 1-char-Zeichen, hier ISO8859-1 codiert
    string s2("ÄHNLICH_Y");
```

```

for(size_t i = 0; i < s1.length() && i < s2.length(); ++i) {
    cout << "Die Zeichen Nr. " << i << " ("
        << s1[i]<< ", " << s2[i] << ") werden als ";
    if(!equalIgnoreCase(s1[i], s2[i])) {
        cout << "un";
    }
    cout << "gleich gewertet." << endl;
}

```



Einzelheiten zur Funktion `equalIgnoreCase()` lesen Sie in Abschnitt 31.4.2.

24.1.8 Von der Groß-/Kleinschreibung unabhängige Suche

Wenn in einem String unabhängig von der Groß- bzw. Kleinschreibung gesucht werden soll, gibt es zwei verschiedene Herangehensweisen:

- Die beteiligten Strings werden umgewandelt, sodass sie großgeschrieben (bzw. kleingeschrieben) sind.
Vorteil: Einfach. Die Mitgliedsfunktion `find()` der Klasse `string` kann genutzt werden.
Nachteil: Es werden temporäre String-Objekte erzeugt.
- Es werden Algorithmen der Standardbibliothek benutzt oder eigene Funktionen, denen jeweils ein Vergleichsobjekt mitgegeben wird.
Vorteil: Es werden keine temporären String-Objekte erzeugt.
Nachteil: Geringfügig komplizierter.

Wegen der Vermeidung temporärer Objekte wird nur der zweite Fall betrachtet. Es wird ein Funktor `EqualIgnoreCase` benutzt, dessen Name sich nur durch die Großschreibung von der obigen Funktion `equalIgnoreCase()` unterscheidet und der diese Funktion intern benutzt:

Listing 24.17: Funktor für Zeichenvergleich und Suchfunktion

```

// Auszug aus cppbuch/k24/strings/localeutils.t
template<typename charT>
struct EqualIgnoreCase { // Funktor
    EqualIgnoreCase(const std::locale& lo = std::locale())
        : loc(lo) {
    }
    bool operator()(charT c1, charT c2) {
        return equalIgnoreCase(c1, c2, loc);
    }
    const std::locale loc;
};

// Suchfunktion
template<typename charT>
size_t findeSuchstringIgnoreCase(const std::basic_string<charT>& text,
    const std::basic_string<charT>& such,
    const std::locale& loc = std::locale("de_DE")) {
    auto pos = std::search(text.begin(), text.end(),
        such.begin(), such.end(),
        EqualIgnoreCase<charT>(loc)); // Funktor
}

```



```

    return pos ==
        text.end() ? std::basic_string<charT>::npos : (pos-text.begin());
}

```

`search()` ist ein Algorithmus der Standardbibliothek (siehe Seite 677). Die Anwendung ist einfach:

Listing 24.18: Beispiel zur Substring-Suche

```

// cppbuch/k24/strings/suchelgnorecase.cpp (UTF-8 codiert)
#include<iostream>
#include"localeutils.t"

using namespace std;

int main() {
    locale::global(locale("de_DE.utf-8")); // deDE global setzen
    wstring text(L"Quer über den großen Sylter Deich");
    wstring suchstring(L"Über");
    size_t pos = findeSuchstringIgnoreCase(text, suchstring);
    if(pos != wstring::npos) {
        wcout << suchstring << " ist ab Position " << pos
            << " in " << text
            << L" enthalten (Groß-/Kleinschreibung ignoriert)." << endl;
    }
    else {
        wcout << suchstring << " ist nicht in " << text
            << " enthalten." << endl;
    }
}

```

Bei einer Codierung, die nur ein Byte pro Zeichen benötigt, wie etwa ISO8859-1, muss `wstring` durch `string` und `wcout` durch `cout` ersetzt werden. Das 'L' vor den Anführungszeichen entfällt, wie auch die utf-8-Kennung der locale.

24.2 Textverarbeitung

24.2.1 Datei durchsuchen

Das folgende Programm durchsucht eine Datei nach einem Begriff, der durch einen regulären Ausdruck definiert wird. Dabei geschieht die Auswertung wie bei dem Unix-Programm *egrep*. Bei Angabe der Option `-i` spielen Klein- und Großschreibung keine Rolle. Die Grundlagen zu den regulären Ausdrücken finden Sie in Kapitel 12.

Listing 24.19: Datei durchsuchen

```

// cppbuch/k24/textverarbeitung/regex/finde.cpp
#include <iostream>

```

```

#include <fstream>
#include<boost/regex.hpp>
#include<string>
using namespace std;

int main(int argc, char* argv[]) {
    // nicht nach ECMA-Standard, sondern wie egrep auswerten:
    boost::regex::flag_type flags = boost::regex::egrep;
    string gebrauch("Gebrauch: finde.exe [-i] \"regex\" \"dateiname\"");
    string option;
    int start = 1;
    // Ist die Option -i gesetzt?
    if(4 == argc) {
        option = argv[1];
        if(option != "-i") {
            cerr << "Falsche Option: " << gebrauch << endl;
            return 1; // EXIT
        }
        ++start;
        flags |= boost::regex::icase; // Klein-/Großschreibung ignorieren
    }
    else if(3 != argc) {
        cout << gebrauch << endl;
        return 2; // EXIT
    }
    // Datei durchsuchen
    try {
        // flags transportiert die Einstellungen egrep und icase
        boost::regex gesucht(argv[start], flags);
        ifstream quelle(argv[start+1]);
        size_t zeilenr = 0;
        if(!quelle.good()) {
            throw ios::failure("Dateifehler");
        }
        while(quelle.good()) {
            string zeile;
            getline(quelle, zeile);
            ++zeilenr;
            if(boost::regex_search(zeile, gesucht)) {
                cout << zeilenr << ": " << zeile << endl;
            }
        }
    }
    catch(boost::regex_error& re) {
        cerr << "Regex-Fehler: " << re.what() << endl;
    }
    catch(ios::failure& e) {
        cerr << e.what() << endl;
    }
}

```

Das Programm gibt die Zeilen, in denen der Suchbegriff gefunden wird, mit der Zeilennummer aus. Beispiele:

- `finde.exe "hallo" "text.txt"` findet jedes Auftreten von `hallo` in der Datei.
- `finde.exe -i "hallo" "text.txt"` findet auch `Hallo`, `HALL0`, `hallo` usw.
- `finde.exe "\\([\\^])*\\\\" "text.txt"` findet alle Zeilen mit öffnender und schließender Klammer. Die Escape-Zeichen sind notwendig, weil sonst die Klammern als Meta-Zeichen für eine Gruppe interpretiert würden. Die runde Klammer innerhalb einer Zeichenklasse ist kein Meta-Zeichen.
- `finde.exe "\\\\" "text.txt"` findet alle Zeilen mit Backslash. Wie im vorherigen Beispiel zu sehen, wird ein Backslash, der als Escape-Zeichen wirken soll, durch `\\` dargestellt. Wenn er selbst gemeint ist, ist er zu verdoppeln – deswegen vier Backslashes.
- `finde.exe "\\bif *\\(" "text.txt"` findet alle Zeilen mit einer `if`-Anweisung. Dabei verhindert `\\b` (Wortgrenze), dass zum Beispiel auch `exif(...)` gefunden wird. `*` bedeutet, dass das davor stehende Leerzeichen beliebig oft zwischen `if` und der Klammer vorkommen darf.
- `finde.exe "\\bhttp://[\\^:/]\\.de" "text.txt"` findet alle Zeilen mit einer auf `.de` endenden HTTP-Adresse (URI bzw. URL). Zur Zeichenklasse: Nach `:` folgt der Port, der deswegen nicht im Host-Namen enthalten sein darf. `/` kann allenfalls nach dem Host-Namen folgen. Die Syntax ist extrem vereinfacht. Die vollständige URI-Syntax finden Sie unter [\[URI\]](#). Dort wird auch auf den Unterschied zwischen URI und URL eingegangen.



Hinweis

Die Anführungszeichen sind nicht immer notwendig. So kann ebenso gut `text.txt` statt `"text.txt"` geschrieben werden. Sie sind dann erforderlich, wenn die Auswertung durch den Kommandointerpreter verhindert werden muss – zum Beispiel bei Dateinamen oder regulären Ausdrücken, die Leerzeichen enthalten.

24.2.2 Ersetzungen in einer Datei

Das obige Programm durchsucht eine Datei wie *zeilenweise*, was normalerweise zur Suche genügt. Bei Textersetzungen kann es jedoch vorkommen, dass zeilenübergreifendes Suchen gewünscht wird. Beispiele: Nach jeder Zeile eine Leerzeile oder vor jeder Zeile `***` einfügen. Aus diesem Grund wird im nächsten Programm eine Datei als Ganzes eingelesen und dann verarbeitet. Das Ergebnis wird auf der Standardausgabe ausgegeben und kann mit `>` in eine Datei umgeleitet werden.

Listing 24.20: Ersetzen in einer Datei

```
// cppbuch/k24/textverarbeitung/regex/ersetze.cpp
#include <iostream>
#include <fstream>
#include <boost/regex.hpp>
#include <string>

std::string ersetzeInDatei(const boost::regex& gesucht,
                          const std::string& ersatz,
                          const char* dateiname) {
```

```

std::ifstream quelle(dateiname, std::ios::binary | std::ios::in);
if(!quelle.good()) {
    throw std::ios::failure("Dateifehler");
}
std::string alles;
while(quelle.good()) {
    char c = (char)quelle.get();
    if(!quelle.fail()) {
        alles += c;
    }
}
return boost::regex_replace(alles, gesucht, ersatz);
}

using namespace std;

int main(int argc, char* argv[]) {
    boost::regex::flag_type flags = 0; // s.unten icase, falls gefordert
    string gebrauch("Gebrauch: ersetze.exe [-i] \"regex\" \"ersatz\" \"dateiname\"");
    string option;
    int start = 1;
    // Ist die Option -i gesetzt?
    if(5 == argc) {
        option = argv[1];
        if(option != "-i") {
            cerr << "Falsche Option: " << gebrauch << endl;
            return 1; // EXIT
        }
        ++start;
        flags |= boost::regex::icase; // Klein-/Großschreibung ignorieren
    }
    else if(4 != argc) {
        cout << gebrauch << endl;
        return 2; // EXIT
    }

    // Datei durchsuchen
    try {
        boost::regex gesucht(argv[start], flags);
        string ersatz(argv[start+1]);
        string ergebnis = ersetzeInDatei(gesucht, argv[start+1], argv[start+2]);
        cout << ergebnis << endl;
    }
    catch(boost::regex_error& re) {
        cerr << "Regex-Fehler: " << re.what() << endl;
    }
    catch(ios::failure& e) {
        cerr << e.what() << endl;
    }
}

```

Auch hier wird gegebenenfalls die Option `-i` wirksam. Einige Beispiele:

- `ersetze.exe -i "hallo" "hello" text.txt`
ersetzt »hallo«, »hallo« und »HALLO« usw. durch »hello«.
- `ersetze.exe "\\n" "\\n\\n" text.txt` fügt nach jeder Zeile eine Leerzeile ein.
- `ersetze.exe "\\n" "***\\n" text.txt` fügt *** am Ende jeder Zeile ein.

Dies sind nur einfache String-Ersetzungen. Es ist aber auch möglich, sich auf Capturing Groups zu beziehen. Um auf das Beispiel am Anfang des Kapitels 12 zurückzukommen:

```
ersetze.exe "Version +1\\. (2|3)" "Version \\1.0" text.txt
```

wandelt den Text : Nach Version 1.1 kamen Version 1.2 und Version 1.3
in : Nach Version 1.1 kamen Version 2.0 und Version 3.0

um. Das `+-`-Zeichen besagt, dass ein oder mehr Leerzeichen vor der Ziffer 1 stehen dürfen. 2 oder 3 wird jeweiliger Inhalt der Capturing Group. Im Ersatzstring wird `\\1` durch den Inhalt der aktuellen ersten (und hier einzigen) Capturing Group ersetzt, gefolgt von einem Punkt und der 0.

24.2.3 Code-Formatierer

Die meisten IDEs haben einen Code-Formatierer, der unsauber gesetzte Klammern zu-rechtrückt. Diese Code-Formater sind aber nicht geeignet, um etwa mit einem Skript alle Dateien eines Verzeichnisses zu formatieren. Der folgende Code-Formatierer arbeitet nach dem Filterprinzip: Jedes Zeichen wird gelesen, analysiert und im Allgemeinen sofort wieder ausgegeben. Im Besonderen werden jedoch alle Leerzeichen und Tabulatorzeichen am Zeilenanfang ignoriert und Leerzeichen für die Einrückung eingefügt. Die geschweiften Klammern bestimmen die Einrückung. Die Zeilenstruktur bleibt erhalten, d.h. eine Zeile erhält eine neue Anfangsposition, wird aber sonst nicht verändert.

Listing 24.21: Code-Formatierer

```
// cppbuch/k24/textverarbeitung/codeformatter.cpp
#include<iostream>
#include<fstream>
#include<cstring>
using namespace std;

int main(int argc, char* argv[]) {
    if(argc != 3) {
        cerr << "Gebrauch: codeformatter eingabe.cpp ausgabe.cpp" << endl;
        return 1;
    }
    const char* const EINRUECKUNG = " ";
    const size_t ANZAHLCHARS = strlen(EINRUECKUNG);
    bool warEOL = true; // War das letzte Zeichen ein Zeilenende?
    ifstream is(argv[1]);
    ofstream os(argv[2]);
    size_t level = 0;
    char c;
    while(is.good()) {
        do { // nächstes Zeichen lesen, ggf. Leerzeichen schlucken
            is.get(c);
```

```

    } while (is.good() && warEOL && (c == ' ' || c == '\t'));
    if (is.good()) {
        if (c == '}') {
            --level;
        }
        if (warEOL) {
            size_t leveltmp = level+1;
            while (--leveltmp > 0)
                os.write(EINRUECKUNG, ANZAHLCHARS);
        }
        os.put(c);
        warEOL = (c == '\n');
        if (c == '{') {
            ++level;
        }
    }
}
}

```

Dieses Programm hat noch eine kleine Schwäche: Geschweifte Klammern in Strings, Zeichenliteralen oder in Kommentaren werden nicht ignoriert.



Übung

24.1 Erweitern Sie das Programm, um die erwähnte Schwäche zu beheben. Denken Sie daran, dass ein Anführungszeichen auch maskiert sein kann, mit einem Backslash oder in einem Zeichenliteral wie `'''`.

24.2.4 Lines of Code (LOC) ermitteln

Die Anzahl der Code-Zeilen eines Programms nach Abzug aller Kommentare und Leerzeilen (englisch *Lines of Code*), oft mit LOC abgekürzt, ist ein in der Softwaretechnik verwendetes Maß für die Arbeitsleistung, die in einem Programm steckt. Dieses Maß ist eine nur grobe Näherung, unter anderem, weil von dem Schwierigkeitsgrad eines Algorithmus abstrahiert wird. Dennoch wird dieses Maß häufig benutzt, zum Beispiel für statistische Zwecke (Anzahl Fehler pro 1000 LOC). Das folgende Programm ermittelt die Anzahl der Code-Zeilen mithilfe regulärer Ausdrücke, indem alles, was nicht zur Anzahl beiträgt, entfernt wird. Anschließend wird die Zahl der übrig gebliebenen Zeilen ermittelt. Die Kommentare im Programm erläutern die einzelnen Schritte. Die Grundlagen zu den regulären Ausdrücken finden Sie in Kapitel 12.

Listing 24.22: Lines of code zählen

```

// cppbuch/k24/textverarbeitung/regex/loc.cpp
#include <algorithm>
#include <iostream>
#include <fstream>
#include <boost/regex.hpp>
#include <string>

int zaehleLOC(const char* dateiname) {

```

```

std::ifstream quelle(dateiname, std::ios::binary | std::ios::in);
if(!quelle.good()) {
    throw std::ios::failure("Dateifehler");
}
std::string alles;
while(quelle.good()) {
    char c = (char)quelle.get();
    if(!quelle.fail()) {
        alles += c;
    }
}

// Zeilenendekennung vereinheitlichen
alles = boost::regex_replace(alles, boost::regex("(r\\n)"), "\\n");

// Leer- und Tabulatorzeichen entfernen
alles = boost::regex_replace(alles, boost::regex("( |\\t)"), "");

// Escape-Zeichen löschen
alles = boost::regex_replace(alles, boost::regex("\\\\\\"), "");

// Alle Strings durch "" nicht-greedy ersetzen
alles = boost::regex_replace(alles, boost::regex("\\".*?""), "\\\"\"");

// alle /* enthaltenen // Kommentare ersetzen. Leerzeichen
// muss sein, um nicht auf // /** -> /** reinzufallen
alles = boost::regex_replace(alles, boost::regex("//[^\n]*\\/\\*"), "// ");

// mit /* beginnende Kommentare löschen
alles = boost::regex_replace(alles, boost::regex("(?s)/\\*.*?\\/\\*"), "");

// alle // Kommentare löschen
alles = boost::regex_replace(alles, boost::regex("//[^\n]*"), "");

// Führende Whitespaces entfernen.
// Damit werden auch Leerzeilen entfernt (\\n am Zeilenanfang)
alles = boost::regex_replace(alles, boost::regex("^\\s+"), "");

// Anzahl der Zeilen zurückgeben
return 1 + std::count(alles.begin(), alles.end(), '\\n');
}

using namespace std;

int main(int argc, char* argv[]) {
    string gebrauch("Gebrauch: loc.exe dateiname");
    if(2 != argc) {
        cout << gebrauch << endl;
        return 1;
    }
    // EXIT

```

```

}
try {
    cout << "Die Datei " << argv[1] << " hat "
         << zaehleLOC(argv[1]) << " Lines of Code (LOC)." << endl;
}
catch(boost::regex_error& re) {
    cerr << "Regex-Fehler: " << re.what() << endl;
}
catch(ios::failure& e) {
    cerr << e.what() << endl;
}
}

```

24.2.5 Zeilen, Wörter und Zeichen einer Datei zählen

Diese Aufgabe übernimmt das Unix-Programm `wc` (word count). Das Kommando `wc datei` gibt die Anzahl der Zeilen, Wörter und Zeichen in dieser Reihenfolge aus. Ein vergleichbares C++-Programm ist die Lösung zur Aufgabe 2.4 auf Seite 99, wobei Umlaute dort keine Berücksichtigung finden. Dies lässt sich leicht ändern:

- `locale::global(locale("de_DE"))`; einstellen.
- Die Abfrage `if (c >= 'A' && c <= 'Z' || c >= 'a' && c <= 'z')` durch `if (isalpha(c))` ersetzen (Header `<cstring>`). `isalpha(c)` berücksichtigt die globale `locale`-Einstellung (siehe auch `cppbuch/k24/textverarbeitung/wcErsatz.cpp`).

24.2.6 CSV-Datei lesen

Tabellenkalkulationsprogramme können Tabellen im CSV-Format exportieren. CSV bedeutet »comma separated values« und ist ein Textformat, also nicht binär. Im folgenden Beispiel wird eine im CSV-Format vorliegende Tabelle mit ganzen Zahlen eingelesen und in einem `vector<vector<int>>` abgelegt, auf den wie auf ein normales zweidimensionales Array zugegriffen werden kann.

Listing 24.23: CSV-Datei lesen

```

// cppbuch/k24/textverarbeitung/csvauswerten.cpp
#include<string>
#include<cstring>
#include<vector>
#include<cstdlib> // für exit()
#include<fstream>
#include<iostream>

std::vector<int> splitInt(char* buf, const char* trennzeichen) { // siehe Text
    std::vector<int> vec;
    char* str = strtok(buf, trennzeichen);
    while(str) {
        vec.push_back(atoi(str));
        str = strtok(NULL, trennzeichen);
    }
    return vec;
}

```



```

using namespace std;

int main( ) {
    // Definieren der Eingangsdatei
    ifstream quelle;           // Datentyp für Eingabestrom
    string csvDateiname;
    cout << "Csv-Datei? ";
    cin >> csvDateiname;
    quelle.open(csvDateiname.c_str(), ios::in);
    if (!quelle) { // Fehlerabfrage
        cerr << csvDateiname << " kann nicht geöffnet werden!\n";
        exit(-1);
    }
    const size_t N = 1000;
    char buf[N];               // Muss groß genug für eine Zeile sein
    vector<vector<int> > tabelle; // Vektor von Tabellenzeilen
    while(quelle.good()) {
        quelle.getline(buf, N);
        if(!quelle.fail()) {
            tabelle.push_back(splitInt(buf, ","));
        }
    }
    quelle.close();
    // Tabelle ausgeben
    for(size_t i=0; i < tabelle.size(); ++i) {
        for(size_t j=0; j < tabelle[i].size(); ++j) {
            cout << tabelle[i][j] << " ";
        }
        cout << endl;
    }
}

```

Die Zeilen werden mit `getline()` eingelesen. Weil die Datei nicht mit `ios::binary` eröffnet wird, werden sowohl die Unix- als auch die Windows-Zeileneindeckung akzeptiert.

Jede eingelesene Zeile wird von der Funktion `splitInt(char* buf, const char* trennzeichen)` analysiert. `trennzeichen` ist dabei eine Folge von Trennzeichen, die in diesem Fall nur aus einem Komma besteht. Die C-Funktion `strtok()` zerlegt die Zeichenkette in Token, die durch Kommata getrennt sind. Der Parameter ist `char*` anstatt `const char*`, weil `strtok()` jedes Trennzeichen durch ein Null-Byte ersetzt. Die Variable `str` verweist jeweils auf den Beginn des nächsten, mit einem Null-Byte abgeschlossenen Tokens. Die C-Funktion `atoi()` (für ASCII to integer) gibt das Token als `int`-Zahl zurück, die an den Vektor angehängt wird. `splitInt()` gibt also jede CSV-Zeile als `vector<int>` zurück, der an die Variable `tabelle` angehängt wird. Wie am Ende des Beispiels zu sehen, kann auf `tabelle` wie auf ein Array zugegriffen werden (Grundlagen dazu siehe Abschnitt 9.8).

Vorteil dieser Vorgehensweise ist, dass die Datei nur einmal gelesen werden muss. Wollte man eine Tabelle mit festen Ausmaßen definieren, müsste die Datei einmal gelesen werden, um die Anzahl der benötigten Zeilen festzustellen, und ein zweites Mal, um die Werte auszulesen – oder man müsste die gesamte Datei als String-Vektor zwischenspeichern.

24.2.7 Kreuzreferenzliste

Eine Kreuzreferenzliste ist eine Liste, die die Worte oder Bezeichner eines Textes alphabetisch mit den Positionen des Vorkommens, hier den Zeilennummern, enthält. Hier ist der Anfang der Kreuzreferenzliste zum Programm *crossrefISO8859_1.cpp*:

```

_           : 54 63 65
a           : 22 23 25
abgespeichert : 11
an          : 68
Anfang      : 53

argc        : 30 31 44
argv        : 30 32 34 35 38 40 45
auch        : 14
auf         : 14 19
auto        : 74

```

Das Programm wurde mit der Locale »de_DE« aufgerufen. Andernfalls, also bei der Standard-Locale C, gäbe es nur eine Auswertung nach dem ASCII, sodass »berücksichtigt« in zwei Bezeichner »ber« und »cksichtigt« zerfiel. Die passende Datenstruktur ist ein Map-Container. Die Wertepaare bestehen aus dem Bezeichner vom Typ `string` als Schlüssel und aus einem Set mit den Zeilennummern. Aufgrund der sortierten Ablage ist kein besonderer Sortiervorgang notwendig. Weil das Programm zeilenweise liest, wäre statt eines Sets eine Liste denkbar. Ein Set hat aber den Vorteil, dass ein mehrfaches Vorkommen eines Worts in einer Zeile nur einfach gezählt wird.

Listing 24.24: Programm für eine Kreuzreferenzliste

```

// cppbuch/k24/textverarbeitung/crossrefISO8859_1.cpp
#include<cctype>
#include<fstream>
#include<iostream>
#include<set>
#include<locale>
#include<map>
#include<string>
#include<showSequence.h>
// Test für Locale de_DE: Mücke Mücke Süßholz Süsse
/* Diese Datei ist im ISO 8859-1 Format abgespeichert, so dass
   Umlaute nur einem Byte entsprechen.
   Konvertierung nach UTF-8 bewirkt, dass der obige Test fehlschlägt!
   Zur korrekten Dartsellung sollte das Terminal auch auf ISO 8859-1
   eingestellt sein.
*/
// Um eine unterschiedliche Sortierung für Groß- und Kleinschreibung
// zu vermeiden, wird die Klasse Vergleich eingesetzt, die die zu
// vergleichenden Strings vorher auf Kleinschreibung normiert.

struct Vergleich {
    bool operator()(std::string a, std::string b) const { // per Wert
        for(size_t i=0; i< a.length(); ++i) a[i]=std::tolower(a[i]);
        for(size_t i=0; i< b.length(); ++i) b[i]=std::tolower(b[i]);
        return a < b; // Stringvergleich berücksichtigt Locale
    }
};

```

```

    }
};

using namespace std;
int main(int argc, char* argv[]) {
    if(argc == 1) {
        cout << "Gebrauch: " << argv[0] << " Dateiname [locale]\n"
              << " Beispiele (vorgegebene Locale: C)\n"
              << argv[0] << " crossrefISO8859_1.cpp de_DE\n"
              << argv[0] << " crossrefISO8859_1.cpp" << endl;
        return 0;
    }
    ifstream quelle(argv[1]);
    if(!quelle.good()) {
        cout << argv[1] << " nicht gefunden!\n";
        return 1;
    }

    if(argc == 3) {
        locale::global(locale(argv[2])); // ggf. global setzen
    }

    map<string, set<int>, Vergleich> bezeichnerZeilenMap;

    int zeilenr = 1;
    while(quelle.good()) {
        char c = '\0';
        // Anfang des Bezeichners finden
        while(quelle.good() && !(isalpha(c) || '_' == c)) {
            quelle.get(c);
            if(c == '\n') {
                ++zeilenr;
            }
        }
        if(quelle.good()) {
            string bezeichner(1, c);
            // Rest des Bezeichners einsammeln
            while(quelle.good() && (isalnum(c) || '_' == c)) {
                quelle.get(c);
                if(isalnum(c) || '_' == c)
                    bezeichner += c;
            }
            quelle.putback(c); // zurück an den Eingabestrom
            if(c) { // Bezeichner gefunden?
                bezeichnerZeilenMap[bezeichner].insert(zeilenr); // Eintrag
            }
        }
    }

    auto iter = bezeichnerZeilenMap.begin();
    while(iter != bezeichnerZeilenMap.end()) {
        cout << (*iter).first; // Bezeichner
        cout.width(20 - (*iter).first.length()); // Position bis : einstellen
        cout << ": ";
    }
}

```

```

        showSequence((*iter++).second); // Zeilennummern
    }
}

```

Das Eintragen der Zeilennummer in die Sets nutzt aus, dass `BezeichnerZeilenMap::operator[]()` eine Referenz auf den Eintrag zurückgibt, auch wenn dieser erst angelegt werden muss, weil der Schlüssel noch nicht existiert. Der Eintrag zu dem Schlüssel `bezeichner` ist ein Set, sodass von vornherein die richtige Reihenfolge sichergestellt ist. Die Ausgabe der Kreuzreferenzliste profitiert von der sortierten Speicherung. Das Element `first` eines Wertepaares ist der Bezeichner (Schlüssel), das Element `second` ist der Set, der mit dem bekannten Template `showSequence` ausgegeben wird.

Der Operator `operator()(string a, string b)` zum Vergleich zweier Strings unabhängig von Groß- und Kleinschreibung mag nicht besonders performant erscheinen, weil eine Kopie der Argumente notwendig und jede Kopie implizit mit einer `new`-Operation verbunden ist. Weil jedoch die Größe der Sets relativ klein ist (proportional zur Anzahl der Zeilen, in denen der Bezeichner vorkommt), hält sich die Anzahl der Aufrufe in Grenzen. Auf die Kopie der Strings zu verzichten, ist nur dann sinnvoll, wenn der Vergleich ausschließlich auf dem ASCII beruhen kann, etwa:

```

// nur für Locale C geeignet!
bool operator()(const std::string& x, const std::string& y) const {
    size_t kleinereLaenge = std::min(x.length(), y.length());
    for(size_t i=0; i < kleinereLaenge; ++i) {
        char cx = std::tolower(x[i]);
        char cy = std::tolower(y[i]);
        if(cx != cy) {
            return cx < cy; // Zeichenvergleich berücksichtigt nicht Locale
        }
    }
    // Falls alle Zeichen gleich sind, ist der kürzere String kleiner
    return x.length() < y.length();
}

```

Für andere Locales ist dieses Verfahren unbrauchbar, weil bei einem sprachlich richtigen Vergleich mehr als ein Zeichen beteiligt sein kann. Beispiel: Nach deutschen Sortierregeln kommt »Masse« vor »Maße«.

24.3 Operationen auf Folgen

Diese Algorithmen beschreiben allgemeine numerische Operationen auf Containern. Im Allgemeinen werden die Container Zahlen enthalten. Aufgrund der Formulierung der Algorithmen als Template ist dies jedoch nicht zwingend; es müssen nur die benötigten Operationen vorhanden sein. So steht das `+`-Zeichen bei Zahlen für die Addition, bei Strings für das Verketteten. Der in Abschnitt 24.3.5 verwendete Algorithmus zur Summenbildung wird daher bei einem Container mit Strings als Ergebnis die Verkettung aller Container-Elemente liefern.

24.3.1 Container anzeigen

In diesem Buch wird zur Abkürzung gelegentlich die Hilfsfunktion `showSequence()` verwendet, um einen Container auf dem Bildschirm anzuzeigen. Sie liegt im Verzeichnis `cppbuch/include` der Beispiele und ist wie folgt definiert:

Listing 24.25: Hilfsfunktion zur Anzeige von Containern

```
// cppbuch/include/showSequence.h
#ifndef SHOWSEQ_H
#define SHOWSEQ_H
#include<iostream>

template<class Container>
void showSequence(const Container& s,
                  const char* abschluss = "\n",
                  const char* trennzeichen = " ") {
    auto iter = s.begin();
    while(iter != s.end()) {
        std::cout << *iter++ << trennzeichen;
    }
    std::cout << abschluss;
}
#endif
```

24.3.2 Folge mit gleichen Werten initialisieren

Wenn eine Sequenz ganz oder teilweise mit immer gleichen Werten, nämlich Kopien von `value`, vorbesetzt werden soll, eignen sich die Algorithmen `fill()` oder `fill_n()`:

```
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

```
// Beispiel: alle Werte eines Vektors v mit 0.45 besetzen
fill(v.begin(), v.end(), 0.45);
```

```
template<class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value);
```

Im Unterschied zu `fill()` erwartet `fill_n()` die Angabe `n`, wie viele Elemente der Sequenz, auf die `first` verweist, mit `value` vorbesetzt werden sollen. Zurückgegeben wird ein Iterator auf das Ende des modifizierten Bereichs.

```
// Auszug aus cppbuch/k24/vermishtes/fill.cpp
// Beispiel: die erste Hälfte eines Vektors v mit 0.45 besetzen
fill_n(v.begin(), v.size()/2, 0.45);
```

24.3.3 Folge mit Werten eines Generators initialisieren

Ein Generator im Algorithmus `generate()` ist ein Funktionsobjekt oder eine Funktion, die ohne Parameter aufgerufen und deren Ergebnis den Elementen der Sequenz der Reihe nach zugewiesen wird. Wie bei `fill()` gibt es eine Variante, die ein Iteratorpaar erwartet, und eine Variante, die den Anfangsiterator und eine Stückzahl benötigt:

```
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

```
template<class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
```

Das Beispiel zeigt die beide Varianten:

- Im ersten Teil wird als Generator eine Funktion genommen, die Zweierpotenzen erzeugt.
- Im zweiten Teil wird die erste Hälfte des Vektors mit zufälligen Zahlen versehen. Als Generator dient der Zufallszahlengenerator von Seite 712.

Listing 24.26: Zahlenfolgen erzeugen

```
// cppbuch/k24/vermishtes/generate.cpp
#include<algorithm>
#include<vector>
#include<showSequence.h>
#include<Random.h>
using namespace std;

int zweierpotenz() {
    static int wert = 1; // mit 1 anfangen
    return wert <<= 1; // Wert verdoppeln
}

int main() {
    vector<int> v(10);
    generate(v.begin(), v.end(), zweierpotenz);
    showSequence(v); // 2 4 8 16 32 64 128 256 512 1024

    Random zufall(10000);
    generate_n(v.begin(), v.size()/2, zufall);
    showSequence(v);
}
```

24.3.4 Folge mit fortlaufenden Werten initialisieren

Die Funktion `iota()` (Header `<numeric>`) füllt ein Intervall mit fortlaufenden Werten. Iota heißt der neunte Buchstabe des griechischen Alphabets (ι). Das entsprechende deutsche Wort Jota bedeutet etwa sehr kleine Menge oder das Geringste. Der Name wurde jedoch nicht deswegen, sondern in Anlehnung an den ι -Operator der Programmiersprache APL gewählt. Die APL-Anweisung $\iota\ n$ liefert als Indexgenerator einen Vektor mit einer ansteigenden Folge der Zahlen 1 bis n . Im Beispiel des nächsten Abschnitts füllt `iota()` einen Vektor. Die Funktion selbst ist recht einfach, wie an der Definition zu sehen ist:

Listing 24.27: Algorithmus `iota`

```
template<class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value) {
    while (first != last)
        *first++ = value++;
}
```

24.3.5 Summe und Produkt

Der Algorithmus `accumulate()` (Header `<numeric>`) wendet auf alle Werte $*i$ eines Iterators i von `first` bis `last` den Plus-Operator an. Es gibt eine interne Variable `acc`, die mit `init` initialisiert wird. Anschließend wird für jeden Iterator die Operation `acc += *i` ausgeführt. Falls statt dessen eine andere Operation treten soll, existiert eine überladene Variante, der die Operation als letzter Parameter übergeben wird. Die jeweils ausgeführte Operation ist dann `acc = binOp(acc, *i)`. Die Prototypen sind:

```
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);
```

Bei numerischen Elementen und ohne Angabe einer Operation berechnet `accumulate()` für einen Vektor v mit den Elementen v_i ($i = 0 \dots v.size()-1$) die Summe $init + \sum_i v_i$. Im Beispiel wird als Startwert für `init` der Wert 0 angenommen. Im zweiten Teil wird der Operator `multiplies` (siehe Seite 753) eingesetzt. Das Ergebnis ist $init \cdot \prod_i v_i$. Weil der Vektor im Beispiel mit der Folge der natürlichen Zahlen initialisiert wird und der Startwert gleich 1 ist, ist das Produkt gleich der Fakultät von 10.

Entsprechend der Bedeutung des `++`-Operators als Verkettungsoperation bei Strings liefert `accumulate()`, auf einen Container mit Strings angewendet, deren Verkettung, wie am Ende des Beispiels gezeigt wird – auch wenn das nicht gerade die übliche Anwendung ist. Wie man einen eigenen Operator für `accumulate()` schreibt, zeigt der folgende Abschnitt »Mittelwert und Standardabweichung«.

Listing 24.28: Summen- und Produktbildung

```
// cppbuch/k24/folgen/accumulate.cpp
#include<vector>
#include<numeric>
#include<iostream>
#include<string>
using namespace std;

int main() {
    vector<int> v(10);
    iota(v.begin(), v.end(), 1);
    cout << "Summe = " << accumulate(v.begin(), v.end(), 0) // 55
          << endl;
    cout << "Produkt = " << accumulate(v.begin(), v.end(), 1L,
                                     multiplies<long>())      // 3628800
          << endl;
    // accumulate() mit string:
    vector<string> vstr(26);
    iota(vstr.begin(), vstr.end(), 'A'); // Vektor mit Buchstaben füllen
    cout << accumulate(vstr.begin(), vstr.end(), string("Alphabet: "))
          << endl;
}
```

24.3.6 Mittelwert und Standardabweichung

Wie oben beschrieben, wird in `accumulate()` für jeden Iterator `i` intern die Anweisung `acc = binOp(acc, *i)` ausgeführt. Der Operator verknüpft `acc` mit dem Wert, auf den der Iterator zeigt, und gibt den neuen Wert von `acc` zurück. Im Beispiel wird dieses Wissen genutzt, um einen Operator als Funktor zur Berechnung des Abweichungsquadrats zu formulieren:

Listing 24.29: Mittelwert und Standardabweichung berechnen

```
// cppbuch/k24/folgen/statistik.cpp
#include<vector>
#include<cmath>
#include<numeric>
#include<iostream>
#include<Random.h>

// Funktor zur Berechnung des Quadrats der Differenz zu einem Wert
template<class T>
class Abweichungsquadrat {
public:
    Abweichungsquadrat(T m)
        : mittel(m) {
    }
    T operator()(const T& acc, const T& iterWert) const {
        const T d = iterWert - mittel;
        return acc + d*d;
    }
private:
    T mittel;
};
using namespace std;

int main() {
    vector<double> v(6);
    iota(v.begin(), v.end(), 1); // Vektor füllen
    double summe = accumulate(v.begin(), v.end(), 0.0); // 0.0 (double), nicht: 0
    // weil der Typ den Typ des Ergebnisses bestimmt (sonst Genauigkeitsverlust).
    cout << "Summe      : " << summe << endl;
    double mittelwert = summe/v.size();
    cout << "Mittelwert   : " << mittelwert << endl;
    Abweichungsquadrat<double> aq(mittelwert);
    double varianz = accumulate(v.begin(), v.end(), 0.0, aq)/v.size();
    cout << "Varianz      : " << varianz << endl;
    cout << "Standardabweichung: " << sqrt(varianz) << endl;
}
```

24.3.7 Skalarprodukt

Der Algorithmus `inner_product()` (Header `<numeric>`) addiert das Skalarprodukt zweier Container `u` und `v`, die meistens Vektoren sein werden, auf den Anfangswert `init`:

$$\text{Ergebnis} = \text{init} + \sum_i v_i \cdot u_i$$

Anstelle der Addition und Multiplikation können auch andere Operationen gewählt werden. Die Prototypen sind:

```
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>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                binaryOperation1 binOp1, binaryOperation2 binOp2);
```

In einem euklidischen n -dimensionalen Raum R^n ist die Länge eines Vektors durch die Wurzel aus dem Skalarprodukt des Vektors mit sich selbst definiert. Das Beispiel berechnet zuerst die Länge eines Vektors $(1, 1, 1, 1)$ im R^4 . Der Wert für `init` muss 0 sein. Im zweiten Teil des Beispiels wird die Entfernung zwischen zwei Punkten, die durch zwei verschiedene Vektoren repräsentiert werden, berechnet.

Listing 24.30: Länge und Entfernung mit `inner_product()`

```
// cppbuch/k24/folgen/innerproduct.cpp
#include<numeric>
#include<vector>
#include<cmath>
#include<iostream>

// Funktor zur Berechnung des Quadrats einer Differenz
template<class T>
struct difference_square {
    T operator()(const T& x, const T& y) {
        const T d = x - y;
        return d*d;
    }
};

using namespace std;

int main() {
    const int DIMENSION = 4;
    vector<int> v(DIMENSION, 1);

    cout << "Länge des Vektors v = "
         << sqrt( (double)inner_product(v.begin(), v.end(), v.begin(), 0))
         << endl;

    // Um die Anwendung anderer mathematischer Operatoren zu zeigen,
    // wird im Folgenden die Entfernung zwischen zwei Punkten berechnet.

    // 2 Punkte p1 und p2
    vector<double> p1(DIMENSION, 1.0); // Einheitsvektor
    vector<double> p2(DIMENSION);
    iota(p2.begin(), p2.end(), 1.0); // beliebiger Vektor
```

```

    cout << "Entfernung zwischen p1 und p2 = "
        << sqrt( inner_product(p1.begin(), p1.end(),
                                p2.begin(), 0.0,
                                plus<double>(),
                                difference_square<double>()))
        << endl;
}

```

24.3.8 Folge der Teilsummen oder -produkte

Die Partialsummenbildung (Header `<numeric>`) funktioniert ähnlich wie `accumulate()`, nur dass das Ergebnis eines jeden Schritts in einem Ergebniscontainer abgelegt wird, der durch den Iterator `result` gegeben ist, und dass es keinen `init`-Wert gibt. Die Prototypen sind:

```

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);

```

Das Beispiel zeigt beide Varianten. Die jeweils letzte Zahl einer Folge korrespondiert mit dem Ergebnis von `accumulate()` aus dem obigen Beispiel.

Listing 24.31: Folge der Teilsummen

```

// cppbuch/k24/folgen/partialsum.cpp
#include<numeric>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
    vector<long> v(10), partialsummen(10);
    iota(v.begin(), v.end(), 1); // natürliche Zahlen 1 bis 10
    partial_sum(v.begin(), v.end(), partialsummen.begin());
    cout << "Partialsummen = ";
    showSequence(partialsummen); // 1 3 6 10 15 21 28 36 45 55
    cout << "Folge von Fakultäten = ";
    partial_sum(v.begin(), v.end(), v.begin(), multiplies<long>());
    showSequence(v); // 1 2 6 24 120 720 5040 40320 362880 3628800
}

```

24.3.9 Folge der Differenzen

Der Algorithmus `adjacent_difference()` (Header `<numeric>`) berechnet die Differenz zweier aufeinanderfolgender Elemente eines Containers `v` und schreibt das Ergebnis in einen Ergebniscontainer `e`. Auf diesen Ergebniscontainer verweist der Iterator `result`. Da es genau einen Differenzwert weniger als Elemente gibt, bleibt das erste Element erhalten. Wenn das erste Element den Index 0 trägt, gilt also:

$$e_0 = v_0$$

$$e_i = v_i - v_{i-1}; \quad i > 0$$

Außer der Differenzbildung sind andere Operationen möglich. Die Prototypen sind:

```
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);
```

Das Beispiel zeigt beide Varianten. In der ersten werden Differenzwerte berechnet, in der zweiten eine Folge von Fibonacci-Zahlen. Leonardo von Pisa, genannt Fibonacci, war ein italienischer Mathematiker und lebte ca. 1180–1240.

Listing 24.32: Folge der Differenzen und Fibonacci-Zahlen

```
// cppbuch/k24/folgen/adjacent_difference.cpp
#include<numeric>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
    vector<long> v(10), adjacDiff(10);
    iota(v.begin(), v.end(), 0);           // 0 1 2 3 4 5 6 7 8 9
    cout << "Differenzen = ";
    adjacent_difference(v.begin(), v.end(), // Quelle
                       adjacDiff.begin(), // Ziel
                       showSequence(adjacDiff), // 0 1 1 1 1 1 1 1 1 1
                       // Zweite Variante: Statt der (vorgegebenen) Differenz wird
                       // ein anderer Operator, hier plus<int>() übergeben.
                       // Im Beispiel werden damit Fibonacci-Zahlen erzeugt.
                       vector<int> fib(16);
                       fib[0] = 1;           // Anfangswert
                       // Ein Startwert genügt hier, weil der erste Wert
                       // an Position 1 eingetragen wird (Formel  $e_i = v_i - v_{i-1}$  oben)
                       // und sich damit der zweite Wert von selbst ergibt
                       // (beachte den um 1 verschobenen result-Iterator in der Parameterliste).
                       cout << "Fibonacci-Zahlen = ";
                       adjacent_difference(fib.begin(), fib.end()-1, // Quelle
                                           fib.begin()+1,             // Ziel
                                           plus<int>(),               // Funktor
                                           showSequence(fib);           // 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
    }
```

Wenn anstatt der vorgegebenen Differenz die Summe der beiden Vorgänger genommen wird, füllt sich der Ergebniscontainer mit einer Folge der Fibonacci-Zahlen. Fibonacci fragte sich, wie viele Kaninchen-Pärchen es wohl nach n Jahren gibt, wenn jedes Pärchen ab dem zweiten Jahr pro Jahr ein weiteres Pärchen erzeugt. Dass Kaninchen irgendwann

sterben, wurde bei der Fragestellung ignoriert. Die Antwort auf diese Frage ist, dass die Anzahl der Kaninchen im Jahre n gleich der Summe der Jahre $n - 1$ und $n - 2$ ist. Die Fibonacci-Zahlen spielen in der Informatik eine Rolle ([CLR]). Man beachte, dass bei der Erzeugung der Folge der Iterator `result` zu Beginn gleich `fib.begin()+1` sein muss.

24.3.10 Minimum und Maximum

Die Templates `min_element()` und `max_element()` geben jeweils einen Iterator auf das kleinste (bzw. das größte) Element in einem Intervall `[first, last)` zurück. Bei Gleichheit der Iteratoren wird der erste zurückgegeben. Die Komplexität ist linear. Die Prototypen sind:

```
template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
```

```
template<class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                           Compare comp);
```

```
template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
```

```
template<class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                           Compare comp);
```

`minmax_element()` gibt dementsprechend ein Paar von Iteratoren auf das kleinste und auf das größte Element zurück.

```
template<class ForwardIterator>
pair<ForwardIterator, ForwardIterator>
minmax_element(ForwardIterator first, ForwardIterator last);
```

```
template<class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator>
minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
```

Das folgende Beispiel enthält einen Funktor (Funktionsobjekt) Absolutbetrag, um eine Variante mit dem Vergleichsobjekt zu zeigen.

Listing 24.33: Minimum und Maximum finden

```
// cppbuch/k24/folgen/minmax.cpp
#include<algorithm>
#include<vector>
#include<numeric>
#include<utility> // pair
#include<showSequence.h>
using namespace std;

struct Absolutbetrag {
    bool operator()(int x, int y) {
        return abs(x) < abs(y);
    }
};
```

```

    }
};

int main() {
    vector<long> v(10);
    iota(v.begin(), v.end(), -5); // -5 ... +4
    showSequence(v);
    cout << "Minimum: " << *min_element(v.begin(), v.end()) << endl; // -5
    cout << "Maximum: " << *max_element(v.begin(), v.end()) << endl; // 4
    cout << "Minimum des Absolutbetrags: "
         << *min_element(v.begin(), v.end(), Absolutbetrag()) << endl; // 0

    cout << "Minimum und Maximum mit nur einem Funktionsaufruf: ";
    typedef vector<long>::iterator iter; // zur Abkürzung
    pair<iter, iter> p = minmax_element(v.begin(), v.end());
    cout << *p.first << " " << *p.second << endl;           // -5 4
}

```



Mehr über `pair` lesen Sie in Abschnitt 27.3.

Anstelle eines Funktors ist auch die Übergabe einer Funktion möglich:

```

bool abskleiner(int x, int y) {
    return abs(x) < abs(y);
}

// ... Rest weggelassen
cout << "Minimum des Absolutbetrags mit Funktion statt Funtor: "
     << *min_element(v.begin(), v.end(), abskleiner) << endl;

```

24.3.11 Elemente rotieren

Der Algorithmus `rotate()` verschiebt die Elemente einer Sequenz nach links, wobei die vorne herausfallenden am Ende wieder eingefügt werden.

```

template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last);

```

```

template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result);

```

Es handelt sich um eine Linksrotation. `first` und `last` geben den Bereich an, in dem rotiert werden soll. Der Iterator `middle` zeigt auf das Element, das nach der Rotation am Anfang der Sequenz stehen soll. Der erste Teil des Beispiels zeigt die vollständige Rotation in einem Vektor `v` um eins und dann um zwei Elemente. Die Laufvariable `shift` bestimmt, um wie viel rotiert wird. Der zweite Teil zeigt `rotate_copy()`. Dieser Algorithmus lässt den Container unverändert `v`, schreibt aber das Ergebnis in einen anderen Container, hier `erg`.

Listing 24.34: Rotation von Elementen

```
// cppbuch/k24/folgen/rotate.cpp
#include<algorithm>
#include<vector>
#include<numeric>
#include<showSequence.h>
using namespace std;

int main() {
    vector<int> v(10);
    iota(v.begin(), v.end(), 0);
    for(size_t shift = 1; shift < 3; shift++) {
        cout << "Rotation um " << shift << endl;
        for(size_t i = 0; i < v.size()/shift; ++i) {
            showSequence(v);
            rotate(v.begin(), v.begin() + shift, v.end());
        }
    }
    cout << "Rotation mit Kopie:" << endl;
    vector<int> erg(10);
    rotate_copy(v.begin(), v.begin() + 3, v.end(), erg.begin());
    showSequence(erg);
}
```

Das Programm gibt aus:

```
Rotation um 1
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 0
2 3 4 5 6 7 8 9 0 1
...
9 0 1 2 3 4 5 6 7 8

Rotation um 2
0 1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 0 1
4 5 6 7 8 9 0 1 2 3
...
8 9 0 1 2 3 4 5 6 7

Rotation mit Kopie:
3 4 5 6 7 8 9 0 1 2
```

24.3.12 Elemente verwürfeln

Der Algorithmus `random_shuffle()` dient zum Mischen der Elemente einer Sequenz, also zur zufälligen Änderung ihrer Reihenfolge. Die Sequenz muss Random-Access-Iteratoren zur Verfügung stellen, zum Beispiel `vector` oder `deque`. Der Algorithmus ist in zwei Varianten vorhanden:

```
template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
```


Im folgenden Beispiel werden überzählige gleiche Elemente eines Vektors entfernt, sodass im Ergebnis kein Element doppelt auftritt.

Listing 24.36: Dubletten entfernen

```
// cppbuch/k24/folgen/unique.cpp
#include<iostream>
#include<algorithm>
#include<vector>
#include<iterator>
#include<showSequence.h>
using namespace std;

int main() {
    vector<int> v(20);
    // Folge mit benachbarten gleichen Elementen
    for(size_t i = 0; i < v.size(); ++i) {
        v[i] = i/3;
    }
    showSequence(v);          // 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 6 6
    // Voraussetzung: Container ist sortiert
    vector<int>::iterator last = unique(v.begin(), v.end());
    showSequence(v);          // 0 1 2 3 4 5 6 2 2 3 3 3 4 4 4 5 5 6 6
    v.erase(last, v.end());
    showSequence(v);          // 0 1 2 3 4 5 6
}
```



Hinweis

`unique()` schiebt alle voneinander verschiedenen Werte nach vorn und gibt die letzte gültige Position zurück. Die Länge des Containers wird nicht verändert!

Das ist im Beispiel der Grund für das Löschen mit `erase()`. Die beiden letzten Schritte können zusammengefasst werden:

```
// Dubletten beseitigen und Vektor entsprechend kürzen
v.erase(unique(v.begin(), v.end()), v.end());
```

Sie fragen sich vielleicht, was die `unique()`-Variante mit dem binären Prädikat soll. Man kann sich vorstellen, dass ein Container, um Platz zu sparen oder aus anderen Gründen, *Zeiger* auf Objekte hält. Die Funktion `unique()` vergleicht die Objekte, um Duplikate erkennen zu können – aber natürlich ist es dabei nicht sinnvoll, Zeiger zu vergleichen. Hier kommt das binäre Prädikat ins Spiel. Im Beispiel gibt es einen Vektor `v` als Container, der Zeiger auf `string`-Objekte aufnimmt.

Schon um diesen Container sortieren zu können, dürfen nicht die Zeiger verglichen werden, sondern die daran hängenden Objekte. Aus diesem Grund wird unten dem Aufruf zum Sortieren des Vektors die binäre Vergleichsfunktion `kleiner()` und dem Aufruf von `unique()` die binäre Vergleichsfunktion `gleich()` mitgegeben. Deren beider Definition sehen Sie unten.

Listing 24.37: Dubletten in Container mit Zeigern entfernen

```
// cppbuch/k24/folgen/uniqueptr.cpp
#include<iostream>
#include<algorithm>
#include<vector>
#include<iterator>
#include<showSequence.h>
using namespace std;

void anzeige(const vector<string*> & v) { // Hilfsfunktion zur Abkürzung
    for(size_t i = 0; i < v.size(); ++i) {
        cout << *v[i] << " ";
    }
    cout << endl;
}

bool gleich(const string* p1, const string* p2) {
    return *p1 == *p2; // Vergleich der Werte, nicht der Zeiger
}

bool kleiner(const string* p1, const string* p2) {
    return *p1 < *p2; // Vergleich der Werte, nicht der Zeiger
}

int main() {
    // automatische Typumwandlung in string
    string strarr[] = {"string", "array", "mit", "mit", "dubletten",
                      "dubletten", "dubletten"};
    size_t anzahl = sizeof(strarr)/sizeof(strarr[0]);
    vector<string*> v;
    for(size_t i = 0; i < anzahl; ++i) {
        v.push_back(&strarr[i]);
    }
    cout << "Original:\n";
    anzeige(v);
    // Voraussetzung für unique(): Container ist sortiert:
    sort(v.begin(), v.end(), kleiner);
    cout << "sortiert:\n";
    anzeige(v);
    v.erase(unique(v.begin(), v.end(), gleich), v.end());
    cout << "nach erase:\n";
    anzeige(v);
}
```

24.3.14 Reihenfolge umdrehen

`reverse()` dreht die Reihenfolge der Elemente einer Sequenz um: Die ersten werden die letzten sein – und umgekehrt. Weil das erste Element mit dem letzten vertauscht wird, das zweite mit dem zweitletzten usw., ist ein bidirektionaler Iterator erforderlich, der die Sequenz beidseitig beginnend bearbeiten kann.

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                           BidirectionalIterator last,
                           OutputIterator result);
```

Im folgenden Beispiel wird `reverse()` jeweils auf ein `char`-Array und einen `String` angewendet. `reverse_copy()` arbeitet mit einem Vektor von `Strings`, das Ergebnis wird im Vektor kopie abgelegt.

Listing 24.38: Reihenfolge umdrehen

```
// cppbuch/k24/folgen/reverse.cpp
#include<algorithm>
#include<string>
#include<cstring>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
    char s0[] = "Madam";
    reverse(s0, s0 + strlen(s0));
    cout << s0 << endl;    // madaM
    string s1("ABCDEFGH");
    reverse(s1.begin(), s1.end());
    cout << s1 << endl;    // HGFEDCBA
    vector<string> vs;
    vs.push_back("eins");
    vs.push_back("zwei");
    vs.push_back("drei");
    showSequence(vs);      // eins zwei drei
    vector<string> kopie(vs.size());
    reverse_copy(vs.begin(), vs.end(), kopie.begin());
    showSequence(kopie);   // drei zwei eins
}
```

24.3.15 Anzahl der Elemente, die einer Bedingung genügen

Dieser Algorithmus gibt die Anzahl zurück, wie viele Elemente gleich einem bestimmten Wert `value` sind bzw. wie viele Elemente ein bestimmtes Prädikat erfüllen. Die Prototypen sind:

```
template<class InputIterator, class T>
iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);
```

```
template<class InputIterator, class Predicate>
iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);
```

Das Programm zeigt beide Varianten. Die zweite mit `count_if()` benutzt ein unäres Prädikat `ungerade`, das als Funktion formuliert ist.

Listing 24.39: Anzahl bestimmter Elemente

```
// cppbuch/k24/folgen/count.cpp
#include<algorithm>
#include<vector>
#include<showSequence.h>
#include<Random.h>
using namespace std;

bool ungerade(int zahl) {
    return zahl % 2 != 0;
}

int main() {
    vector<int> v;
    Random zufall;
    for(size_t i = 0; i < 20; ++i) {
        v.push_back(zufall(1000));
    }
    showSequence(v);
    int gesucht = 277;
    cout << "Es sind "
         << count(v.begin(), v.end(), gesucht)
         << " Elemente mit dem Wert " << gesucht << " vorhanden." << endl;

    cout << "Es sind "
         << count_if(v.begin(), v.end(), ungerade)
         << " ungerade Elemente vorhanden." << endl;
}
```

24.3.16 Gilt X für alle, keins oder wenigstens ein Element einer Folge?

X steht dabei für ein Prädikat. Die zugehörigen Algorithmen sind neu in den Standard aufgenommen worden.

```
template<class Iterator, class Predicate>
bool all_of(Iterator first, Iterator last, Predicate pred);
```

gibt true zurück, wenn für *alle* Iteratoren i zwischen first und last (ausschließlich) (pred(*i) == true) gilt.

```
template<class Iterator, class Predicate>
bool none_of(Iterator first, Iterator last, Predicate pred);
```

gibt true zurück, wenn für *keinen* Iterator i zwischen first und last (ausschließlich) (pred(*i) == true) gilt.

```
template<class Iterator, class Predicate>
bool any_of(Iterator first, Iterator last, Predicate pred);
```

gibt true zurück, wenn für *wenigstens einen* Iterator i zwischen first und last (ausschließlich) (pred(*i) == true) gilt.

```
// Auszug aus cppbuch/k24/folgen/all_any_none.cpp
struct istPositiv { // zu prüfendes Prädikat
    bool operator()(int x) const {
        return x >= 0;
    }
};

int main() {
    vector<int> folge(12);
    for(size_t i = 0; i < folge.size(); ++i) {
        folge[i] = -i-1; // ggf. je nach Fall verändern
    }
    cout << "Folge = ";
    showSequence(folge);
    if(all_of(folge.begin(), folge.end(), istPositiv())) {
        cout << "Bedingung >=0 gilt für alle Elemente" << endl;
    }
    else {
        cout << "Bedingung >=0 gilt nicht für alle Elemente" << endl;
    }
    if(none_of(folge.begin(), folge.end(), istPositiv())) {
        cout << "Bedingung >= 0 gilt für kein Element" << endl;
    }
    else {
        cout << "Bedingung >=0 gilt für mindestens ein Element" << endl;
    }
    if(any_of(folge.begin(), folge.end(), istPositiv())) {
        cout << "Bedingung >= 0 gilt für wenigstens ein Element" << endl;
    }
    else {
        cout << "Bedingung >=0 gilt für kein Element" << endl;
    }
}
```

24.3.17 Permutationen

Eine Permutation entsteht aus einer Sequenz durch Vertauschung zweier Elemente. (0, 2, 1) ist eine Permutation, die aus (0, 1, 2) entstanden ist. Für eine Sequenz mit N Elementen gibt es $N! = N(N-1)(N-2)\dots 2 \cdot 1$ Permutationen, das heißt $3 \cdot 2 \cdot 1 = 6$ im obigen Beispiel:

(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1, 0)

Man kann sich die Menge aller $N!$ Permutationen einer Sequenz wie oben geordnet vorstellen, sei es, dass die Ordnung mit dem $<$ -Operator oder mit einem Vergleichsobjekt `comp` hergestellt wurde. Aus der Ordnung ergibt sich eine eindeutige Reihenfolge, sodass die nächste oder die vorhergehende Permutation eindeutig bestimmt ist. Dabei wird die Folge zyklisch betrachtet, das heißt, die auf (2, 1, 0) folgende Permutation ist (0, 1, 2). Die Algorithmen `prev_permutation()` und `next_permutation()` verwandeln eine Sequenz in die jeweils vorhergehende bzw. nächste Permutation:

```
template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);
```

```
template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
                     BidirectionalIterator last, Compare comp);
```

```
template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);
```

```
template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last
                     Compare comp);
```

Listing 24.40: Permutationen

```
// cppbuch/k24/folgen/permute.cpp
#include<algorithm>
#include<showSequence.h>
#include<vector>
#include<numeric>
using namespace std;

long fakultaet(unsigned n) { // Fakultät n! berechnen
    long fac = 1;
    while(n > 1) {
        fac *= n--;
    }
    return fac;
}

int main() {
    vector<int> v(4);
    iota(v.begin(), v.end(), 0); // 0 1 2 3
    long anzahl = fakultaet(v.size()); // Anzahl der Permutationen
    for(int i = 0; i < anzahl; ++i) {
        if(!prev_permutation(v.begin(), v.end())) {
            cout << "Zyklusbeginn:\n"; // siehe Text
        }
        showSequence(v);
    }
}
```

Wenn eine Permutation gefunden wird, ist der Rückgabewert `true`. Andernfalls handelt es sich um das Ende eines Zyklus. Dann wird `false` zurückgegeben und die Sequenz in die kleinstmögliche (bei `next_permutation()`) beziehungsweise die größtmögliche (bei `prev_permutation()`) entsprechend dem Sortierkriterium verwandelt. Das Beispiel produziert zuerst die Meldung »Zyklusbeginn«, weil die Vorbesetzung des Vektors mit (0, 1, 2, 3) die Bestimmung einer *vorherigen* Permutation nicht ohne Zyklusüberschreitung erlaubt. Deswegen wird die nach der Sortierung größte Sequenz, nämlich (3, 2, 1, 0), als Nächstes gebildet. Die Meldung »Zyklusbeginn« entfiel, wenn im Beispiel `prev_permutation()` durch `next_permutation()` ersetzt oder wenn alternativ ein Vergleichsobjekt `greater<int>()` als dritter Parameter übergeben würde.

24.3.18 Lexikografischer Vergleich

Der lexikografische¹ Vergleich dient zum Vergleich zweier Sequenzen, die durchaus verschiedene Längen haben können. Die Funktion gibt `true` zurück, wenn die erste Sequenz lexikografisch kleiner ist. Falls eine der beiden Sequenzen bereits vollständig durchsucht ist, ehe ein unterschiedliches Element gefunden wurde, gilt die kürzere Sequenz als kleiner. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, InputIterator2 last2);
```

```
template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, InputIterator2 last2,
                           Compare comp);
```

Damit können z.B. Zeichenketten alphabetisch sortiert werden. Ein Anwendungsbeispiel:

Listing 24.41: Lexikografischer Vergleich

```
// cppbuch/k24/folgen/lexicmp.cpp
#include<algorithm>
#include<iostream>
#include<functional>
using namespace std;

int main () {
    char text1[] = "Arthur";
    int length1 = sizeof(text1);
    char text2[] = "Vera";
    int length2 = sizeof(text2);
    if(lexicographical_compare(text1, text1 + length1, text2, text2 + length2)) {
        cout << text1 << " kommt vor " << text2 << endl;
    }
    else {
        cout << text2 << " kommt vor " << text1 << endl;
    }
    if(lexicographical_compare(text1, text1 + length1, text2, text2 + length2,
                             greater<char>())) { // umgekehrte Reihenfolge
        cout << text1 << " kommt nach " << text2 << endl;
    }
    else {
        cout << text2 << " kommt nach " << text1 << endl;
    }
}
```

¹ Siehe Glossar Seite [953](#)

24.4 Sortieren und Verwandtes

24.4.1 Partitionieren

Eine Sequenz kann mit `partition()` so in zwei Bereiche zerlegt werden, dass alle Elemente, die einem bestimmten Kriterium `pred` genügen, anschließend vor allen anderen liegen. Es wird ein Iterator zurückgegeben, der auf den Anfang des zweiten Bereichs zeigt. Alle vor diesem Iterator liegenden Elemente genügen dem Prädikat. Eine typische Anwendung für eine derartige Zerlegung findet sich im bekannten Quicksort-Algorithmus.

Die zweite Variante `stable_partition()` garantiert darüber hinaus, dass die relative Ordnung der Elemente innerhalb eines Bereichs erhalten bleibt. Diese zweite Variante ist von der Funktion her ausreichend, sodass man die erste normalerweise nicht benötigt. Bei knappem Speicher benötigt die zweite Variante jedoch geringfügig mehr Laufzeit ($O(N \log N)$ statt $O(N)$, $N = \text{last} - \text{first}$), sodass es beide Varianten gibt. Die Prototypen sind:

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                               BidirectionalIterator last,
                               Predicate pred);
```

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(
    BidirectionalIterator first,
    BidirectionalIterator last,
    Predicate pred);
```

Im folgenden Beispiel wird ein Vektor mit negativen und positiven Zahlen erzeugt, die zufällig durcheinandergewürfelt werden. Zwei Kopien des Vektors (`stable` und `unstable` genannt) werden partitioniert, wobei in den Partitionen des Vektors `stable` die ursprüngliche Reihenfolge der Elemente erhalten bleibt.

Listing 24.42: Folge partitionieren

```
// cppbuch/k24/sortieren/partition.cpp
#include<algorithm>
#include<vector>
#include<numeric>
#include<functional>
#include<showSequence.h>
using std::bind;
using namespace std::placeholders;
using namespace std;

int main() {
    vector<int> v(12);
    iota(v.begin(), v.end(), -6);
    random_shuffle(v.begin(), v.end());
    vector<int> unstable = v, stable = v;
    partition(unstable.begin(), unstable.end(), bind(<less<int>(), _1, 0));
    stable_partition(stable.begin(), stable.end(), bind(<less<int>(), _1, 0));
```

```

cout << "In negative und positive Elemente zerlegen\n";
cout << "Sequenz vor der Zerlegung :";
showSequence(v);      // -5 -1 3 2 -3 5 -4 -6 4 0 1 -2
cout << "stabile Partitionierung  :";
showSequence(stable); // -5 -1 -3 -4 -6 -2 3 2 5 4 0 1
cout << "unstabile Partitionierung :";
// Die negativen Elemente sind nicht mehr in ihrer ursprünglichen Reihenfolge
showSequence(unstable); // -5 -1 -2 -6 -3 -4 5 2 4 0 1 3
}

```

24.4.2 Sortieren

Der Algorithmus `sort()` sortiert zwischen den Iteratoren `first` und `last`. Er ist nur für Container mit Random-Access-Iteratoren geeignet, wie zum Beispiel `vector` oder `deque`. Ein wahlfreier Zugriff auf Elemente einer Liste ist nicht möglich, deshalb ist für eine Liste vom Typ `list` die dafür definierte Elementfunktion `list::sort()` zu nehmen.

```

template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

```

```

template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

```

Die Sortierung ist nicht stabil, das heißt, dass verschiedene Elemente, die jedoch denselben Sortierschlüssel haben, in der sortierten Folge nicht unbedingt dieselbe Reihenfolge untereinander wie vorher in der unsortierten Folge haben. Der Aufwand ist im Mittel $O(N \log N)$ mit $N = \text{last} - \text{first}$. Über das Verhalten im schlechtesten Fall (englisch *worst case*) wird keine Aufwandsschätzung gegeben. Falls das Worst-case-Verhalten wichtig ist, wird jedoch empfohlen, lieber `stable_sort()` zu verwenden. Ein Beispiel finden Sie im nächsten Abschnitt.

24.4.3 Stabiles Sortieren

Die Komplexität von `stable_sort()` ist auch im schlechtesten Fall $O(N \log N)$, falls genug Speicher zur Verfügung steht. Andernfalls ist der Aufwand höchstens $O(N(\log N)^2)$. Der Algorithmus basiert intern auf dem Sortieren durch Verschmelzen (*merge sort*, siehe mehr dazu auf Seite 672), das im Durchschnitt um einen konstanten Faktor von etwa 1,4 mehr Zeit als Quicksort benötigt. Dem zeitlichen Mehraufwand von 40 % stehen das sehr gute Verhalten im schlechtesten Fall und die Stabilität von `stable_sort()` gegenüber. Die Deklarationen von `stable_sort()` sind:

```

template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

```

```

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);

```

`sort()` und `stable_sort()` arbeiten mit dem `<`-Operator der Elemente oder einem Comparator-Objekt bzw. einer Funktion zum Vergleich. Im Beispiel werden beide Varianten gezeigt und der Unterschied zwischen stabiler und unstabiler Sortierung demonstriert.

Listing 24.43: Sortieren

```
// cppbuch/k24/sortieren/sort.cpp
#include<algorithm>
#include<vector>
#include<showSequence.h>
#include<Random.h> // von Seite 712
using namespace std;

bool intVergleich(double x, double y) {
    return long(x) < long(y);
}

int main() {
    vector<double> v(17);
    Random zufall;
    // Vektor mit Zufallswerten initialisieren. Dabei sollen einige Werte denselben
    // ganzzahligen Anteil haben, um den Unterschied von stabiler und nicht-stabiler
    // Sortierung sehen zu können.
    for(size_t i = 0; i < v.size(); ++i) {
        v[i] = zufall(v.size()/3)
            + double(zufall(1000)/1000.0);
    }
    vector<double> unstable = v, // Hilfsvektoren
        stable = v;
    cout << "Vektor          :\n";
    showSequence(v);

    // Sortierung mit < Operator:
    stable_sort(stable.begin(), stable.end());
    cout << "Kein Unterschied, weil double-Zahlen als Key genommen werden\n"
        "stabile Sortierung  :\n";
    showSequence(stable);

    sort(unstable.begin(), unstable.end());
    cout << "unstable Sortierung :\n";
    showSequence(unstable);

    // Sortierung mit Funktion statt <
    unstable = v;
    stable = v;
    cout << "Unterschied, weil nur der int-Teil Sortierkriterium ist\n";

    stable_sort(stable.begin(), stable.end(), intVergleich);
    cout << "stabile Sortierung (int-Kriterium) :\n";
    showSequence(stable);

    sort(unstable.begin(), unstable.end(), intVergleich);
    cout << "unstable Sortierung (int-Kriterium) :\n";
    showSequence(unstable);
}
```

24.4.4 Partielles Sortieren

Teilweises Sortieren bringt die M kleinsten Elemente nach vorn, der Rest bleibt unsortiert. Der Algorithmus verlangt jedoch nicht die Zahl M , sondern einen Iterator `middle` auf die entsprechende Position, sodass $M = \text{middle} - \text{first}$ gilt. Die Prototypen sind:

```
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                 RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                 RandomAccessIterator last, Compare comp);
```

Die Komplexität ist etwa $O(N \log M)$. Der Programmauszug für einen Vektor `v` zeigt die teilweise Sortierung. Im Ergebnis sind in der ersten Hälfte alle Elemente kleiner als in der zweiten. In der ersten Hälfte sind sie darüber hinaus sortiert, in der zweiten jedoch nicht.

```
partial_sort(v.begin(), v.begin() + v.size()/2, v.end());
```

Beide Varianten gibt es auch in einer kopierenden Form, wobei `result_first` bzw. `result_last` sich auf den Zielcontainer beziehen. Die Anzahl der sortierten Elemente ergibt sich aus der kleineren der beiden Differenzen `result_last - result_first` bzw. `last - first`.

```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(
    InputIterator first, InputIterator last,
    RandomAccessIterator result_first,
    RandomAccessIterator result_last);
```

```
template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator partial_sort_copy(
    InputIterator first, InputIterator last,
    RandomAccessIterator result_first,
    RandomAccessIterator result_last,
    Compare comp);
```

Der zurückgegebene Random-Access-Iterator zeigt auf das Ende des beschriebenen Bereichs, also auf `result_last` oder auf `result_first + (last - first)`, je nachdem, welcher Wert kleiner ist.

24.4.5 Das n -größte oder n -kleinste Element finden

Das n -größte oder n -kleinste Element einer Sequenz mit Random-Access-Iteratoren kann mit `nth_element()` gefunden werden.

```
template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                RandomAccessIterator last, Compare comp);
```

Der Iterator `nth` wird auf die gewünschte Stelle gesetzt, zum Beispiel auf den Beginn des Containers. Nach Aufruf von `nth_element()` ist das kleinste Element an diese Stelle gerutscht. Die Reihenfolge der Elemente im Container wird also *geändert*. Falls `nth` vor Aufruf zum Beispiel auf die Position `v.begin() + 6` zeigt, steht dort anschließend das siebt kleinste Element.

Nach Aufruf des Algorithmus stehen links von `nth` nur Elemente, die kleiner oder gleich (`*nth`) und allen Elementen rechts davon sind. Der Aufwand des Algorithmus ist im Durchschnitt linear ($O(N)$). Der Aufwand in der vorliegenden Implementierung ist im schlechtesten, wenn auch seltenen Fall $O(N^2)$, weil ein Quicksort-ähnlicher Zerlegungsmechanismus verwendet wird. Das Beispiel ermittelt das kleinste und das größte Element sowie den Medianwert. Der Medianwert teilt eine Folge, sodass die Hälfte der Werte größer gleich und die Hälfte der Werte kleiner gleich dem Medianwert ist.

Listing 24.44: n.-größtes und n.-kleinstes Element finden

```
// cppbuch/k24/sortieren/nth.cpp
#include<algorithm>
#include<deque>
#include<functional>
#include<showSequence.h>
#include<Random.h> // von Seite 712
using namespace std;

int main() {
    deque<int> d;
    Random zufall;
    for(size_t i = 0; i < 15; ++i) {
        d.push_front(zufall(100));
    }
    showSequence(d); // 95 51 36 62 47 55 27 76 33 19 91 79 78 39 84
    deque<int>::iterator nth = d.begin();
    nth_element(d.begin(), nth, d.end());
    cout << "Kleinstes Element: " << (*nth) << endl;           // 19

    // Das Standard-Vergleichsobjekt greater dreht die Reihenfolge um.
    // In diesem Fall steht das größte Objekt an der ersten Position.
    // Hier ist immer noch nth == d.begin().
    nth_element(d.begin(), nth, d.end(), greater<int>());
    cout << "Größtes Element : " << (*nth) << endl;           // 95

    // Mit dem < -Operator steht das größte Element am Ende:
    nth = d.end();
    --nth;           // zeigt nun auf das letzte Element
    nth_element(d.begin(), nth, d.end());
    cout << "Größtes Element : " << (*nth) << endl;           // 95

    // Median
    nth = d.begin() + d.size()/2;
    nth_element(d.begin(), nth, d.end());
    cout << "Medianwert      : " << (*nth) << endl;           // 55
}
```

24.4.6 Verschmelzen (merge)

Verschmelzen, auch Mischen genannt, ist ein Verfahren, zwei sortierte Sequenzen zu einer zu vereinigen. Dabei werden schrittweise die jeweils ersten Elemente beider Sequenzen verglichen, und es wird das kleinere (oder größere, je nach Sortierkriterium) Element in die Ausgabesequenz gepackt. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator2 last2,
                    OutputIterator result);
```

```
template<class InputIterator1, class InputIterator2,
        class OutputIterator, class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator2 last2,
                    OutputIterator result, Compare comp);
```

`merge()` setzt eine vorhandene Ausgabesequenz voraus, die ausreichend Platz haben muss. Falls eine der beiden Eingangssequenzen erschöpft ist, wird der Rest der anderen in die Ausgabe kopiert. Ein kleines Programm soll dies zeigen:

Listing 24.45: Sortieren durch Verschmelzen

```
// cppbuch/k24/sortieren/merge0.cpp
#include<algorithm>
#include<numeric>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
    vector<int> folge1(6);
    iota(folge1.begin(), folge1.end(), 0); // initialisieren
    showSequence(folge1);

    vector<int> folge2(10);
    iota(folge2.begin(), folge2.end(), 0); // initialisieren
    showSequence(folge2);

    // Verschmelzen zweier Folgen v1 und v2, Ablage in result
    vector<int> result(folge1.size()+folge2.size()); // Platz schaffen
    merge(folge1.begin(), folge1.end(),
          folge2.begin(), folge2.end(),
          result.begin());
    showSequence(result);
}
```

Das Ergebnis von *merge0.cpp*:

0 1 2 3 4 5

0 1 2 3 4 5 6 7 8 9

0 0 1 1 2 2 3 3 4 4 5 5 6 7 8 9

Vom Prinzip her erlaubt das Verschmelzen sehr schnelles Sortieren der Komplexität $O(N \log N)$ nach dem rekursiven Schema

1. Teile die Liste in zwei Hälften.
2. Falls die Hälften mehr als ein Element haben, sortiere beide Hälften mit *diesem Verfahren* (Rekursion).
3. Beide Hälften zur Ergebnisliste verschmelzen.

Eine nichtrekursive Variante ist natürlich möglich. Die Sortierung ist stabil. Der Nachteil besteht im notwendigen zusätzlichen Speicher für das Ergebnis. Zum Vergleich mit dem obigen Schema sei der Merge-Sort genannte Algorithmus mit den Mitteln der C++-Standardbibliothek formuliert (ohne dass er selbst dazugehört!):

Listing 24.46: Mergesort

```
// cppbuch/k24/sortieren/mergesort.t
#include<algorithm>

template<class ForwardIterator, class OutputIterator>
void mergesort(ForwardIterator first, ForwardIterator last,
               OutputIterator result) {
    // auto statt typename std::iterator_traits<ForwardIterator>::difference_type
    auto n = std::distance(first, last);
    auto haelfte = n/2;
    ForwardIterator mitte = first;
    std::advance(mitte, haelfte);
    if(haelfte > 1) { // linke Hälfte sortieren, falls notwendig
        mergesort(first, mitte, result); // Rekursion
    }

    if(n - haelfte > 1) { // rechte Hälfte sortieren, falls notwendig
        OutputIterator ergebnis = result;
        std::advance(ergebnis, haelfte);
        mergesort(mitte, last, ergebnis); // Rekursion
    }

    // beide Hälften mischen und zurückkopieren
    OutputIterator end = std::merge(first, mitte, mitte, last, result);
    std::copy(result, end, first);
}
```

Die letzten beiden Anweisungen der Funktion können auf Kosten der Lesbarkeit zusammengefasst werden, wie es oft in der Implementierung der C++-Standardbibliothek zu finden ist:

```
// Beide Hälften verschmelzen und Ergebnis zurückkopieren
copy(result, merge(first, mitte, mitte, last, result), first);
```

Der Vorteil des hier beschriebenen Algorithmus gegenüber `stable_sort()` besteht darin, dass nicht nur Container, die mit Random-Access-Iteratoren zusammenarbeiten, sortiert werden können. Es genügen bidirektionale Iteratoren, sodass `v` im obigen Programm auch eine Liste sein kann. Sie kann mit `push_front()` gefüllt werden. Voraussetzung ist nun, dass eine Liste `puffer` vorhanden ist, die mindestens so viele Elemente wie `v` hat.

Listing 24.47: Mergesort-Anwendung

```
// cppbuch/k24/sortieren/mergesort_list.cpp
#include<algorithm>
#include<list>
#include<showSequence.h>
#include"mergesort.t"
#include<Random.h>
using namespace std;

int main() {    // mit Liste statt Vektor
    list<int> liste;
    Random zufall;
    for(size_t i = 0; i < 20; ++i) {
        liste.push_front(zufall(1000)); // pseudo-zufällige Zahlen
    }
    showSequence(liste);
    list<int> buffer = liste;
    mergesort(liste.begin(), liste.end(), buffer.begin());
    showSequence(liste);    // sortierte Zahlen
}
```

Verschmelzen an Ort und Stelle (inplace_merge)

Wenn Sequenzen an Ort und Stelle gemischt werden sollen, muss der Weg über einen Pufferspeicher gehen. Die Funktion `inplace_merge()` mischt Sequenzen so, dass das Ergebnis an die Stelle der Eingangssequenzen tritt. Die Prototypen sind:

```
template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
                  BidirectionalIterator middle,
                  BidirectionalIterator last);
```

```
template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
                  BidirectionalIterator middle,
                  BidirectionalIterator last,
                  Compare comp);
```

Der Pufferspeicher wird intern und implementationsabhängig bereitgestellt.

Listing 24.48: `inplace_merge()`

```
// cppbuch/k24/sortieren/merge1.cpp
#include<algorithm>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
    vector<int> v(16);           // gerade Zahl
    int middle = v.size()/2;
    for(int i = 0; i < middle; ++i) {
        v[i] = 2*i;             // gerade
    }
}
```

```

        v[middle + i] = 2*i + 1;    // ungerade
    }
    showSequence(v);
    inplace_merge(v.begin(), v.begin() + middle, v.end());
    showSequence(v);
}

```

Die erste Hälfte eines Vektors wird hier mit geraden Zahlen belegt, die zweite mit ungeraden. Nach dem Verschmelzen enthält derselbe Vektor alle Zahlen, ohne dass explizit ein Ergebnisbereich angegeben werden muss:

```

0 2 4 6 8 10 12 14 1 3 5 7 9 11 13 15    vorher
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15    nachher

```

24.5 Suchen und Finden

24.5.1 Element finden

Der Algorithmus `find()` tritt in zwei Arten auf: ohne und mit erforderlichlichem Prädikat (als `find_if()`). Es wird die Position in einem Container gesucht, an der ein bestimmtes Element zu finden ist. Das Ergebnis ist ein Iterator, der auf die gefundene Stelle zeigt oder gleich dem Ende-Iterator `end()` ist. Die Prototypen sind:

```

template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);

```

```

template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);

```

Im folgenden Beispiel wird nur die zweite Variante gezeigt. Es wird die erste ungerade Zahl im Vektor `v` gesucht:

Listing 24.49: Erste ungerade Zahl suchen

```

// cppbuch/k24/finden/find_if.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<showSequence.h>
using namespace std;

class Ungerade {
public:
    bool operator()(int x) const {
        return x % 2 != 0;
    }
};

int main() {

```

```
vector<int> v(8);
for(size_t i = 0; i < v.size(); ++i) {
    v[i] = 2*i;           // nur gerade Zahlen
}
v[5] = 99;               // eine ungerade Zahl
showSequence(v);
// nach ungerader Zahl suchen
auto iter = find_if(v.begin(), v.end(), Ungerade());
if(iter != v.end()) {
    cout << "Die erste ungerade Zahl (" << *iter
        << ") wurde an Position " << (iter - v.begin())
        << " gefunden." << endl;
}
else {
    cout << "Keine ungerade Zahl gefunden." << endl;
}
}
```

Man kann ohne die Funktorklasse `Ungerade` auskommen, wenn man eine der Standardfunktionen benutzt:

```
// Auszug aus cppbuch/k24/finden/find_if2.cpp
vector<int>::iterator iter
    = find_if(v.begin(), v.end(), bind(modulus<int>(), _1, 2));
```

24.5.2 Element einer Menge in der Folge finden

Der Algorithmus findet das erste Auftreten eines Elements einer Menge innerhalb einer Sequenz. Die Prototypen sind:

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of(
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);
```

```
template<class ForwardIterator1, ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_first_of(
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
    BinaryPredicate pred);
```

Das Intervall `[first1, last1)` ist der zu durchsuchende Bereich, das Intervall `[first2, last2)` beschreibt einen Bereich mit zu suchenden Elementen. Zurückgegeben wird der erste Iterator `i` im zu durchsuchenden Bereich, der auf ein Element zeigt, das auch im zweiten Bereich vorhanden ist. Es sei angenommen, dass ein Iterator `j` auf das Element im zweiten Bereich zeigt. Dann gilt

```
*i == *j    beziehungsweise
pred(*i, *j) == true.
```

Falls kein Element aus dem ersten Bereich im zweiten Bereich gefunden wird, gibt der Algorithmus `last1` zurück. Die Komplexität ist $O(N_1 * N_2)$, wenn N_1 und N_2 die Längen der Bereiche sind. Das nachfolgende Programm gibt Folgendes aus:

*Ist eines der Elemente der Menge (1 5 7) in der Folge
0 2 4 6 8 10 12 14 enthalten?*

Nein.

Menge modifiziert:

*Ist eines der Elemente der Menge (1 6 7) in der Folge
0 2 4 6 8 10 12 14 enthalten?*

*Ja. Element 6 ist in beiden Bereichen vorhanden. Sein erstes
Vorkommen in der Folge ist Position 3.*

Listing 24.50: Erstes Auftreten finden

```
// Auszug aus cppbuch/k24/finden/find_first_of.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<showSequence.h>
using namespace std;

int main() {
    vector<int> folge(8);
    vector<int> menge(3);
    // Folge und Menge initialisieren
    for(size_t i = 0; i < folge.size(); ++i) {
        folge[i] = 2*i; // gerade Zahlen
    }
    menge[0] = 1;
    menge[1] = 5;
    menge[2] = 7;
    // Zwei Tests:
    for(int testNumber = 0; testNumber < 2; ++testNumber) {
        if(testNumber == 1) {
            menge[1] = 6;
            cout << endl << "Menge modifiziert:" << endl;
        }
        cout << "Ist eines der Elemente der Menge ("
            showSequence(menge, "");
        cout << ") in der Folge" << endl;
        showSequence(folge, "");
        cout << "enthalten?" << endl;
        // Suche nach Element, das auch in der Menge enthalten ist
        auto iter = find_first_of(folge.begin(), folge.end(),
                                menge.begin(), menge.end());
        if(iter != folge.end()) {
            cout << "Ja. Element " << *iter
                << " ist in beiden Bereichen vorhanden. Sein erstes Vorkommen "
                << " in der Folge ist Position "
                << (iter - folge.begin()) << "." << endl;
        }
        else {
            cout << "Nein." << endl;
        }
    }
}
```

24.5.3 Teilfolge finden

Der Algorithmus `search()` durchsucht eine Sequenz, ob eine zweite Sequenz in ihr enthalten ist. Es wird ein Iterator auf die Position innerhalb der ersten Sequenz zurückgegeben, an der die zweite Sequenz beginnt, sofern sie in der ersten enthalten ist. Andernfalls wird ein Iterator auf die `last1`-Position der ersten Sequenz zurückgegeben. Die Prototypen sind:

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                       ForwardIterator2 first2, ForwardIterator2 last2);
```

```
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                       ForwardIterator2 first2, ForwardIterator2 last2,
                       BinaryPredicate binary_pred);
```

Geht es darum, das letzte Auftreten einer Teilfolge zu finden, bietet sich der Algorithmus `find_end()` an (siehe unten). Ein Beispiel für `search()`:

Listing 24.51: Teilfolge finden[illegible]

```

    if(position != folge.end()) {
        cout << "Die Teilfolge ist in der Folge ab Position "
              << (position - folge.begin())
              << " enthalten." << endl;
    }
    else {
        cout << "Die Teilfolge ist nicht in der Folge enthalten." << endl;
    }
    // Fall2: binäres Prädikat. Dafür negative Zahlen setzen
    for(size_t i = 0; i < teilfolge.size(); ++i) {
        teilfolge[i] = -(i + 5); // -5 -6 -7 -8
    }
    cout << "Teilfolge = ";
    showSequence(teilfolge);

    // Teilfolge in Folge suchen, dabei Vorzeichen ignorieren
    position = search(folge.begin(), folge.end(),
                     teilfolge.begin(), teilfolge.end(),
                     Absolutvergleich());
    if(position != folge.end()) {
        cout << "Die Teilfolge ist in der Folge ab Position "
              << (position - folge.begin())
              << " enthalten. Vorzeichen wurden ignoriert" << endl;
    }
    else {
        cout << "Die Teilfolge ist nicht in der Folge enthalten." << endl;
    }
}

```

Letztes Auftreten einer Teilfolge finden

Der Algorithmus findet das letzte Auftreten einer Teilfolge innerhalb einer Sequenz. Die Prototypen sind:

```

template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2);

```

```

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2,
                        BinaryPredicate pred);

```

Das Intervall `[first1, last1)` ist der zu durchsuchende Bereich, das Intervall `[first2, last2)` beschreibt die zu suchende Folge. Zurückgegeben wird der letzte Iterator im zu durchsuchenden Bereich, der auf den Beginn der Teilfolge zeigt. Falls die Teilfolge nicht gefunden wird, gibt der Algorithmus `last1` zurück. Falls der zurückgegebene Iterator mit `i` bezeichnet wird, gilt

```

*(i+n) == *(first2+n)    beziehungsweise
pred(*(i+n), *(first2+n)) == true

```

für alle n im Bereich 0 bis $(\text{last2} - \text{first2})$. Die Komplexität ist $O(N_2(N_1 - N_2))$, wenn N_1 und N_2 die Länge des zu durchsuchenden Bereichs bzw. der zu suchenden Teilfolge sind. Die Benutzung entspricht der von `search()`, sodass hier nur ein Auszug eines Beispiels gezeigt wird:

```
// Auszug aus cppbuch/k24/finden/find_end.cpp
vector<int>::const_iterator iter
    = find_end(folge.begin(), folge.end(), teilfolge1.begin(), teilfolge1.end());
```

24.5.4 Bestimmte benachbarte Elemente finden

Zwei gleiche, direkt benachbarte (englisch *adjacent*) Elemente werden mit der Funktion `adjacent_find()` gefunden. Es gibt auch hier zwei überladene Varianten – eine ohne und eine mit binärem Prädikat. Die erste Variante vergleicht die Elemente mit dem Gleichheitsoperator `==`, die zweite benutzt das Prädikat. Die Prototypen sind:

```
template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);
```

```
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                             BinaryPredicate binary_pred);
```

Der zurückgegebene Iterator zeigt auf das erste der beiden Elemente, sofern ein entsprechendes Paar gefunden wird. Beispiel:

Listing 24.52: Gleiche benachbarte Elemente finden

```
// cppbuch/k24/finden/adjacent_find.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<showSequence.h>
using namespace std;

int main() {
    vector<int> v(8);
    for(size_t i = 0; i < v.size(); ++i) {
        v[i] = 2*i;          // gerade
    }
    v[5] = 99;              // 2 gleiche benachbarte Elemente
    v[6] = 99;
    showSequence(v);
    auto iter = adjacent_find(v.begin(), v.end()); // finde gleiche Nachbarn
    if(iter != v.end()) {
        cout << "Die ersten gleichen benachbarten Zahlen ("
              << *iter << ") wurden an Position "
              << (iter - v.begin()) << " gefunden." << endl;
    }
    else {
        cout << "Keine gleichen Nachbarn gefunden." << endl;
    }
}
```

Wenn zum Beispiel nach einem doppelt so großen Nachfolger gesucht werden soll, kommt ein binäres Prädikat zum Einsatz, zum Beispiel

```
class IstDoppeltSoGross {
public:
    bool operator()(int a, int b) const { return (b == 2*a); }
};
```

Der entsprechende Aufruf, um den doppelt so großen Nachfolger zu finden, lautet:

```
// Auszug aus cppbuch/k24/finden/adjacent_find_1.cpp
auto iter = adjacent_find(v.begin(), v.end(), IstDoppeltSoGross());
```

Die Auswertung kann wie oben mit der Iterator-Differenz (`iter - v.begin()`) geschehen.

24.5.5 Bestimmte aufeinanderfolgende Werte finden

Der Algorithmus `search_n()` durchsucht eine Sequenz daraufhin, ob eine Folge von gleichen Werten in ihr enthalten ist. Die Prototypen sind:

```
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                        Size count, const T& value);
```

```
template<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                        Size count, const T& value,
                        BinaryPredicate binary_pred);
```

Zurückgegeben wird von der ersten Funktion der Iterator auf den Beginn der ersten Folge mit wenigstens `count` Werten, die gleich `value` sind. Falls eine derartige Folge nicht gefunden wird, gibt die Funktion `last` zurück. Die zweite Funktion prüft nicht auf Gleichheit, sondern wertet das binäre Prädikat aus. Im Erfolgsfall muss für wenigstens `count` aufeinanderfolgende Werte `x` das Prädikat `binary_pred(x, value)` gelten. Im Beispiel wird erst gesucht, ob der Wert 4 mindestens 3-mal nacheinander auftritt. Dann wird gesucht, ob es mindestens 3-mal nacheinander Werte gibt, die größer als 4 sind:

Listing 24.53: Aufeinanderfolgende Werte suchen, die einer Bedingung genügen

```
// Auszug aus cppbuch/k24/finden/search_n.cpp
#include<algorithm>
#include<vector>
#include<iostream>
#include<showSequence.h>
using namespace std;

int main() {
    vector<int> folge(12);
    for(size_t i = 0; i < folge.size(); ++i) {
        folge[i] = i; // 0 1 2 3 4 5 6 7 8 9 10 11
    }
    folge[6] = folge[5] = folge[4]; // aufeinanderfolgende gleiche Werte
    cout << "Folge = ";
    showSequence(folge);
```

```

int wert = 4;
int wieoft = 3;
auto position = search_n(folge.begin(), folge.end(), wieoft, wert);

if(position != folge.end()) {
    cout << wert << " wurde " << wieoft << "-mal nacheinander ab Position "
        << (position - folge.begin()) << " gefunden." << endl;
}
else {
    cout << wert << " wurde nicht " << wieoft
        << "-mal nacheinander gefunden." << endl;
}
position = search_n(folge.begin(), folge.end(), wieoft, wert, greater<int>());
if(position != folge.end()) {
    cout << "Ab Position " << (position - folge.begin())
        << " wurden " << wieoft << "-mal nacheinander Werte größer als "
        << wert << " gefunden." << endl;
}
else {
    cout << wieoft << "-mal nacheinander Werte größer als "
        << wert << " sind nicht vorhanden." << endl;
}
}

```

24.5.6 Binäre Suche

Alle Algorithmen dieses Abschnitts sind Variationen der binären Suche. Wenn ein wahlfreier Zugriff mit einem Random-Access-Iterator auf eine sortierte Folge mit n Elementen möglich ist, ist die binäre Suche sehr schnell. Es werden maximal $1 + \log_2 n$ Zugriffe benötigt, um das Element zu finden, oder festzustellen, dass es nicht vorhanden ist. Falls ein wahlfreier Zugriff nicht möglich ist wie zum Beispiel bei einer Liste, in der man sich von Element zu Element hangeln muss, um ein bestimmtes Element zu finden, ist die Zugriffszeit von der Ordnung $O(n)$. Die C++-Standardbibliothek stellt vier Algorithmen bereit, die im Zusammenhang mit dem Suchen und Einfügen in sortierte Folgen sinnvoll sind und sich algorithmisch sehr ähneln:

binary_search

```

template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value);

```

```

template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value, Compare comp);

```

Dies ist die eigentliche binäre Suche. Die Funktion gibt `true` zurück, falls der Wert `value` gefunden wird. In der ersten Variante wird der Operator `<` benutzt, indem die Beziehung

```

(!(*i < value) && !(value < *i))

```

betrachtet wird (Äquivalenz). i ist ein Iterator im Bereich $[first, last)$. Falls ein Bereich mit einem Vergleichsobjekt `comp` sortiert wurde, muss dies bei der Suche berücksichtigt werden (zweite Variante). Es wird entsprechend

```
(!comp(*i, value) && !comp(value, *i))
```

ausgewertet. Die Äquivalenzbeziehung entspricht nicht in jedem Fall der Gleichheit. Beim Vergleich ganzer Zahlen mit dem `<`-Operator fallen beide Begriffe zusammen. Aber: Im Duden werden Umlaute bei der Sortierung wie Selbstlaute behandelt. Die Wörter Mücke und Mücke stehen beide vor dem Wort mucken. Obwohl ungleich, sind sie doch äquivalent bezüglich der Sortierung. Ein Beispiel für `binary_search()` wird nach Vorstellung der nächsten drei Algorithmen gezeigt.

lower_bound

Dieser Algorithmus findet die erste Stelle, an der ein Wert `value` eingefügt werden kann, ohne die Sortierung zu stören. Der zurückgegebene Iterator, er sei hier i genannt, zeigt auf diese Stelle, sodass ein Einfügen ohne weitere Suchvorgänge mit `insert(i, value)` möglich ist. Für alle Iteratoren j im Bereich $[first, i)$ gilt, dass $*j < value$ ist bzw. `comp(*j, value) == true`. Die Prototypen sind:

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                           const T& value);
```

```
template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                           const T& value, Compare comp);
```

upper_bound

Dieser Algorithmus findet die *letzte* Stelle, an der ein Wert `value` eingefügt werden kann, ohne die Sortierung zu stören. Der zurückgegebene Iterator i zeigt auf diese Stelle, sodass ein schnelles Einfügen mit `insert(i, value)` möglich ist. Die Prototypen sind:

```
template<class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                           const T& value);
```

```
template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                           const T& value, Compare comp);
```

equal_range

Dieser Algorithmus ermittelt den größtmöglichen Bereich, innerhalb dessen an jeder beliebigen Stelle ein Wert `value` eingefügt werden kann, ohne die Sortierung zu stören. Bezüglich der Sortierung enthält dieser Bereich also äquivalente Werte. Die Elemente `p.first` und `p.second` des zurückgegebenen Iteratorpaars, hier p genannt, begrenzen den Bereich. Für jeden Iterator k , der die Bedingung $p.first \leq k < p.second$ erfüllt, ist schnelles Einfügen mit `insert(k, value)` möglich. Die Prototypen sind:

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
            const T& value);
```

```
template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
            const T& value, Compare comp);
```

Die beschriebenen Algorithmen werden anhand eines Beispielprogramms demonstriert, wobei `upper_bound()` wegen seiner Ähnlichkeit mit `lower_bound()` nicht aufgeführt ist. Die Sortierung des Containers muss gewährleistet sein, weil alle Algorithmen dieses Abschnitts dies voraussetzen.

Listing 24.54: `binary_search()` und Verwandte

```
// cppbuch/k24/finden/binarysearch.cpp
#include<algorithm>
#include<list>
#include<string>
#include<showSequence.h>
using namespace std;

int main() {
    list<string> staedte;
    staedte.push_front("Bremen");
    staedte.push_front("Paris");
    staedte.push_front("Mailand");
    staedte.push_front("Hamburg");
    staedte.sort();           // Wichtige Vorbedingung!
    showSequence(staedte);
    string stadt;
    cout << "Welche Stadt eintragen/suchen? ";
    cin >> stadt;
    if(binary_search(staedte.begin(), staedte.end(), stadt)) {
        cout << stadt << " existiert.\n";
    }
    else {
        cout << stadt << " existiert noch nicht.\n";
    }
    // an der richtigen Stelle einfügen
    cout << stadt << " wird eingefügt:\n";
    auto i = lower_bound(staedte.begin(), staedte.end(), stadt);
    staedte.insert(i, stadt);
    showSequence(staedte);
    // Bereich gleicher Werte
    auto p = equal_range(staedte.begin(), staedte.end(), stadt);

    // Die zwei Iteratoren des Pairs p begrenzen den Bereich, in dem stadt vorkommt.
    auto n = distance(p.first, p.second);
    cout << stadt << " ist " << n << " mal in der Liste enthalten\n";
}
```


24.6 Mengenoperationen auf sortierten Strukturen

Die folgenden Algorithmen beschreiben die grundlegenden Mengenoperationen wie Vereinigung, Durchschnitt usw. auf *sortierten* Strukturen. In der C++-Standardbibliothek basiert auch die Klasse `set` auf sortierten Strukturen (siehe Abschnitt 28.3.3). Die Komplexität der Algorithmen ist $O(N_1 + N_2)$, wobei N_1 und N_2 die jeweilige Anzahl der Elemente der beteiligten Mengen sind. Mengenoperationen auf sortierten Strukturen sind nur unter bestimmten Bedingungen sinnvoll, und es sind Randbedingungen zu beachten.

- Standard-Container aus Kapitel 28: `vector`, `list`, `deque`
 - Der Ergebniscontainer bietet ausreichend Platz. Nachteil: Nach dem Ende der Ergebnissequenz stehen noch alte Werte im Container, falls der Platz mehr als genau ausreichend ist.
 - Der Output-Iterator darf nicht identisch mit `v1.begin()` oder `v2.begin()` sein. `v1` und `v2` sind die zu verknüpfenden Mengen.
 - Der Ergebniscontainer ist leer. In diesem Fall ist ein Insert-Iterator als Output-Iterator zu nehmen.
- Assoziative Container: `set`, `map`

Grundsätzlich ist ein Insert-Iterator zu nehmen. Der Inhalt eines Elements darf nicht direkt, das heißt über eine Referenz auf das Element, geändert werden. So würde sich ein nicht einfügender Output-Iterator verhalten, und die Sortierung innerhalb des Containers und damit seine Integrität würde verletzt.

24.6.1 Teilmengenrelation

Die Funktion `includes` gibt an, ob jedes Element einer zweiten sortierten Struktur S_2 in der ersten Struktur S_1 enthalten ist, also eine Teilmenge der ersten ist. Der Rückgabewert ist `true`, falls $S_2 \subseteq S_1$ gilt, ansonsten `false`. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);
```

```
template<class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              Compare comp);
```

Das folgende Beispiel initialisiert einige `set`-Objekte als sortierte Strukturen. Man kann an deren Stelle natürlich auch schlichte Vektoren nehmen, vorausgesetzt, sie sind sortiert. Weil das Beispiel in den weiteren Abschnitten aufgegriffen wird, enthält es bereits hier mehr, als für `includes()` notwendig ist.

Listing 24.55: Operationen auf Mengen

```
// Auszug aus cppbuch/k24/mengen/set_algorithmen.cpp
#include<algorithm>
```

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

int main () {
    int v1[] = {1, 2, 3, 4};
    int v2[] = {0, 1, 2, 3, 4, 5, 7, 99, 13};
    int v3[] = {-2, 5, 12, 7, 33};
    // Weiter unten benötigte Sets mit den Vektorinhalten initialisieren.
    // Voreingestelltes Vergleichsobjekt: less<int>()
    // (implizite automatische Sortierung)
    // sizeof v/sizeof *v1 ist die Anzahl der Elemente in v
    set<int> s1(v1, v1 + sizeof v1/sizeof *v1);
    set<int> s2(v2, v2 + sizeof v2/sizeof *v2);
    set<int> s3(v3, v3 + sizeof v3/sizeof *v3);
    set<int> result; // leere Menge (s1, s2, s3 wie oben)
    if (includes(s2.begin(), s2.end(), s1.begin(), s1.end())) {
        showSequence(s1); // 1 2 3 4
        cout << " ist eine Teilmenge von ";
        showSequence(s2); // 0 1 2 3 4 5 7 99
    } // Fortsetzung nächster Abschnitt
```

24.6.2 Vereinigung

Die Funktion `set_union` bildet eine sortierte Struktur, in der alle Elemente enthalten sind, die in wenigstens einer von zwei anderen sortierten Strukturen S_1 und S_2 vorkommen. Es wird die Vereinigung beider Strukturen gebildet:

$$S = S_1 \cup S_2$$

Voraussetzung ist, dass die aufnehmende Struktur genügend Platz bietet oder dass sie leer ist und ein Insert-Iterator als Output-Iterator verwendet wird. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result);
```

```
template<class InputIterator1, class InputIterator2,
        class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result, Compare comp);
```

Die Ergebnismenge `result` (siehe unten) ist anfangs leer. Im nachfolgenden Beispiel muss der Output-Iterator ein Insert-Iterator sein. Dazu wird die Funktion `inserter()`, die auf Seite 813 beschrieben ist, in der Parameterliste aufgeführt. Sie gibt einen Insert-Iterator zurück. Nur `result.begin()` als Output-Iterator zu verwenden, führt zu Fehlern.

```
set<int> result; // leere Menge (s1, s2, s3 seien wie oben)
set_union(s1.begin(), s1.end(), s3.begin(), s3.end(),
          inserter(result, result.begin()));
showSequence(s1); // 1 2 3 4
cout << " vereinigt mit ";
showSequence(s3); // -2 5 7 12 33
```

```
cout << " ergibt ";
showSequence(result);           // -2 1 2 3 4 5 7 12 33
```

24.6.3 Schnittmenge

Die Funktion `set_intersection` bildet eine sortierte Struktur, in der alle Elemente enthalten sind, die sowohl in der einen als auch in der anderen von zwei sortierten Strukturen S_1 und S_2 vorkommen. Es wird die Schnittmenge oder der Durchschnitt beider Strukturen gebildet:

$$S = S_1 \cap S_2$$

Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                               InputIterator2 first2, InputIterator2 last2,
                               OutputIterator result);
```

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                               InputIterator2 first2, InputIterator2 last2,
                               OutputIterator result,
                               Compare comp);
```

Um die alten Ergebnisse zu löschen, wird `clear()` aufgerufen. Andernfalls würden sie mit ausgegeben.

```
// Fortsetzung des Beispiels von oben
// Zuerst Menge leeren:
result.clear();
// Durchschnitt bilden:
set_intersection(s2.begin(), s2.end(),
                 s3.begin(), s3.end(),
                 inserter(result, result.begin()));
showSequence(s2);           // 0 1 2 3 4 5 7 99
cout << " geschnitten mit ";
showSequence(s3);           // -2 5 7 12 33
cout << " ergibt ";
showSequence(result);       // 5 7
```

24.6.4 Differenz

Die Funktion `set_difference` bildet eine sortierte Struktur, in der alle Elemente enthalten sind, die in der ersten Struktur S_1 , aber nicht in einer zweiten sortierten Struktur S_2 vorkommen. Es wird die Differenz $S_1 - S_2$ beider Strukturen gebildet, auch als $S_1 \setminus S_2$ geschrieben. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result);
```

```
template<class InputIterator1, class InputIterator2,
        class OutputIterator, class Compare>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, InputIterator2 last2,
                           OutputIterator result,
                           Compare comp);
```

Das Beispiel folgt dem obigen Muster:

```
result.clear();
set_difference(s2.begin(), s2.end(),
              s1.begin(), s1.end(),
              inserter(result, result.begin()));
showSequence(s2);           // 0 1 2 3 4 5 7 99
cout << " minus ";
showSequence(s1);           // 1 2 3 4
cout << " ergibt ";
showSequence(result);       // 0 5 7 99
```

24.6.5 Symmetrische Differenz

Die Funktion `set_symmetric_difference` bildet eine sortierte Struktur, in der alle Elemente enthalten sind, die entweder in der ersten Struktur S_1 oder in einer zweiten sortierten Struktur S_2 vorkommen, aber nicht in beiden. Es wird die symmetrische Differenz beider Strukturen gebildet, auch als Exklusiv-Oder bezeichnet. Mit den vorangegangenen Operationen kann die symmetrische Differenz ausgedrückt werden:

$$S = (S_1 - S_2) \cup (S_2 - S_1)$$

oder

$$S = (S_1 \cup S_2) - (S_2 \cap S_1)$$

Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
```

```
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);
```

Das letzte Beispiel dieser Art zeigt die symmetrische Differenz:

```
result.clear();
set_symmetric_difference(s2.begin(), s2.end(),
                        s3.begin(), s3.end(),
                        inserter(result, result.begin()));
showSequence(s2);           // 0 1 2 3 4 5 7 99
cout << " exklusiv oder ";
```

```

showSequence(s3);           // -2 5 7 12 33
cout << " ergibt ";
showSequence(result);       // -2 0 1 2 3 4 12 33 99
} // Ende von cppbuch/k24/mengen/set_algorithmen.cpp

```

24.7 Heap-Algorithmen

Die in Abschnitt 28.2.7 beschriebene Priority-Queue basiert auf einem binären Heap (englisch für Haufen oder Halde). Vor der Beschreibung der Heap-Algorithmen seien kurz die wichtigsten Eigenschaften der Datenstruktur Heap (nicht zu verwechseln mit dem ebenfalls Heap genannten Freispeicher) charakterisiert:

- Die N Elemente eines Heaps liegen in einem kontinuierlichen Array auf den Positionen 0 bis $N - 1$. Es wird vorausgesetzt, dass ein wahlfreier Zugriff möglich ist (Random-Access-Iterator).
- Die Art der Anordnung der Elemente im Array entspricht einem vollständigen binären Baum, bei dem alle Ebenen mit Elementen besetzt sind. Die einzig mögliche Ausnahme bildet die unterste Ebene, in der alle Elemente auf der linken Seite erscheinen. Abbildung 24.1 zeigt die Array-Repräsentation eines Heaps H mit 14 Elementen, wobei die Zahlen in den Kreisen die Array-Indizes darstellen (*nicht* die Elementwerte). Das Element $H[0]$ ist also stets die Wurzel, und jedes Element $H[j]; (j > 0)$ hat einen Elternknoten $H[(j - 1)/2]$.

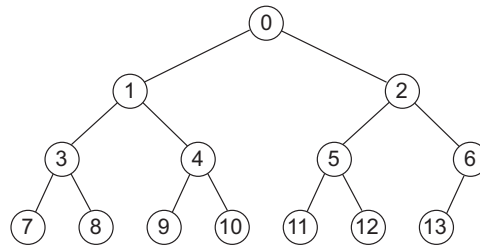


Abbildung 24.1: Array-Repräsentation eines Heaps (Zahl = Array-Index)

- Jedem Element $H[j]$ ist eine Priorität zugeordnet, die größer oder gleich der Priorität der Kindknoten $H[2j + 1]$ und $H[2j + 2]$ ist. Hier und im Folgenden sei zur Vereinfachung angenommen, dass große Zahlen hohe Prioritäten bedeuten. Im Allgemeinen kann es auch umgekehrt sein oder es können gänzlich andere Kriterien die Priorität bestimmen. Abbildung 24.2 zeigt beispielhafte *Elementwerte* eines Heaps: $H[0]$ ist gleich 98 usw.

Beachten Sie, dass der Heap nicht vollständig sortiert ist, sondern dass es nur auf die Prioritätsrelation zwischen Eltern- und zugehörigen Kindknoten ankommt.

Ein Array H mit N Elementen ist genau dann ein Heap, wenn $H[(j - 1)/2] \geq H[j]$ für $1 \leq j < N$ gilt. Daraus folgt automatisch, dass $H[0]$ das größte Element ist. Eine Priority-

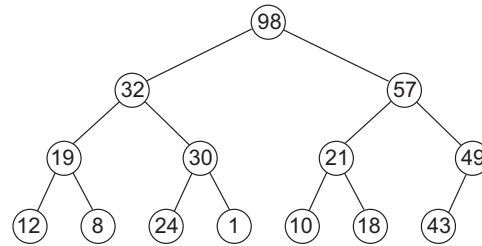


Abbildung 24.2: Array-Repräsentation eines Heaps (Zahl = Elementwert)

Queue entnimmt einfach immer das oberste Element eines Heaps. Anschließend wird er restrukturiert, das heißt, das nächstgrößte Element wandert an die Spitze. Bezogen auf die Abbildungen 24.1 und 24.2 wäre dies das Element Nr. 2 mit dem Wert 57.

Die C++-Standardbibliothek bietet vier Heap-Algorithmen an, die auf alle Container, auf die mit Random-Access-Iteratoren zugegriffen werden kann, anwendbar sind.

- `pop_heap()` entfernt das Element mit der höchsten Priorität.
- `push_heap()` fügt ein Element einem vorhandenen Heap hinzu.
- `make_heap()` arrangiert alle Elemente innerhalb eines Bereichs, sodass dieser Bereich einen Heap darstellt.
- `sort_heap()` verwandelt einen Heap in eine sortierte Folge.

Diese Algorithmen müssen keine Einzelheiten über die Container wissen. Ihnen werden lediglich zwei Iteratoren übergeben, die den zu bearbeitenden Bereich markieren. Zwar ist `less<T>` als Prioritätskriterium vorgegeben, aber vielleicht wird ein anderes Kriterium gewünscht. Daher gibt es für jeden Algorithmus eine überladene Variante, welche die Übergabe eines Vergleichsobjekts erlaubt.

24.7.1 pop_heap

Die Funktion `pop_heap()` entnimmt ein Element aus einem Heap. Der Bereich `[first, last)` sei dabei ein gültiger Heap. Die Prototypen sind:

```
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

Die Entnahme besteht nur darin, dass der Wert mit der höchsten Priorität, der an der Stelle `first` steht, mit dem Wert an der Stelle `(last - 1)` vertauscht wird. Anschließend wird der Bereich `[first, last-1)` in einen Heap verwandelt. Die Komplexität von `pop_heap()` ist $O(\log(last - first))$. Anwendung für einen Vektor `v`:

Listing 24.56: Heap-Operationen

```
// Auszug aus cppbuch/k24/heap/heap.cpp
// gültigen Heap erzeugen
make_heap(v.begin(), v.end()); // Seite 691
// Die beiden Zahlen mit der höchsten Priorität anzeigen und entnehmen:
```

```
vector<int>::iterator last = v.end();
cout << *v.begin() << endl;
pop_heap(v.begin(), last--);
cout << *v.begin() << endl;
pop_heap(v.begin(), last--);
```



Hinweis

Bitte beachten Sie, dass nicht mehr `v.end()` das Heap-Ende anzeigt, sondern der Iterator `last`. Der Bereich dazwischen ist bezüglich der Heap-Eigenschaften von `v` *undefiniert*.

24.7.2 push_heap

Die Funktion `push_heap()` fügt ein Element einem vorhandenen Heap hinzu. Wie die Prototypen zeigen, werden der Funktion nur zwei Iteratoren und gegebenenfalls ein Vergleichsobjekt übergeben. Das einzufügende Element tritt hier nicht auf:

```
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

Es muss die Vorbedingung gelten, dass der Bereich `[first, last-1)` ein gültiger Heap ist. `push_heap()` kümmert sich nicht selbst um den einzutragenden Wert. An die Stelle (`last`) wird deshalb *vorher* der auf den Heap abzulegende Wert eingetragen. Der anschließende Aufruf von `push_heap(first, ++last)` sorgt dafür, dass nach dem Aufruf der Bereich `[first, last)` ein Heap ist. Die Funktion ist etwas umständlich zu bedienen, aber sie ist auch nur als Hilfsfunktion gedacht und sehr schnell. Die Komplexität von `push_heap()` ist $O(\log(last - first))$. In den Beispiel-Heap werden nun zwei Zahlen wie beschrieben eingefügt (das vorhergehende Beispiel wird fortgesetzt):

```
// eine »wichtige Zahl« (99) eintragen
*last = 99;
push_heap(v.begin(), ++last);
// eine »unwichtige Zahl« (-1) eintragen
*last = -1;
push_heap(v.begin(), ++last);
```

Beim Einfügen muss beachtet werden, dass `last` nicht über `v.end()` hinausläuft. Durch die Tatsache, dass bei der Entnahme immer der Wert mit der höchsten Priorität an die Spitze gesetzt wird, ist die Ausgabe sortiert:

```
// Ausgabe und Entfernen aller Zahlen der Priorität nach:
while(last != v.begin()) {
    cout << *v.begin() << ' ';
    pop_heap(v.begin(), last--);
}
```

24.7.3 make_heap

`make_heap()` sorgt dafür, dass die Heap-Bedingung für alle Elemente innerhalb eines Bereichs gilt. Die Prototypen sind:

```
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

Die Komplexität ist proportional zur Anzahl der Elemente zwischen `first` und `last`. Das Beispiel von oben zeigt die Anwendung auf einem Vektor als Container:

```
make_heap(v.begin(), v.end());
```

24.7.4 sort_heap

`sort_heap()` verwandelt einen Heap in eine sortierte Sequenz. Die Sortierung ist nicht stabil, die Komplexität ist $O(N\log N)$, wenn N die Anzahl der zu sortierenden Elemente ist. Die Prototypen sind:

```
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

Die Sequenz ist *aufsteigend* sortiert. Damit ist gemeint, dass die Elemente hoher Priorität *an das Ende* der Sequenz kommen:

```
// neuen gültigen Heap aus allen Elementen erzeugen
make_heap(v.begin(), v.end());
// und sortieren
sort_heap(v.begin(), v.end());
```



Übung

24.2 Gegeben sei die Template-Klasse `Heap` mit den folgenden Schnittstellen:

```
template<typename T, class Compare = std::less<T> >
class Heap {
public:
    Heap(const Compare& cmp = Compare());
    void push(const T& t);
    void pop();
    const T& top() const;
    bool empty() const;
    vector<T> toSortedVector() const;
};
```

Die letzte Methode gibt den Heap-Inhalt als sortierten Vektor zurück. Implementieren Sie die Klasse unter Verwendung einiger Algorithmen des Abschnitts 24.7. Testen Sie die

Klasse, indem Sie das `priority_queue`-Objekt in der Lösung von Aufgabe 28.2 durch ein gleichnamiges `Heap`-Objekt ersetzen.

24.7.5 is_heap

`is_heap()` gibt zurück, ob ein Bereich den Heap-Kriterien genügt. Gegebenenfalls kann ein Vergleichsobjekt übergeben werden.

```
template<class RandomAccessIterator>
bool is_heap(RandomAccessIterator first,
             RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
bool is_heap(RandomAccessIterator first,
             RandomAccessIterator last,
             Compare comp);
```

Ergänzend dazu gibt es den Algorithmus `is_heap_until(first, last)`, der den letzten Iterator im Bereich `[first, last)` zurückgibt, bis zu dem der Bereich als Heap angesehen werden kann.

24.8 Vergleich von Containern auch ungleichen Typs

24.8.1 Unterschiedliche Elemente finden

`mismatch()` überprüft zwei Container auf Übereinstimmung ihres Inhalts, wobei eine Variante ein binäres Prädikat benutzt. Die Prototypen sind:

```
template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2> mismatch(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2);
```

```
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2> mismatch(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    BinaryPredicate binary_pred);
```

Der Algorithmus gibt ein Paar von Iteratoren zurück, die auf die erste Stelle der Nicht-übereinstimmung in den jeweiligen korrespondierenden Containern zeigen. Falls beide Container übereinstimmen, ist der erste Iterator des zurückgegebenen Paares gleich `last1`. Die Container müssen nicht vom selben Typ sein. In dem Beispiel wird ein Vektor `v` mit einem Set `s` verglichen:

Listing 24.57: Prüfung auf (Nicht-)Übereinstimmung

```
// cppbuch/k24/vergleich/mismatch.cpp
#include<algorithm>
#include<vector>
#include<set>
#include<showSequence.h>
using namespace std;

int main() {
    vector<int> v(8);
    for(size_t i = 0; i < v.size(); ++i) {
        v[i] = 2*i;           // sortierte Folge
    }
    set<int> s(v.begin(), v.end()); // Set mit v initialisieren
    v[3] = 7;                 // Nichtübereinstimmung erzeugen
    showSequence(v);          // Anzeige
    showSequence(s);
    // Prüfung mit Iterator-Paar 'wo'
    // auto = pair<vector<int>::iterator, set<int>::iterator>
    auto wo = mismatch(v.begin(), v.end(), s.begin());
    if(wo.first == v.end()) {
        cout << "Inhalt der Container stimmt überein." << endl;
    }
    else {
        cout << "Der erste Unterschied (" << *wo.first << " != "
              << *wo.second << ") wurde an Position "
              << (wo.first - v.begin()) << " gefunden." << endl;
    }
}
```

Eine indexartige Position ist in einem Set nicht definiert, deswegen ist ein Ausdruck der Art `(wo.second - s.begin())` ungültig. Zwar zeigt `wo.second` auf die Stelle der Nichtübereinstimmung in `s`, aber die Arithmetik ist nicht erlaubt. Wenn man die relative Nummer bezüglich des ersten Elements in `s` unbedingt benötigen sollte, kann man `distance()` verwenden.

Die Variante mit dem binären Prädikat ist geeignet, wenn es um nicht-exakte Übereinstimmung geht. Wenn etwa berechnete Resultate, die mit Rundungsfehlern behaftet sind, verglichen werden sollen, genügt es, wenn die Ergebnisse innerhalb einer gewissen Genauigkeit übereinstimmen. Im folgenden Beispiel werden zwei nicht genau gleiche Container als übereinstimmend betrachtet. Wenn der Schwellenwert 0.01 auf z.B. 0.005 abgesenkt wird, werden Unterschiede angezeigt.

Listing 24.58: Prüfung mit binärem Prädikat

```
// cppbuch/k24/vergleich/mismatch_b.cpp
#include<algorithm>
#include<vector>
#include<cmath> // fabs
#include<showSequence.h>
#include<Random.h>
using namespace std;
```

```

class VergleichMitToleranz {
public:
    VergleichMitToleranz(double e)
        : eps(e) {
    }
    bool operator()(double x, double y) {
        return fabs(x-y) < eps;
    }
private:
    double eps;
};

int main() {
    vector<double> v1(8), v2(8);
    Random zufall;
    for(size_t i = 0; i < v1.size(); ++i) {
        v1[i] = i + zufall(100)/10000.0;
        v2[i] = i + zufall(100)/10000.0;
    }
    showSequence(v1);           // Anzeige
    showSequence(v2);

    // Prüfung mit Iterator-Paar 'wo'
    // auto = pair<vector<double>::iterator, vector<double>::iterator>
    auto wo = mismatch(v1.begin(), v1.end(), v2.begin(),
                      VergleichMitToleranz(0.01));
    if(wo.first == v1.end()) {
        cout << "Inhalt der Container stimmt innerhalb der Toleranz überein."
              << endl;
    }
    else {
        cout << "Der erste Unterschied (" << *wo.first << " != "
              << *wo.second << ") wurde an Position "
              << (wo.first - v1.begin()) << " gefunden." << endl;
    }
}

```

24.8.2 Prüfung auf gleiche Inhalte

`equal()` überprüft zwei Container auf Übereinstimmung ihres Inhalts, wobei eine Variante ein binäres Prädikat benutzt. Im Unterschied zu `mismatch()` wird jedoch kein Hinweis auf die Position gegeben. Wie am Rückgabetyt `bool` erkennbar, wird nur festgestellt, ob die Übereinstimmung besteht oder nicht. Die Prototypen sind:

```

template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);

```

```

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2,
           BinaryPredicate binary_pred);

```

24.9 Rechnen mit komplexen Zahlen: Der C++-Standardtyp `complex`

Komplexe Zahlen werden in den Natur- und Ingenieur-Wissenschaften viel verwendet. Sie bestehen aus dem Real- und dem Imaginärteil, die beide von einem der möglichen Typen für reelle Zahlen sein können (`float`, `double`, `long double`). Der Standarddatentyp für komplexe Zahlen ist deshalb kein Grunddatentyp, sondern zusammengesetzt. Weil die Bestandteile eines `complex`-Objekts, also Real- und Imaginärteil, von einem der Typen `float`, `double` oder `long double` sein können, sind komplexe Zahlen als Klassen-Template realisiert. Die für den Compiler nötigen Informationen enthält der Header `<complex>`. Die Tabelle 24.1 enthält die Funktionen, die mit komplexen Zahlen verwendet werden können. Einige Beispiele zeigen, wie mit komplexen Zahlen in C++ gerechnet wird. Eine andere Möglichkeit zur Ermittlung von π ist die Benutzung der Konstanten `M_PI` bzw. `M_PI_4` aus `<cmath>` (wird nicht immer zur Verfügung gestellt, weil nicht Teil des C-Standards[ISO]).

Listing 24.59: Anwendung komplexer Zahlen

```
// cppbuch/k24/complex/complex.cpp
#include<iostream>
#include<complex>           // Header für komplexe Zahlen
#include<cmath>             // atan()
using namespace std;
// Anstatt double sind auch float und long double möglich.
int main() {
    complex<double> c1;      // komplexe Zahl 0.0 + 0.0i erzeugen
    complex<double> c2(1.2, 3.4); // (1.2 + 3.4i) erzeugen
    cout << c2 << endl;      // Standard-Ausgabeformat: (1.2,3.4)
    c1 += c2 + c1;           // beispielhafte Rechenoperationen
    c1 = c2 * 5.0;
    double re = c1.real();   // Realteil ermitteln
    cout << re << endl;      // und ausgeben
    cout << c1.imag() << endl; // Imaginärteil direkt ausgeben
    // Beispiele mit Hilfsfunktionen
    complex<double> c3 = {1.0, 2.0}; // (1.0 + 2.0i) erzeugen, alternative Schreibweise
    c1 = conj(c3);             // konjugiert komplex: (1.0 - 2.0i)
    // Umrechnung aus Polarkoordinaten
    const double PI = 4.0 * atan(1.0); //  $\pi$  berechnen
    double betrag = 100.0;
    double phase = PI/4.0;      //  $\pi/4 = 45^\circ$ 
    c1 = polar(betrag, phase);

    // Umrechnung in Polarkoordinaten
    double rho = abs(c1);      // Betrag  $\rho$ 
    double theta = arg(c1);    // Winkel  $\theta$ 
    double nrm = norm(c1);     // Betragsquadrat
    cout << "Betrag = " << betrag << endl;
    cout << "rho = " << rho << endl;
```

Tabelle 24.1: Mathematische Funktionen für complex-Zahlen

Schnittstelle	mathematische Entsprechung
T real(const C& x)	Realteil von x ($= x.\text{real}()$)
T imag(const C& x)	Imaginärteil von x ($= x.\text{imag}()$)
T abs(const C& x)	Betrag von x
C fabs(const C& x)	Betrag von x als komplexe Zahl
T arg(const C& x)	Phasenwinkel von x in rad
T norm(const C& x)	Betragsquadrat von x
C conj(const C& x)	zu x konjugiert-komplexe Zahl
C polar(const T& rho, const T& theta)	Zahl entsprechend Betrag rho und Phase theta
C acos(const C& x)	$\arccos x$
C acosh(const C& x)	$\operatorname{acosh} x$
C asin(const C& x)	$\arcsin x$
C asinh(const C& x)	$\operatorname{asinh} x$
C atan(const C& x)	$\arctan x$
C atanh(const C& x)	$\operatorname{atanh} x$
C cos(const C& x)	$\cos x$
C cosh(const C& x)	$\cosh x$
C exp(const C& x)	e^x
C log(const C& x)	$\ln x$
C log10(const C& x)	$\log_{10} x$
C pow(const C& x, int y)	x^y
C pow(const C& x, const T& y)	x^y
C pow(const T& x, const C& y)	x^y
C pow(const C& x, const C& y)	x^y
C sin(const C& x)	$\sin x$
C sinh(const C& x)	$\sinh x$
C sqrt(const C& x)	$+\sqrt{x}$
C tan(const C& x)	$\tan x$
C tanh(const C& x)	$\tanh x$

Abkürzungen:
T = einer der Typen float, double oder long double
C = complex<T>

```
// Fortsetzung
cout << "Norm = " << nrm << endl;
cout << "Phase = " << phase << endl;
cout << "theta = " << theta
    << " = " << theta/PI*180. << " Grad" << endl;
cout << "Komplexe Zahl eingeben. Erlaubte Formate z.B.: "
    << "\n (1.78, -98.2)\n (1.78)\n 1.78\n:";
cin >> c1;
cout << "komplexe Zahl = " << c1 << endl;
}
```

Mit komplexen Zahlen kann wie mit reellen Zahlen gerechnet werden. Ausgenommen sind nur die Operatoren ++, - und %. Auch die Prüfung auf Gleichheit (==) oder Ungleich-

heit (!=) ist möglich. Die anderen Vergleichsoperatoren ergeben bei komplexen Zahlen keinen Sinn.



Übung

24.3 Schreiben Sie ein Programm zur Lösung der quadratischen Gleichung $x^2 + px + q = 0$, wobei Sie komplexe Zahlen für das Ergebnis verwenden. Die Lösungsformeln: $D = p^2/4 - q$, $x_1 = -p/2 + \sqrt{D}$, $x_2 = -p/2 - \sqrt{D}$. Eine komplexe Lösung ergibt sich, wenn $D < 0$ ist.

24.10 Schnelle zweidimensionale Matrix

In Abschnitt 5.7.3 wird eine Klasse für ein dynamisches 2-dimensionales Array beschrieben. Die folgende Matrix-Klasse ist eine Erweiterung mit überladenen Operatoren. Dabei ist wie bei der Standard-Vector-Klasse der Zugriff auf ein Matrix-Element über den Indexoperator `[]` ungeprüft, der Zugriff über `at(i, j)` geprüft. Bei der Zuweisung müssen die Dimensionen links und rechts übereinstimmen.

Der Index-Operator `[z]` gibt einen Zeiger auf die Zeile `z` zurück, sodass `[z][s]` das Element `s` dieser Zeile zurückgibt. Das folgende Listing zeigt die Klasse. Im danach anschließenden Abschnitt geht es um die Optimierung mathematischer Operationen.

Listing 24.60: Dynamisches 2-dimensionales Array

```
// cppbuch/k24/array2d/array2d.h
// 2-dim. Array-Klasse mit zusammenhängendem (contiguous) Memory-Bereich
#ifndef ARRAY2D_H
#define ARRAY2D_H
#include<stdexcept>
#include<utility>
#include<algorithm>
#include<iostream>

template<typename T>
class Array2d {
public:
    // Typedefs wegen STL-Konformität
    typedef T value_type;
    typedef T& reference;
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef int difference_type;
    typedef unsigned int size_type;
```

```

// Konstruktoren, Destruktor, Zuweisung
Array2d(int zeilen, int spalten);
Array2d(int zeilen, int spalten, const T& init);
Array2d(const Array2d& a);
Array2d(Array2d&& a);
~Array2d();
Array2d& operator=(const Array2d& a);
Array2d& operator=(Array2d&& a);

// Elementfunktionen
int getZeilen() const;
int getSpalten() const;
void init(const T& wert); // Alle Elemente mit wert initialisieren
const T& at(int z, int s) const;
T& at(int z, int s);
const T* operator[](int z) const;
T* operator[](int z);
const T* data() const;

// STL-entsprechende Funktionen
size_t size() const { return zeilen*spalten;}
iterator begin() { return arr;}
iterator end() { return arr + zeilen*spalten;}
const_iterator begin() const { return arr;}
const_iterator end() const { return arr + size();}
const_iterator cbegin() const { return arr;}
const_iterator cend() const { return arr + size();}
void swap(Array2d& a);

private:
void checkIndizes(int z, int s) const;
void checkDimension(const Array2d<T>& a) const;
int zeilen;
int spalten;
T* arr;
};

// relationale Operatoren
template<typename T>
bool operator==(const Array2d<T>& a, const Array2d<T>& b) {
    if(a.getZeilen() != b.getZeilen()) {
        return false;
    }
    if(a.getSpalten() != b.getSpalten()) {
        return false;
    }
    return std::equal(a.begin(), a.end(), b.begin());
}

template<typename T>
bool operator<(const Array2d<T>& a, const Array2d<T>& b) {
    return std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end());
}

```

```

// um aus == und < alle anderen relationalen Operatoren bei Bedarf automatisch zu bilden:
using namespace std::rel_ops;
// Implementierung, soweit nicht oben schon vorhanden:

template<typename T>
Array2d<T>::Array2d(int z, int s)
    : zeilen(z), spalten(s), arr(new T[z*s]) {
}

template<typename T>
Array2d<T>::Array2d(int z, int s, const T& wert)
    : zeilen(z), spalten(s), arr(new T[z*s]) {
    init(wert);
}

template<typename T>
Array2d<T>::Array2d(const Array2d<T>& a)
    : zeilen(a.getZeilen()), spalten(a.getSpalten()),
      arr(new T[zeilen*spalten]) {
    for(size_t i = 0; i < size(); ++i) {
        arr[i] = a.arr[i];
    }
}

template<typename T>
Array2d<T>::Array2d(Array2d<T>&& a) // bewegender (moving) Kopierkonstruktor
    : zeilen(0), spalten(0), arr(0) {
    swap(a);
}

template<typename T>
void Array2d<T>::swap(Array2d<T>& a) {
    std::swap(a.zeilen, zeilen);
    std::swap(a.spalten, spalten);
    std::swap(a.arr, arr);
}

template<typename T>
Array2d<T>::~~Array2d() {
    delete [] arr;
}

template<typename T>
Array2d<T>& Array2d<T>::operator=(const Array2d<T>& a) {
    checkDimension(a);
    for(size_t i = 0; i < size(); ++i) {
        arr[i] = a.arr[i];
    }
    return *this;
}

template<typename T>
Array2d<T>& Array2d<T>::operator=(Array2d<T>&& a) {

```



```

        swap(a);
        return *this;
    }

    template<typename T>
    inline int Array2d<T>::getZeilen() const {
        return zeilen;
    }

    template<typename T>
    inline int Array2d<T>::getSpalten() const {
        return spalten;
    }

    template<typename T>
    void Array2d<T>::init(const T& wert) {
        for(size_t i = 0; i < size(); ++i) {
            arr[i] = wert;
        }
    }

    template<typename T>
    inline const T* Array2d<T>::operator[](int z) const {
        return &arr[z*spalten];
    }

    template<typename T>
    inline T* Array2d<T>::operator[](int z) {
        return &arr[z*spalten];
    }

    template<typename T>
    inline void Array2d<T>::checkIndizes(int z, int s) const {
        if(z < 0 || z >= zeilen)
            throw std::range_error(
                "Array2d: Zeile ausserhalb des erlaubten Bereichs");
        if(s < 0 || s >= spalten)
            throw std::range_error(
                "Array2d: Spalte ausserhalb des erlaubten Bereichs");
    }

    template<typename T>
    inline void Array2d<T>::checkDimension(const Array2d<T>& a) const {
        if(zeilen != a.getZeilen() || spalten != a.getSpalten())
            throw std::range_error("Array2d: ungleiche Zeilen-/Spaltenanzahl");
    }

    template<typename T>
    inline const T& Array2d<T>::at(int z, int s) const {
        checkIndizes(z, s);
        return arr[z*spalten + s];
    }

```

```
template<typename T>
inline T& Array2d<T>::at(int z, int s) {
    checkIndizes(z, s);
    return arr[z*spalten + s];
}

template<typename T>
inline const T* Array2d<T>::data() const {
    return arr;
}
#endif
```

Durch die Verwendung von inline-Methoden ist die Klasse genauso schnell wie ein dynamisches C-Array. Sie ist aber leichter zu benutzen; insbesondere muss man sich keine Gedanken über das Zusammenspiel von `new` und `delete` machen und hat auf Wunsch einen indexgeprüften Zugriff. Durch den bewegenden Kopierkonstruktor und den bewegenden Zuweisungsoperator werden temporäre Objekte vermieden.



Mehr zu bewegenden Kopierkonstruktoren usw. lesen Sie auf den Seiten 591 ff.

Durch die Einbindung des Namensraums `std::rel_ops` wird erreicht, dass die relationalen Operatoren `!=`, `>`, `>=`, `<=` bei Bedarf aus den Operatoren `==` und `<` erzeugt werden.



Mehr zu `rel_ops` lesen Sie auf Seite 747.



24.10.1 Optimierung mathematischer Array-Operationen



Hinweis

Das Verständnis dieses Abschnitts setzt die Kenntnis der folgenden Kapitel voraus:
 Abschnitt 6.5: Templates mit variabler Parameterzahl
 Kapitel 22: Performance, Wert- und Referenzsemantik
 Abschnitt 27.4: Tupel

Die Klasse `Array2d` kann mit mathematischen Operationen versehen werden. Wenn dies mit überladenen Operatoren geschieht, sind Anweisungen wie die folgenden möglich:

```
// Addition
Array2d<int> a1(2, 3);
Array2d<int> a2(2, 3);
Array2d<int> a3(2, 3);
Array2d<int> ergebnis(2, 3);
// Berechnung der Array-Inhalte weggelassen
ergebnis = a1 + a2 + a3;           // operator+()
```

Das Problem aus Performance-Sicht ist die Erzeugung temporärer Objekte, wie schon in Kapitel 22 erläutert. Eine naiver `operator+()` würde auf der rechten Seite der Zuweisung ein temporäres Objekt erzeugen. Die Summation geschieht mit einer Schleife über alle Matrix-Elemente. Dasselbe geschieht im zweiten Schritt. Anschließend wird der Inhalt des Ergebnisses der Variablen `ergebnis` wieder mit einer Schleife über alle Elemente zuge-

wiesen. Zuletzt wird das temporäre Objekt vernichtet. Abschnitt 22.2 (Optimierung durch Referenzsemantik für R-Werte) zeigt, wie einige der temporären Objekte vermieden werden können. Noch besser wäre es jedoch, wenn das Entstehen aller temporären Objekte verhindert werden könnte und nur eine einzige Schleife notwendig wäre, etwa

```
for(size_t i = 0; i < ergebnis.getZeilen(); ++i) {
    for(size_t j = 0; j < ergebnis.getSpalten(); ++j) {
        ergebnis[i][j] = a1[i][j] + a2[i][j] + a3[i][j];
    }
}
```

Sinn des Überladens ist es, solche Schleifen zugunsten einfacher Anweisungen wie `ergebnis = a1 + a2 + a3;` zu vermeiden, wobei die Anzahl der Operanden auf der rechten Seite beliebig sein kann. Am Ende des Kapitels 22 wird auf die Möglichkeit hingewiesen, das Problem mit Expression Templates zu lösen. Eine andere Möglichkeit bieten Tupel in der Kombination mit Templates variabler Parameterzahl (*variadic templates*). Dieser Ansatz wird im Folgenden erläutert. Es zeigt sich, dass er genauso gute Ergebnisse wie Expression Templates liefert. Die Grundidee ist:

- Es gibt eine Template-Klasse `Summand`. In einem Objekt dieser Klasse ist nur ein Verweis auf ein Array und ein möglicherweise vorhandener multiplikativer Faktor gespeichert.

Listing 24.61: Klasse `Summand`

```
// Auszug aus cppbuch/k24/array2dmath/array2d.h
template<typename T>
struct Summand {
    Summand(const T& faktor, const Array2d<T>& arr)
        : f(faktor), a(arr) {}

    Summand(const Array2d<T>& arr)
        : f(T(1)), a(arr) {}

    T getWert(size_t index) const {
        return *(a.data()+index) * f;
    }

    size_t getZeilen() const {
        return a.getZeilen();
    }

    size_t getSpalten() const {
        return a.getSpalten();
    }

    Summand<T>& mult(const T& faktor) {
        f *= faktor;
        return *this;
    }
}
```

```
T f;
    const Array2d<T>& a;
};
```

- `operator+(const Array2d&, const Array2d&)` berechnet nichts, sondern erzeugt zwei Verweisobjekte des Typs `Summand` und gibt ein Tupel, das diese zwei Objekte enthält, zurück:

```
// Auszug aus cppbuch/k24/array2dmath/array2d.h
template<typename T>
tuple<Summand<T>, Summand<T>>
inline operator+(const Array2d<T>& x, const Array2d<T>& y) {
    return tuple<Summand<T>, Summand<T>>(Summand<T>(x), Summand<T>(y));
}
```

- Bei mehr als zwei Summanden wird dem zurückgegebenen Tupel ein `Summand`-Objekt mit einem Verweis auf das nächste zu addierende `Array2d` hinzugefügt.

```
ergebnis = Tupel(a1, a2) + Summand(a3); // Pseudocode!
```

Der zugehörige Plus-Operator erweitert einfach das Tupel um `a3`. Da dieses Schema für beliebig viele Summanden gilt, kommen hier die *variadic templates* ins Spiel:

```
template<typename T, typename... Args>
tuple<Summand<T>, Args...>
inline operator+(const tuple<Args...>& t, const Array2d<T>& y) {
    return t + Summand<T>(y);
}
```

Die letzte Zeile ruft den folgenden Operator auf, der mit der Funktion `tuple_cat()` das Tupel verlängert:

```
template<typename T, typename... Args>
tuple<Summand<T>, Args...>
inline operator+(const tuple<Args...>& t, const Summand<T>& y) {
    return tuple_cat(tuple<Summand<T>>(y), t);
}
```

- Bei den vorangehenden Schritten ist entscheidend, dass keinerlei `new`-Operation und keinerlei Schleife vorkommen. Der letzte Schritt ist die Zuweisung des letzten Tupels, das Verweise auf alle beteiligten Arrays enthält:

```
ergebnis = Tupel(a1, a2, a3, ...); // Pseudocode!
```

Dieser Zuweisungsoperator erledigt die eigentlichen Additionen in einer einzigen Schleife, indem er auf ein Array-Element die Summe der entsprechenden Array-Elemente des Tupels addiert:

```
Array2d& operator=(const tuple<Args...>& t) {
    for(size_t i = 0; i < size(); ++i) {
        arr[i] = Summe<sizeof...(Args), T, Args...>::ergebnis(i, t);
    }
    return *this;
}
```

Die Klasse `Summe` unten wertet die entsprechenden Array-Elemente *zur Compilerzeit* aus. Das bedeutet, dass es keine `for`-Schleife dafür geben kann, die zur Laufzeit ausgeführt wird, sondern dass der Compiler die Addition rekursiv bewerkstelligt, ähnlich wie in Abschnitt 6.5 beschrieben.

```
// Berechnung über alle Summanden
template<size_t N, typename T, typename... Args>
struct Summe {
    static T ergebnis(size_t index, const tuple<Args...>& t) {
        return get<N-1>(t).getWert(index)
            + Summe<N-1, T, Args...>::ergebnis(index, t);
    }
};

// partielle Spezialisierung zum Rekursionsabbruch
template<typename T, typename... Args>
struct Summe<1, T, Args...> {
    static T ergebnis(size_t index, const tuple<Args...>& t) {
        return get<0>(t).getWert(index);
    }
};
```

`get<M>(t)` gibt das Tupel-Element Nr. `M` zurück, also ein `Summand`-Objekt. `getWert(size_t index)` ist eine Methode der Klasse `Summand`, die das Array-Element an der Stelle `index` zurückgibt. Die Funktion `ergebnis<N, T, Args>()` gibt also das entsprechende Array-Element plus die Summe der restlichen Elemente zurück. Falls `N = 0` ist, gibt es keine restlichen Elemente mehr. Der Compiler instanziiert die Funktion `ergebnis()` für alle rekursiv ermittelten Typen.

Der Vorteil dieses Ansatzes ist derselbe wie der Vorteil von Expression Templates, dass nämlich die eigentliche Berechnung möglichst spät geschieht, und dass der Compiler bei der Auswertung eines Ausdrucks nur Stack-Objekte erzeugt. Durch Inlining und Optimierung wird der Code dann auf das Wesentliche reduziert.

Die Datei `cppbuch/k24/array2dmath/array2d.h` wurde noch um die Multiplikation mit Faktoren und die Matrixmultiplikation erweitert, sodass auch Ausdrücke wie `ergebnis = -a1 + 2.0 * a2 + 3.0 * a3*a4`; möglich sind. Alle Operanden sind 2-dimensionale Matrizen. Die Datei wird aus Platzgründen hier nicht abgedruckt. Abschließend folgt ein Vergleich mit anderen Matrix-Klassen. Es wurden 500x500-Matrizen addiert. Jede Matrix hatte somit 250.000 Elemente. Die Compilation wurde mit dem Schalter `-O3` für maximale Optimierung durchgeführt.

Tabelle 24.2: Testergebnisse für die Addition von 500x500-Matrizen

Bibliothek	Klasse	Anweisung	Dauer [ms]	Hinweis
Standard-C++	<code>Array2d<double></code>	<code>aerg = -a1 + 2.0 * a2 + a3;</code>	1,5	1.
	<code>valarray<double></code>	<code>verg = -v1 + 2.0 * v2 + v3;</code>	1,5	2.
Boost	<code>matrix<double></code>	<code>aerg = -a1 + 2.0 * a2 + a3;</code>	2,5	3.
Blitz	<code>Array<double, 2></code>	<code>aerg = -a1 + 2.0 * a2 + a3;</code>	1,5	4.

Die Testergebnisse beziehen sich auf meinen PC; auf Ihrem Rechner können die Zahlen andere sein. Es ist zu sehen, dass die Boost-Matrix nicht ganz so gut optimiert ist. Den

gesamten Test finden Sie im Verzeichnis `cppbuch/k24/array2dmath/performancectest`. Bitte lesen Sie zuerst die dortige Datei `README.txt`, wenn Sie die Tests durchführen wollen. Die Hinweise zu der Tabelle 24.2:

1. Keine Bibliothek, weil es sich um die oben vorgestellte Klasse handelt.
2. Ein `valarray` ist keine Matrix. Um dennoch einen Vergleich durchzuführen, wurden `valarray`-Objekte mit je 250.000 Elementen verwendet.
3. Die Matrix ist in der Datei `boost/numeric/ublas/matrix.hpp` definiert. Die zugrundeliegende Boost-Version ist 1.45.
4. Mit der Blitz-Bibliothek (Quelle: <http://oonumerics.org/blitz/>) sind multi-dimensionale Arrays möglich. Die Zahl 2 im Array-Typ gibt die Dimension an. Die zugrundeliegende Blitz-Version ist 0.9 mit einem Patch, der die Compilation mit G++ ab Version 4.3 ermöglicht (siehe `cppbuch/k24/array2dmath/performancectest/blitz/readme.txt`).

Tabelle 24.3: Testergebnisse für die Multiplikation von 500x500-Matrizen

Bibliothek	Klasse	Anweisung	Dauer [s]	Hinweis
Boost	<code>Array2d<double></code>	<code>aerg = a1 * a2;</code>	0,65	1.
	<code>matrix<double></code>	<code>aerg = prod(a2, a1);</code>	0,65	2.
Blitz	<code>Array<double, 2></code>	<code>aerg = sum(a1(i, k) * a2(k, j), k);</code>	0,65	3.

Die Tabelle 24.3 zeigt die Ergebnisse für die Matrix-Multiplikation. Alle Verfahren sind offensichtlich gleich gut optimiert. Zu den Hinweisnummern der letzten Spalte:

1. Nur `Array2d` hat überladene Operatoren, und dies ohne Geschwindigkeitsverlust.
2. Das Überladen von Operatoren ist nicht realisiert, sodass ein Funktionsaufruf verwendet wird.
3. Eine Matrix-Multiplikation wird mit Blitz *nicht* mit `a1 * a2`, sondern durch die o.a. Anweisung erreicht. Dabei sind `i`, `j` und `k` mithilfe vorgegebener Typen (`firstIndex` usw.) definiert, wie in `cppbuch/k24/array2dmath/performancectest/blitz/multblitz.cpp` und unten zu sehen. Näheres bitte ich der Blitz-Dokumentation zu entnehmen.

```
firstIndex i; // Platzhalter für den ersten Index
secondIndex j; // Platzhalter für den zweiten Index
thirdIndex k; // Platzhalter für den dritten Index
```

Die Klasse `Array2d` demonstriert gut die Wirksamkeit des Konzepts »Variadic Templates«. Die Klasse implementiert noch jedoch nicht alle typischen mathematischen Operationen wie etwa Klammerrechnung (`2.0*(a1+a2)`) oder die Division von Matrizen.

24.11 Singleton

Gelegentlich kommt es vor, dass man nur genau *eine* Instanz einer Klasse braucht, zum Beispiel zum Verwalten von Ressourcen, die definitiv nur einmal vorkommen. Das kann ein Objekt zur Ansprache mehrerer Drucker sein oder eine Log-Datei, die an vielen Stellen eines Programms benutzt wird. Das Singleton-Entwurfsmuster [Gamma] ist dafür

geeignet. Es stellt sicher, dass es exakt eine Instanz gibt, auf die über eine eindeutige Schnittstelle zugegriffen wird. Wegen der einfachen Anforderungen, die aber nicht ganz einfach zu implementieren sind, zählt dieses Entwurfsmuster zu den am häufigsten diskutierten.

24.11.1 Implementierung mit einem Zeiger

Eine einfache Implementierung mit einem Zeiger könnte so aussehen:

```
class Singleton {
public:
    static Singleton* getInstance() {
        if(!pInstance) { // statisches Attribut
            pInstance = new Singleton;
        }
        return pInstance;
    }
    // Restliche Methoden weggelassen
private:
    static Singleton* pInstance;
    Singleton(); // direkte Konstruktion verbieten
    Singleton(const Singleton&); // Kopie verbieten
    Singleton& operator=(const Singleton&); // sinnlose Zuweisung verbieten
};
```

In einer zugehörigen Implementationsdatei *Singleton.cpp* wird der Zeiger mit 0 initialisiert: `Singleton* Singleton::pInstance = 0;` Es ist sichergestellt, dass die Erzeugung mit `new` exakt einmal geschieht. Konstruktor und Kopierkonstruktor sind `private`, damit eine Objekterzeugung unter Umgehung von `getInstance()` nicht möglich ist. Weil es nur genau eine einzige Instanz gibt, ist der Gebrauch des Zuweisungsoperators sinnlos. Er ist deshalb ebenfalls als `private` deklariert.

Nachteile

Ein Problem ist die Entsorgung so eines Singleton-Objekts. Natürlich wird der Speicher ganz am Ende des Programms vom Betriebssystem wieder freigegeben. Der Destruktor wird aber nicht automatisch aufgerufen, sodass zwar kein Speicherleck entsteht, aber es kann sein, dass Ressourcen wie Dateien oder Netzwerkverbindungen, die vom Singleton vielleicht genutzt wurden, nicht freigegeben werden.

Abhilfe könnte man schaffen, indem explizit vor Ende des `main`-Programms `delete` auf den Zeiger angewendet wird – ein Weg, der dem C++-Prinzip RAII (siehe Glossar) widerspricht.

Ein weiterer Nachteil besteht darin, dass ein `delete` an einer beliebigen anderen Stelle nicht verhindert werden kann. Oder man macht den Destruktor `privat` – dann kann das Singleton-Objekt gar nicht gelöscht werden.

24.11.2 Implementierung mit einer Referenz

Das Problem der Ressourcenfreigabe ist besser lösbar, wenn ein statisches Attribut angelegt wird. Der Destruktor würde automatisch nach dem Ende von `main()` ausgeführt

und die Ressourcen freigeben. Von der Funktion `getInstance()` würde einfach eine Referenz auf das Attribut zurückgegeben:

```
class Singleton {
public:
    static Singleton& getInstance() {
        return instance;
    }
    // Rest weggelassen
private:
    static Singleton instance;
    // Rest weggelassen
};
```

Nachteil

Ein Problem könnte entstehen, wenn in einer beliebigen anderen Übersetzungseinheit auf das Singleton zugegriffen würde, etwa

```
// irgendeineDatei.cpp
int ergebnis = Singleton::getInstance().eineFunktion();
```

Zwar werden globale und statische Objekte initialisiert, ehe die erste Zeile von `main()` ausgeführt wird. Weil C++ aber keine Reihenfolge bei der Initialisierung dieser Objekte festlegt, wenn diese in verschiedenen Übersetzungseinheiten vorliegen, könnte es sein, dass in der obigen Anweisung das Singleton-Objekt noch nicht existiert.

24.11.3 Meyers' Singleton

Meyers [ScM] hat die bisher beschriebenen Probleme auf elegante Art gelöst, indem er eine *funktionslokale* statische Variable nutzt. Ein funktionslokales statisches Objekt, das keine zur Compilationszeit bekannte Konstante ist, wird erst dann initialisiert, wenn die Funktion zum ersten Mal aufgerufen wird. Lokale statische Variable haben Sie schon in Abschnitt 3.1.3 (Seite 105) bei der »Funktion mit Gedächtnis« kennengelernt. Das Prinzip:

```
static Singleton& getInstance() {
    static Singleton instance;
    return instance;
}
```

Natürlich müssen auch hier Konstruktor, Kopierkonstruktor, Zuweisungsoperator und Destruktor `private` sein. Nun kann es sein, dass man dem Singleton bei der Initialisierung Parameter übergeben möchte, zum Beispiel den Namen einer Log-Datei – als Alternative zur Abfrage einer Umgebungsvariablen oder Konfigurationsdatei. Der Funktion `getInstance()` jedesmal den Parameter mitzugeben, ist unsicher. Eine `setParameter()`-Funktion würde nicht garantieren, dass die Initialisierung nur einmal erfolgt.

Erweiterung um Initialisierung mit Parametern

Aus diesem Grund wird im Folgenden ein Beispiel auf der Basis von [ScM] gezeigt, das die Initialisierung des Singletons vor der Verwendung garantiert, und auch, dass eine mehrfach versuchte Initialisierung nicht möglich ist. Die Initialisierungsfunktion muss

nur einmalig vor der sonstigen Verwendung aufgerufen werden. Das Singleton ist ein Objekt zum Schreiben in eine Log-Datei, deren Name `main()` übergeben wird. Zuerst sei eine mögliche Anwendung gezeigt:

Listing 24.62: Singleton-Anwendung

```
// cppbuch/k24/singleton/main.cpp
#include<iostream>
#include "LogSingleton.h" // siehe unten
using namespace std;

int main(int argc, char* argv[]) {
    if(argc == 2) {
        try {
            // 1. initialisieren
            LogSingleton::initialize(argv[1]); // nicht Thread-sicher, siehe unten
            // 2. Instanz holen und verwenden:
            LogSingleton& s1 = LogSingleton::getInstance();
            cout << "s1.getLogfilename() = " << s1.getLogfilename() << endl;
            // Dieselbe Instanz unter anderem Namen:
            LogSingleton& s2 = LogSingleton::getInstance();
            cout << "Identische Objekte. Die Adressen sind dieselben:" << endl
                 << &s1 << endl
                 << &s2 << endl;
            // Verwendung des Singletons
            s1.log("Die erste Nachricht!");
            s2.log("Die zweite Nachricht!");
        }
        catch (const logic_error& e) { // Erklärung siehe LogSingleton.h unten
            cout << "Programmierfehler: " << e.what() << endl;
        }
        catch (const runtime_error& e) { // Erklärung siehe LogSingleton.h unten
            cout << "Runtime-Fehler: " << e.what() << endl;
        }
    }
    else {
        cout << "Gebrauch: " << argv[0] << " Log-Dateiname" << endl;
    }
} // Hier wird ggf. der LogSingleton-Destruktor aufgerufen.
```

Es folgt die `LogSingleton`-Klasse. In der Funktion `getInstance()` wird geprüft, ob es sich um den ersten Aufruf handelt. Es kein Problem, wenn das Singleton dann noch nicht initialisiert ist – die Funktion `initialize()` benötigt einmalig die Instanz, um die Attribute wie gewünscht zu belegen. Anschließend wird das Flag initialisiert gesetzt, sodass weitere Aufrufe von `getInstance()` möglich sind.

Listing 24.63: Logging-Klasse als Singleton

```
// cppbuch/k24/singleton/LogSingleton.h
#ifndef LOGSINGLETON_H
#define LOGSINGLETON_H
#include<iostream>
#include<fstream>
#include<stdexcept>
```

```

class LogSingleton {
public:
    static LogSingleton& getInstance() { // liefert stets Referenz auf dasselbe Objekt
        static LogSingleton single;
        static bool ersterAufruf = true;
        if (!ersterAufruf && !single.initialisiert) {
            throw std::logic_error("LogSingleton ist nicht initialisiert!");
        }
        ersterAufruf = false;
        return single;
    }

    // initialize() muss exakt einmal vor getInstance() aufgerufen werden.
    static void initialize(const std::string& lfn) {
        LogSingleton& s = getInstance();
        if (s.initialisiert) {
            throw std::logic_error("mehrfache Initialisierung ist "
                                   "nicht möglich!");
        }
        s.logfilename = lfn; // Initialisierung
        s.datei.open(lfn);
        if (!s.datei.good()) {
            throw std::runtime_error(lfn + " kann nicht geöffnet werden.");
        }
        s.initialisiert = true;
    }

    void log(const std::string& message) { // normale Benutzung
        datei << message << std::endl;
    }

    const std::string& getLogfilename() const {
        return logfilename;
    }
private:
    bool initialisiert;
    std::string logfilename;
    std::ofstream datei;
    LogSingleton() : initialisiert(false) { // Konstruktor. Externen Aufruf verbieten
    }
    LogSingleton(const LogSingleton&); // Kopierkonstruktor verbieten
    void operator=(const LogSingleton&); // sinnlose Zuweisung verbieten
    virtual ~LogSingleton() { // externen Aufruf verbieten
        // Ausgabe nur zur Demonstration
        std::cout << "LogSingleton-Destruktor aufgerufen" << std::endl;
        // Ressourcen freigeben. Dies ist hier nicht unbedingt notwendig, weil
        datei.close(); // der ofstream-Destruktor dies automatisch erledigt hätte.
    }
};
#endif

```

**Hinweis**

Das vorgestellte Singleton ist nicht Thread-safe. Das heißt, ein gleichzeitiger Zugriff von verschiedenen Threads aus kann zu Inkonsistenzen führen. Auch gibt es je nach Anwendungsfall verschiedene Singleton-Konzepte. Wenn Sie über darüber mehr wissen möchten, empfehle ich Ihnen [\[Alex\]](#).

24.12 Vermischtes

24.12.1 Erkennung eines Datums

Mit Hilfe eines regulären Ausdrucks soll überprüft werden, ob ein Datum der Form `tt.mm.jjjj` gültig ist. Dabei soll es auch möglich sein, nur eine Ziffer für den Tag bzw. den Monat einzugeben. Da hier nur die Gültigkeit interessiert und nicht, wo sich ein gültiges Datum innerhalb einer Zeichenkette befindet, genügt die Abfrage mit `regex_match()`. Ein passender regulärer Ausdruck (von vielen möglichen) ist `\d\d?\.\d\d?\.\d\d\d\d`.

Listing 24.64: Syntaktische Gültigkeit eines Datums

```
// cppbuch/k24/vermishtes/regex/datumsyntaxpruefen.cpp
#include <iostream>
#include<boost/regex.hpp>
#include<string>

bool datumok(const std::string& eingabe) {
    boost::regex datumregex("\\d\\d?\\.\\d\\d?\\.\\d\\d\\d\\d");
    return boost::regex_match(eingabe, datumregex);
}

using namespace std;

int main(int argc, char* argv[]) {
    if(2 != argc) {
        cout << "Gebrauch: datumpruefen.exe tt.mm.jjjj" << endl;
    }
    else {
        try {
            if(datumok(argv[1])) {
                cout << argv[1] << " ist ein gültiges Datum." << endl;
            }
            else {
                cout << argv[1] << " ist kein gültiges Datum." << endl;
            }
        } catch(boost::regex_error& re) {
            cerr << "Fehler: " << re.what() << endl;
        }
    }
}
```

Syntax oder Semantik?

Am obigen Beispiel wird sofort ein Problem klar: 00.0.0000 ist ein ungültiges Datum, wird aber vom Programm als gültiges Datum interpretiert. Der Grund liegt darin, dass das Programm nur die vorgegebene Syntax prüft, aber die Semantik (Bedeutung) des Inhalts ignoriert. Was tun? Es gibt verschiedene Abstufungen der Prüfung:

1. Die einfachste Syntaxprüfung ist die oben dargestellte. Um die Gültigkeit zu prüfen, müssen Tag, Monat und Jahr aus dem String extrahiert werden, um dann die Gültigkeit zum Beispiel mit der Funktion `bool istGueltigesDatum(int t, int m, int j)` von Seite 336 zu prüfen.

2. Man kann aber auch in Erwägung ziehen, bereits bei der syntaktischen Analyse die Gültigkeit der Daten zu überprüfen. Für den Tag gibt es folgende Möglichkeiten:

- Wenn der Tag einstellig ist, liegt die Ziffer im Bereich [1-9].
- Ist die erste Ziffer eine 0, liegt die zweite im Bereich [1-9].
- Ist die erste Ziffer eine 1 oder eine 2, liegt die zweite im Bereich [0-9].
- Ist die erste Ziffer eine 3, liegt die zweite im Bereich [0-1].

Der erste und der zweite Punkt werden mit dem regulären Ausdruck `0?[1-9]` realisiert, der dritte mit `(1|2)\d`, der vierte mit `3[01]`. Der reguläre Ausdruck für den Tag einschließlich des Punktes kann damit als `(0?[1-9])|((1|2)\d)|(3[01])\.` formuliert werden. Für den Monat gilt:

- Wenn der Monat einstellig ist oder mit einer 0 beginnt, liegt die Ziffer vor dem Punkt im Bereich [1-9].
- Wenn der Monat zweistellig ist, kann die erste Ziffer nur eine 1 sein und die zweite liegt im Bereich [0-2].

Der reguläre Ausdruck für den Monat einschließlich des Punktes kann damit als `(0?[1-9])|(1[0-2])\.` formuliert werden.

3. Mit der vorstehenden Lösung ist aber noch nicht alles erreicht, denn manche Monate haben nur 30 Tage, und der Februar sogar nur 28 bzw. 29 in einem Schaltjahr. 30 oder 28 Tage lassen sich mit einigem Aufwand in der Syntax berücksichtigen, aber das Schaltjahr nur noch mit sehr großem, unverhältnismäßigem Aufwand.

Man sieht, dass der Aufwand recht hoch getrieben werden kann. Bei der Überprüfung von Dialogeingaben oder von einer Datei eingelesenen Informationen ist daher abzuwägen, wie der Gesamtaufwand aus Syntaxprüfung und inhaltlicher Prüfung minimiert werden kann.



Empfehlung

Einfache reguläre Ausdrücke bevorzugen, ergänzt durch eine inhaltliche Prüfung.

Diese Empfehlung entspricht dem obigen Punkt 1. Sie erspart Planungs- und Rechenzeitaufwand. Die Funktion `datumok()` des obigen Listings wird dazu ersetzt durch

Listing 24.65: Gültigkeit eines Datums

```
// Auszug aus cppbuch/k24/vermischtes/regex/datumpruefen.cpp

bool datumok(const std::string& eingabe) {
    boost::regex datumregex("\\d\\d?\\.\\d\\d?\\.\\d\\d\\d\\d");
```

```

bool ergebnis = boost::regex_match(eingabe, datumregex);
if(ergebnis) { // Syntax ok, Inhalt prüfen
    std::vector<std::string> v;
    boost::split(v, eingabe, boost::is_any_of("."));
    ergebnis = istGultigesDatum(boost::lexical_cast<int>(v[0]),
                                boost::lexical_cast<int>(v[1]),
                                boost::lexical_cast<int>(v[2]));
}
return ergebnis;
}

```

Die Funktion `istGultigesDatum()` wird von Seite 336 übernommen.



Mehr über `split()` und `lexical_cast` lesen Sie in den Abschnitten 24.1.1 und 24.1.2.

24.12.2 Erkennung einer IP-Adresse

Die Lösung dieses Problem lässt sich auf das vorhergehende zurückführen. Eine IP-Adresse hat vier durch einen Punkt getrennte Felder, wobei jedes Feld aus einer vorzeichenlosen Zahl mit maximal drei Ziffern besteht. Ein möglicher regulärer Ausdruck ist `\d\d?\d?\.\d\d?\d?\.\d\d?\d?\.\d\d?\d?` oder kürzer `(?:\d\d?\d?\.){3}\d\d?\d?`. Es gilt die Bedingung, dass jede Zahl höchstens den Wert 255 annehmen kann. Im Folgenden ist nur die überprüfende Funktion gezeigt:

Listing 24.66: Gültigkeit einer IP-Adresse

```

// Auszug aus cppbuch/k24/vermischtes/regex/IPadressepruefen.cpp
bool ipok(const std::string& eingabe) {
    boost::regex datumregex("(?:\d\d?\d?\.){3}\d\d?\d?");
    bool ergebnis = boost::regex_match(eingabe, datumregex);
    if(ergebnis) { // Syntax ok, Inhalt prüfen
        std::vector<std::string> v;
        boost::split(v, eingabe, boost::is_any_of("."));
        for(size_t i = 0; i < v.size(); ++i) {
            ergebnis = ergebnis && boost::lexical_cast<int>(v[i]) < 256;
        }
    }
    return ergebnis;
}

```

24.12.3 Erzeugen von Zufallszahlen

[ISOC++] enthält einen großen Abschnitt über die Erzeugung von (Pseudo-)Zufallszahlen. Der Abschnitt ist für dieses Buch zu umfangreich und eher aus mathematischer Sicht als aus programmiersprachlicher Sicht interessant. Die Gleichverteilung und die Normalverteilung werden am häufigsten benutzt. Deshalb beschränke ich mich auf die folgenden drei Möglichkeiten:

1. Ein einfacher Zufallszahlengenerator `Random`, der auf den C-Funktionen `rand()` und `srand()` beruht.
2. Eine C++-Standardfunktion zur Erzeugung gleichverteilter Zufallszahlen.

3. Eine C++-Standardfunktion zur Erzeugung normalverteilter Zufallszahlen.

Die letzten beiden Möglichkeiten wie auch alle anderen neu in C++ aufgenommenen Zufallsverteilungen werden noch nicht von allen Compilern unterstützt. Der GNU C++ Compiler kennt sie erst ab Version 4.5. Allerdings gibt es eine funktionierende Boost-Library. Deswegen empfehle ich Ihnen bei Bedarf, die Library Boost.Random zu verwenden und die dazugehörige Dokumentation zu Rate zu ziehen.

Reproduzierbarkeit

Für alle gezeigten Verfahren gilt, dass die berechneten Zufallswerte reproduzierbar, also pseudo-zufällig sind. Erneutes Starten eines Programms liefert dieselbe Zahlenfolge. Entscheidend dabei ist die Initialisierung des Zufallszahlengenerators mit den Funktionen `srand(unsigned seed)` im ersten und `seed(unsigned seed)` in den letzten beiden Fällen. Die Reproduzierbarkeit erleichtert das Testen. Um bei jedem Programmstart eine *andere* Zahlenfolge zu erreichen, kann man der Seed-Funktion die aktuelle Systemzeit mit `time(NULL)` oder einen davon abgeleiteten Wert übergeben.

Zufallszahlen auf der Basis von `rand()`

`rand()` gibt eine etwa gleichverteilte Pseudozufallszahl zwischen 0 und `RAND_MAX` zurück, `srand(unsigned seed)` initialisiert den Zufallszahlengenerator. Im Beispiel wird die Einbindung als Funktionsobjekt gezeigt.

Listing 24.67: Zufallszahlengenerator

```
// cppbuch/include/Random.h
#ifndef RANDOM_H
#define RANDOM_H
#include<cstdlib> // rand(), srand() und RAND_MAX
class Random {
public:
    Random(size_t g = 100)
        : grenze(g) {
    }
    void setSeed(size_t seed) {
        srand(seed);
    }
    // gibt eine Pseudo-Zufallszahl zwischen 0 und range-1 zurück
    size_t operator()(size_t range) {
        return (size_t)((double)rand()*range/(RAND_MAX+1.0));
    }
    // gibt eine Pseudo-Zufallszahl zwischen 0 und grenze-1 zurück
    size_t operator()() {
        return (size_t)((double)rand()*grenze/(RAND_MAX+1.0));
    }
private:
    size_t grenze;
}; #endif
```

Ein `Random`-Objekt wird mit dem Grenzwert initialisiert. Gegebenenfalls kann mit `setSeed()` ein Anfangswert für eine andere Zufallsfolge eingestellt werden. Die Typumwandlung in `double` bei der Berechnung vermeidet Überlaufprobleme. Durch Überladen des `()`-

Operators kann das Objekt wie eine Funktion benutzt werden: Falls die zweite, parameterlose Form des Funktionsoperators `operator()()` gewünscht wird, kann der Bereich im Konstruktor eingestellt werden.

Listing 24.68: Anwendung des Zufallszahlengenerators

```
// cppbuch/k24/zufallszahlen/random.cpp
#include<iostream>
#include"Random.h"
using namespace std;

int main() {
    Random zufall_1;
    for(int i=0; i < 5; ++i) { // 5 Zufallszahlen zwischen 0 und 999 ausgeben
        cout << zufall_1(1000) << endl;
    }

    Random zufall_2(1000);    // andere Möglichkeit
    for(int i=0; i < 5; ++i) {
        cout << zufall_2() << endl;
    }
}
```

Die zweite Form wird von manchen Algorithmen der STL benutzt, siehe zum Beispiel `generate_n()` in Abschnitt 24.3.3.



Hinweis

Weil `Random` in den Beispielen häufig benutzt wird, wurde die Datei `Random.h` in das `include`-Verzeichnis der Beispiele aufgenommen.

Gleichverteilung

Die Erzeugung der Zufallszahlen wird von einem Generator übernommen. Es gibt nach [ISOC++] mehrere Generatoren; der gewählte ist `default_random_engine`. Der Gleichverteilung `uniform_int_distribution` wird der gewünschte Bereich übergeben. Die Zufallszahl wird durch Aufruf des Verteilungsobjekts erzeugt, wobei dem Funktionsobjekt der Generator als Parameter übergeben wird. Im folgenden Beispiel werden 100000 Zufallszahlen zwischen 10 und 20 erzeugt. Zur Kontrolle werden die Häufigkeiten für jeden Wert und der berechnete Mittelwert ausgegeben

Listing 24.69: Erzeugung gleichverteilter Zufallszahlen

```
// cppbuch/k24/zufallszahlen/verteilungen/gleichverteilung.cpp
#include <iostream>
#include <ctime>
#include <random>
#include <vector>
using namespace std;

int main() {
    const int MIN = 10;
```

```

const int MAX = 20;
static_assert(MAX > MIN, "MAX muss > MIN sein!");
const size_t ITERATIONEN = 100000;
vector<size_t> haeufigkeit(MAX-MIN+1, 0);
uniform_int_distribution<> verteilung(MIN, MAX); // <>: Vorgabe ist int
default_random_engine generator;
//generator.seed(time(NULL)); // // ohne '//': keine Reproduzierbarkeit
// Zufallszahl erzeugen, dabei die Häufigkeit hochzählen:
for(size_t i = 0; i < ITERATIONEN; ++i) {
    ++haeufigkeit[verteilung(generator)-MIN];
}
// Häufigkeiten und Mittelwert ausgeben:
int summe = 0.0;
for(size_t i = 0; i < haeufigkeit.size(); ++i) {
    int wert = (int)i+MIN;
    summe += wert*haeufigkeit[i];
    cout << wert << ": " << haeufigkeit[i] << endl;
}
cout << "Mittelwert: " << (double)summe/ITERATIONEN << endl;
}

```

Normalverteilung

Neben der Gleichverteilung ist die Normalverteilung am bekanntesten, charakterisiert durch die Gaußsche Glockenkurve. Sie wird insbesondere in der Messtechnik verwendet. So gehorchen die Abweichungen vom Sollmaß bei der industriellen Herstellung von Werkstücken der Normalverteilung. Das folgende Beispiel berechnet 1000000 normalverteilte Zufallszahlen, wobei zur Veranschaulichung das Ergebnis auf der Konsole ausgegeben wird (Abbildung 24.3).

Listing 24.70: Erzeugung normalverteilter Zufallszahlen

```

// cppbuch/k24/zufallszahlen/verteilungen/normalverteilung.cpp
#include <iostream>
#include <ctime>
#include <random>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    const double MIN = -3.0; // darzustellender Bereich
    const double MAX = 3.0;
    const size_t INTERVALLE = 80; // Auflösung der Ausgabe (Spalten)
    const size_t ZEILEN = 20; // Auflösung der Ausgabe (Zeilen)
    const double INTERVALLBREITE = (MAX-MIN)/INTERVALLE;
    const size_t ITERATIONEN = 1000000;

    vector<size_t> haeufigkeit(INTERVALLE, 0);
    normal_distribution<> gauss(0.0, 1.0); // Mittelwert, Standardabweichung
    default_random_engine generator;
    //generator.seed(time(NULL)); // // ohne '//': keine Reproduzierbarkeit
    // Zufallszahl erzeugen, dabei die Häufigkeit hochzählen

```



```

for(size_t i = 0; i < ITERATIONEN; ++i) {
    double wert = gauss(generator);
    if(wert > MIN && wert < MAX) {
        ++haeufigkeit[ (size_t)((wert - MIN)/INTERVALLBREITE)];
    } // else.. Außenbereiche ignorieren
}
// Darstellung der Glockenkurve auf der Konsole
size_t max = *max_element(haeufigkeit.begin(), haeufigkeit.end());
for(size_t i = 0; i < ZEILEN; ++i) {
    for(size_t j = 1; j < INTERVALLE; ++j) {
        double grenze = (double)haeufigkeit[j]*ZEILEN/max + 0.5;
        cout << (grenze <= (ZEILEN-i) ? ' ' : '*');
    }
    cout << endl;
}
}

```

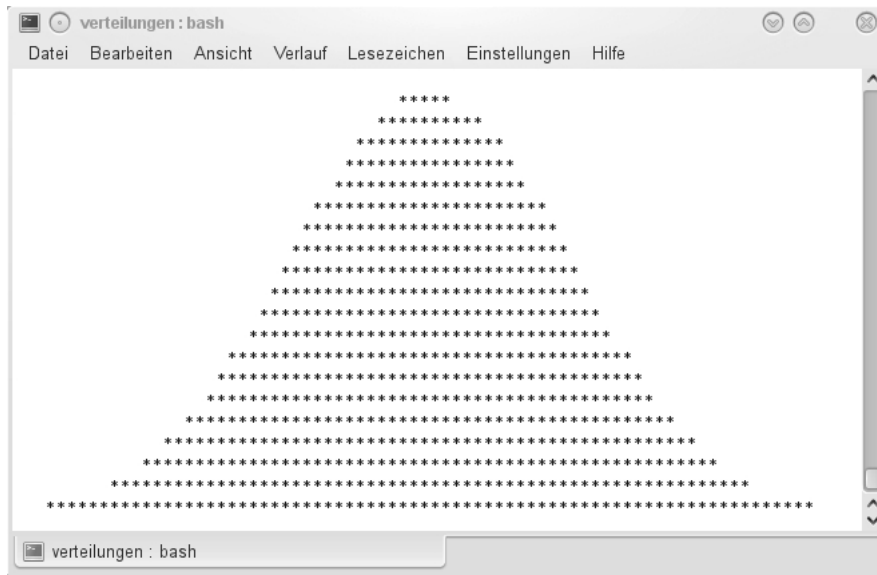


Abbildung 24.3: Glockenkurve auf der Konsole

24.12.4 for_each – Auf jedem Element eine Funktion ausführen

Der Algorithmus `for_each` bewirkt, dass auf jedem Element eines Containers eine Funktion ausgeführt wird. Die Deklaration ist:

```

template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);

```

`f` kann sowohl eine Funktion (aber keine Methode einer Klasse) als auch ein Funktionsobjekt sein und wird nach Gebrauch zurückgegeben. Beispiel:

```
template<typename T>
void anzeige(const T& x) {
    std::cout << x << std::endl;
}
// Anzeige aller Elemente eines Containers c. Für die Elemente muss operator<<() definiert sein.
for_each(c.begin(), c.end(), anzeige);
```

24.12.5 Verschiedene Möglichkeiten, Container-Bereiche zu kopieren

Der Algorithmus `copy()` kopiert die Elemente eines Quellbereichs in den Zielbereich, wobei das Kopieren am Anfang oder am Ende der Bereiche (mit `copy_backward()`) beginnen kann. Falls der Zielbereich nicht überschrieben, sondern in ihn eingefügt werden soll, ist als Output-Iterator ein Iterator zum Einfügen (Insert-Iterator) zu nehmen. Der Algorithmus `copy()` ist immer dann zu nehmen, wenn Ziel- und Quellbereich sich nicht oder so überlappen, dass der Anfang des Quellbereichs im Zielbereich liegt. `result` muss anfangs auf den Anfang des Zielbereichs zeigen. Zur Verdeutlichung der Wirkungsweise sind hier ausnahmsweise nicht die Prototypen, sondern die vollständigen Definitionen gezeigt:

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                   OutputIterator result) {
    while (first != last) {
        *result++ = *first++;
    }
    return result;
}
```

Der Algorithmus `copy_backward()` ist immer dann zu nehmen, wenn Ziel- und Quellbereich sich so überlappen, dass der Anfang des Zielbereichs im Quellbereich liegt. `result` muss anfangs auf das Ende des Zielbereichs zeigen.

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
                                     BidirectionalIterator1 last,
                                     BidirectionalIterator2 result) {
    while (first != last) {
        *--result = *--last;
    }
    return result;
}
```

Auch hier gilt wie allgemein in der C++-Standardbibliothek, dass `last` nicht die Position des letzten Elements bezeichnet, sondern die Position nach dem letzten Element. `result` darf niemals zwischen `first` und `last` liegen. Wie Abbildung 24.4 zeigt, sind drei Fälle zu berücksichtigen:

- Die Bereiche sind voneinander vollständig getrennt. Die Bereiche können in demselben oder in verschiedenen Containern liegen. `result` zeigt auf den Beginn des Zielbereichs. `copy()` kopiert den Quellbereich beginnend mit `*first`. Zurückgegeben wird `result + (last - first)`, also die Position nach dem letzten Element des Zielbereichs.
- Die Bereiche überlappen sich so, dass der Zielbereich *vor* dem Quellbereich beginnt. `result` zeigt auf den Beginn des Zielbereichs. `copy()` kopiert den Quellbereich beginn-

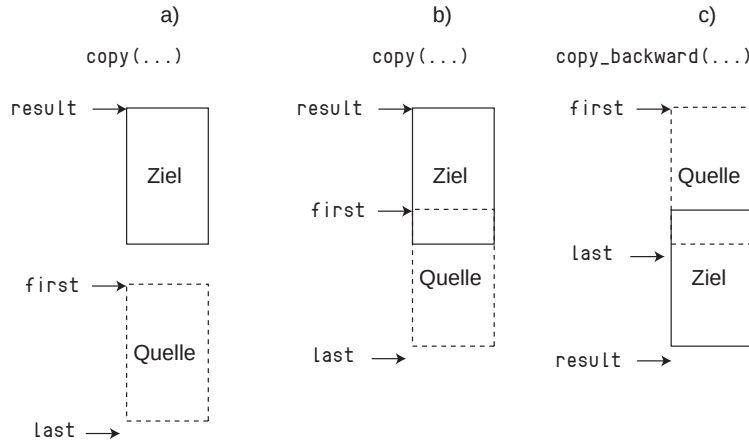


Abbildung 24.4: Kopieren ohne und mit Bereichsüberlappung

nend mit `*first`. Wie bei a) wird die Position nach dem letzten Element des Zielbereichs zurückgegeben.

- c) Die Bereiche überlappen sich so, dass der Zielbereich *mitten im* Quellbereich beginnt. Um die Daten nicht zu zerstören, muss vom Ende her beginnend kopiert werden. `result` zeigt auf die Position direkt nach dem *Ende* des Zielbereichs. `copy_backward()` kopiert den Quellbereich, indem zuerst `*(--last)` an die Stelle `--result` kopiert wird. Hier wird `result - (last - first)` zurückgegeben, also die Position des zuletzt kopierten Elements im Zielbereich.

```
// Beispiele:
// v1 nach v2 kopieren
copy(v1.begin(), v1.end(), v2.begin());

// v1 nach v2 kopieren, dabei am Ende beginnen
copy_backward(v1.begin(), v1.end(), v2.end());

// v1 nach cout kopieren, Separator *
ostream_iterator<int> ausgabe(cout, "*");
copy(v1.begin(), v1.end(), ausgabe);
```

Kopieren mit Bedingung

In Analogie zu anderen Algorithmen mit der Endung `_if` gibt es auch den Algorithmus `copy_if()`, der nur Elemente kopiert, die einer bestimmten Bedingung genügen. Der Prototyp ist

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator copy_if(InputIterator first, InputIterator last,
                      OutputIterator result, Predicate pred)
```

Im folgenden Beispiel werden alle Elemente des `container1` am Ende des `container2` eingefügt, sofern sie größer als 10 sind.

Listing 24.71: copy_if

```
// Auszug aus cppbuch/k24/vermischtes/copy/copy_if.cpp
int main() {
    typedef vector<int> Container;
    Container container1(20);
    iota(container1.begin(), container1.end(), 1);
    showSequence(container1);

    Container container2; // leeren Container anlegen
    // alle Elemente > 10 am Ende einfügen:
    copy_if(container1.begin(), container1.end(),
            back_inserter(container2),
            bind(greater<int>(), _1, 10));
    showSequence(container2);
}
```

Kopieren einer bestimmten Zahl von Elementen

Der Algorithmus `copy_n()` kopiert eine bestimmte Anzahl von Elementen. Der Prototyp ist

```
template<class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(InputIterator first, Size n, OutputIterator result);
```

Listing 24.72: copy_n

```
// Auszug aus cppbuch/k24/vermischtes/copy/copy_n.cpp
vector<int> v1(20);
iota(v1.begin(), v1.end(), 1);
vector<int> v2(20, 0);
// die ersten 10 Elemente kopieren
copy_n(v1.begin(), 10, v2.begin());
}
```

24.12.6 Vertauschen von Elementen, Bereichen und Containern

Der Algorithmus `swap()` vertauscht Elemente von Containern oder Container selbst. Er tritt in vier Varianten auf:

- `swap()` vertauscht zwei einzelne Elemente. Die beiden Elemente können in verschiedenen, in demselben oder in keinem Container sein.

```
template<typename T>
void swap(T& a, T& b);
```

- `iter_swap()` nimmt zwei Iteratoren und vertauscht die dazugehörigen Elemente. Die beiden Iteratoren können zu verschiedenen oder zu demselben Container gehören.

```
template<class ForwardIterator1,
        class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

```
// erstes und letztes Element per Iterator vertauschen:
vector<int>::iterator first = v.begin(),
                    last = v.end();
--last;
iter_swap(first, last);          // Tausch
```

- `swap_ranges()` vertauscht zwei Bereiche.

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1,
                            ForwardIterator1 last1,
                            ForwardIterator2 first2);
```

`first1` zeigt auf den Anfang des ersten Bereichs, `last1` auf die Position nach dem letzten Element des ersten Bereichs. Der Anfang des zweiten Bereichs wird durch `first2` gegeben. Die Anzahl der auszutauschenden Elemente wird durch die Größe des ersten Bereichs gegeben. Die Bereiche können in demselben Container sein, dürfen sich jedoch nicht überlappen. `swap_ranges()` gibt einen Iterator auf das Ende des zweiten Bereichs zurück.

```
// Vertauschen der beiden Hälften eines Vektors v
// mit einer geradzahligen Anzahl von Elementen
vector<double>::iterator Mitte = v.begin()+v.size()/2;
swap_ranges(v.begin(), Mitte, Mitte);
```

- `swap()` ist für diejenigen Container spezialisiert, die eine Methode `swap()` zum Vertauschen bereitstellen, also `deque`, `list`, `vector`, `set`, `map`, `multiset` und `multimap`. Diese Methoden sind sehr schnell ($O(1)$), weil nur Verwaltungsinformationen ausgetauscht werden². `swap()` ruft intern die Methoden der Container auf.

24.12.7 Elemente transformieren

Wenn es darum geht, nicht nur etwas zu kopieren, sondern dabei gleich umzuwandeln, ist `transform()` der richtige Algorithmus. Die Umwandlung kann sich auf nur ein Element oder auf zwei Elemente gleichzeitig beziehen. Dementsprechend gibt es zwei überladene Formen:

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
                        OutputIterator result, UnaryOperation op);
```

Hier wird auf jedes Element des Bereichs von `first` bis ausschließlich `last` die Operation `op` angewendet und das Ergebnis in den mit `result` beginnenden Bereich kopiert. `result` darf identisch mit `first` sein, wobei dann die Elemente durch die transformierten ersetzt werden. Der Rückgabewert ist ein Iterator auf die Position nach dem Ende des Zielbereichs. Im Beispiel

```
string s("ABC123");
transform(s.begin(), s.end(), s.begin(), tolower);
```

werden alle Großbuchstaben eines C-Strings in Kleinbuchstaben umgewandelt. Die Funktion `tolower()` ist im Header `<cctype>` deklariert.

² Ausnahme: `array`, vgl. Seite 765

```
template<class InputIterator1, class InputIterator2, class OutputIterator,
        class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, OutputIterator result,
                        BinaryOperation bin_op);
```

In der zweiten Form werden zwei Bereiche betrachtet. Der erste ist das Intervall $[first1, last1)$, der zweite das Intervall $[first2, first2 + last1 - first1)$, das heißt, der zweite Bereich ist genauso groß wie der erste. Die Operation `bin_op` nimmt jeweils ein Element aus jedem der zwei Bereiche und legt ihr Ergebnis in `result` ab. `result` darf identisch mit `first1` oder `first2` sein, wobei dann die Elemente durch die transformierten ersetzt werden. Der Rückgabewert ist ein Iterator auf die Position nach dem Ende des Zielbereichs. Damit sind komplexere Operationen möglich.

Im Beispiel werden jeweils zwei Strings verkettet. Dabei wird sowohl eine unäre Operation als Funktion eingesetzt wie auch eine binäre Operation als Funktor. Erstere wandelt alle Buchstaben eines C++-Strings in Großbuchstaben um und gibt den veränderten String zurück. Der Funktor verkettet zwei String-Objekte miteinander und fügt dabei das Wort »und« ein.

Listing 24.73: `transform()`

```
// cppbuch/k24/vermischtes/transform.cpp
#include<algorithm>
#include<cstddef>
#include<cctype>
#include<locale>
#include<string>
#include<vector>
#include<showSequence.h>

std::string upper_case(std::string s) { // unäre Operation als Funktion
    for(size_t i = 0; i < s.length(); ++i)
        s[i] = toupper(s[i]);
    return s;
}

class Verketten { // binäre Operation als Funktor
public:
    std::string operator()(const std::string& a, const std::string& b) const {
        return a + " und " + b;
    }
};

using namespace std;

int main() {
    locale::global(locale("de_DE")); // falls Umlaute vorkommen
    const size_t ANZAHL = 3;
    vector<string> maedels(ANZAHL), jungs(ANZAHL), paare(ANZAHL);
    maedels[0] = "Annabella";
    maedels[1] = "Scheherazade";
    maedels[2] = "Julia";
```

```

jungs[0] = "Nikolaus";
jungs[1] = "Amadeus";
jungs[2] = "Romeo";
transform(jungs.begin(), jungs.end(),
          jungs.begin(), // Ziel == Quelle
          upper_case);   // in Großbuchstaben wandeln
transform(maedels.begin(), maedels.end(),
          jungs.begin(), paare.begin(), Verketten());
showSequence(paare, "", "\n"); // gebildete Paare ausgeben
}

```

24.12.8 Ersetzen und Varianten

Der Algorithmus `replace()` ersetzt in einer Sequenz jeden vorkommenden Wert `old_value` durch `new_value`. Alternativ ist mit `replace_if()` eine bedingungsgesteuerte Ersetzung mit einem unären Prädikat möglich:

```

template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);

```

```

template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value);

```

Erstmalig treten nun auch kopierende Varianten von Algorithmen auf, die sich im Namen durch ein hinzugefügtes `_copy` unterscheiden:

```

template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& old_value,
                           const T& new_value);

```

```

template<class InputIterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
                              OutputIterator result, Predicate pred,
                              const T& new_value);

```

Das Beispiel zeigt alle vier Varianten:

Listing 24.74: `replace()` mit Varianten

```

// cppbuch/k24/vermishtes/replace.cpp
#include<algorithm>
#include<string>
#include<vector>
#include<showSequence.h>

// Unäres Prädikat als Funktor
class Zitrusfrucht {
public:
    bool operator()(const std::string& a) const {
        return a == "Zitrone" || a == "Apfelsine" || a == "Limonen";
    }
}

```

```

);
using namespace std;

int main() {
    vector<string> obstkorb(3), kiste(3);
    obstkorb[0] = "Apfel";
    obstkorb[1] = "Apfelsine";
    obstkorb[2] = "Zitrone";
    showSequence(obstkorb); // Apfel Apfelsine Zitrone
    cout << "replace: Apfel durch Quitte ersetzen:\n";
    replace(obstkorb.begin(), obstkorb.end(),
            string("Apfel"), string("Quitte"));
    showSequence(obstkorb); // Quitte Apfelsine Zitrone
    cout << "replace_if: Zitrusfrüchte durch Pflaumen ersetzen:\n";
    replace_if(obstkorb.begin(), obstkorb.end(),
               Zitrusfrucht(), string("Pflaume"));
    showSequence(obstkorb); // Quitte Pflaume Pflaume
    cout << "replace_copy: kopieren + ersetzen der Pflaumen durch Limonen:\n";
    replace_copy(obstkorb.begin(), obstkorb.end(),
                 kiste.begin(), string("Pflaume"), string("Limone"));
    showSequence(kiste); // Quitte Limone Limone
    cout << "replace_copy_if: kopieren und ersetzen "
           "der Zitrusfrüchte durch Tomaten:\n";
    replace_copy_if(kiste.begin(), kiste.end(),
                    obstkorb.begin(), Zitrusfrucht(), string("Tomate"));
    showSequence(obstkorb); // Quitte Tomate Tomate
}

```

24.12.9 Elemente herausfiltern

Der Algorithmus entfernt alle Elemente aus einer Sequenz, die gleich einem Wert `value` sind beziehungsweise einem Prädikat `pred` genügen. Hier sind die Prototypen einschließlich der kopierenden Varianten aufgeführt:

```

template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first,
                      ForwardIterator last,
                      const T& value);

```

```

template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first,
                        ForwardIterator last,
                        Predicate pred);

```

```

template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first,
                          InputIterator last,
                          OutputIterator result,
                          const T& value);

```

```

template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first,

```



```
InputIterator last,
OutputIterator result,
Predicate pred);
```

»Entfernen eines Elements« bedeutet in Wirklichkeit, dass alle nachfolgenden Elemente um eine Position nach links rücken. Das letzte Element wird bei Entfernen eines einzigen Elements verdoppelt, weil eine Kopie davon dem vorhergehenden Platz zugewiesen wird. `remove()` gibt einen Iterator auf das nunmehr verkürzte Ende der Sequenz zurück.

Dabei ist zu beachten, dass die gesamte Länge der Sequenz sich nicht ändert! Es wird keine Neuordnung des Speicherplatzes vorgenommen. Falls nicht kopiert wird, enthält der Bereich zwischen dem zurückgegebenen Iterator und `last` nur noch bedeutungslos gewordene Elemente. Das Beispiel zeigt nur die nicht-kopierenden Varianten:

Listing 24.75: `remove()` mit Varianten

```
// cppbuch/k24/vermishtes/remove.cpp
#include<iostream>
#include<algorithm>
#include<vector>
#include<iterator>
#include<cstring>
#include<numeric>
#include<showSequence.h>

// Umlaute werden nicht berücksichtigt
bool istVokal(char c) {
    return std::strchr("aeiouyAEIOUY", c) != 0;
}
using namespace std;

int main() {
    vector<char> v(26);
    // Alphabet mit Kleinbuchstaben erzeugen
    iota(v.begin(), v.end(), 'a');
    showSequence(v, "\n", "");
    cout << "remove 't': ";
    vector<char>::iterator last = remove(v.begin(), v.end(), 't');
    // last = neues Ende nach der Verschiebung
    // v.end() bleibt unverändert!
    // Die Sequenz wird hier nicht mit showSequence() angezeigt,
    // weil nur die Elemente von begin() bis last signifikant sind
    ostream_iterator<char> ausgabe(cout, "");
    copy(v.begin(), last, ausgabe); // abcdefghijklmnopqrsuvwxyz (t fehlt)
    cout << endl;
    last = remove_if(v.begin(), last, istVokal);
    cout << "nur noch Konsonanten übrig: ";
    copy(v.begin(), last, ausgabe); // bcd fghijklmnopqrsvwxyz
    cout << endl;
    cout << "Vollständige Sequenze bis end() einschließlich "
           "bedeutungsloser Elemente am Ende: ";
    showSequence(v, "\n", "");
}
```

24.12.10 Grenzwerte von Zahltypen

Im Header `<limits>` wird die Template-Klasse `numeric_limits` definiert. Sie hat Spezialisierungen für die ganzzahligen Grunddatentypen `bool`, `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, sowie für die Gleitkommazahltypen `float`, `double` und `long double`. Für diese Grunddatentypen beschreiben die Spezialisierungen verschiedene implementationsabhängige Funktionen und Eigenschaften, die alle `public` sind und von denen die wichtigsten in der Tabelle 24.4 aufgelistet sind. Eine Anwendung wird in Abschnitt 1.6.3 (Seite 47) gezeigt.

Tabelle 24.4: `<limits>`: Attribute und Funktionen (Auszug). NaN = not a number, T = Typ der Zahl

Schnittstelle	Bedeutung
<code>bool is_specialized</code>	<code>true</code> nur für Grunddatentypen, für die eine Spezialisierung vorliegt, <code>false</code> für alle anderen
<code>T min()</code>	minimal möglicher Wert
<code>T max()</code>	maximal möglicher Wert
<code>int radix</code>	Zahlenbasis, normal 2 (Ausnahme z.B. BCD-Zahlen)
<code>int digits</code>	Ganzzahlen: Anzahl der Bits (ohne Vorzeichen-Bit). Gleitkommazahlen: Anzahl der Bits in der Mantisse. Annahme: <code>radix == 2</code>
<code>int digits10</code>	Anzahl signifikanter Dezimalziffern bei Gleitkommazahlen, zum Beispiel 6 bei <code>float</code> , 10 bei <code>double</code>
<code>bool is_signed</code>	<code>true</code> bei vorzeichenbehafteten Zahlen
<code>bool is_integer</code>	<code>true</code> bei Ganzzahltypen
<code>bool is_exact</code>	<code>true</code> bei exakten Zahlen, z.B. ganze oder rationale Zahlen – nicht aber Gleitkommazahlen
<code>T epsilon()</code>	kleinster positiver Wert x , für den die Maschine die Differenz zwischen 1.0 und $(1.0 + x)$ noch unterscheidet
<code>T round_error()</code>	maximaler Rundungsfehler
<code>int min_exponent</code>	kleinster negativer Exponent für Gleitkommazahlen
<code>int min_exponent10</code>	kleinster negativer 10er-Exponent für Gleitkommazahlen (≤ -37)
<code>int max_exponent</code>	größtmöglicher Exponent für Gleitkommazahlen
<code>int max_exponent10</code>	größtmöglicher 10er-Exponent für Gleitkommazahlen ($\geq +37$)
<code>T infinity()</code>	Repräsentation von $+\infty$, falls vorhanden
<code>bool has_infinity</code>	<code>true</code> , falls der Zahltyp eine Repräsentation für $+\infty$ hat
<code>bool is_iec559</code>	<code>true</code> , falls der Zahltyp dem IEC 559 (= IEEE 754)-Standard genügt.
<code>bool is_bounded</code>	<code>true</code> für alle Grunddatentypen, <code>false</code> wenn die Menge der darstellbaren Werte unbegrenzt ist, z.B. bei Typen mit beliebiger Genauigkeit.
<code>bool is_modulo</code>	<code>true</code> , falls bereichsüberschreitende Operationen wieder eine gültige Zahl ergeben. Z.B. ergibt die Addition einer Zahl auf die größtmögliche Integerzahl bei den meisten Maschinen wieder eine Integerzahl, die kleiner als die größtmögliche ist. Dies gilt im Allgemeinen nicht für Gleitkommazahlen.
<code>round_style</code>	Art der Rundung Ganzzahlen: <code>round_toward_zero</code> (=0) Gleitkommazahlen: <code>round_to_nearest</code> (=1)

24.12.11 Minimum und Maximum

Die inline-Templates `min()` und `max()` geben jeweils das kleinere (bzw. das größere) von zwei Elementen zurück. Bei Gleichheit wird das erste Element zurückgegeben. Die Prototypen sind:

```
template<typename T>  
const T& min(const T& a, const T& b);
```

```
template<typename T, class Compare>  
const T& min(const T& a, const T& b, Compare comp);
```

```
template<typename T>  
const T& max(const T& a, const T& b);
```

```
template<typename T, class Compare>  
const T& max(const T& a, const T& b, Compare comp);
```

`minmax()` gibt ein `pair`-Objekt zurück, das den kleineren Wert an der Stelle `first` und den größeren an der Stelle `second` enthält:

```
template<typename T> pair<const T&, const T&>  
minmax(const T& a, const T& b);
```

```
template<typename T, class Compare>  
pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

25

Ein- und Ausgabe

Dieses Kapitel behandelt die folgenden Themen:

- Datei- und Verzeichnisoperationen
- Formatierte Daten schreiben
- Formatierte Daten lesen
- Array als Block lesen und schreiben

25.1 Datei- und Verzeichnisoperationen

Wie eine Datei kopiert werden kann, wird in Kapitel [2](#) beschrieben. Andere Dateioperationen, wie Löschen oder Umbenennen von Dateien und Verzeichnissen, das Anlegen eines Verzeichnisses und mehr, werden nicht vom C++-Standard unterstützt. Es gibt im Wesentlichen zwei Möglichkeiten der Realisierung:

- Nutzen der entsprechenden Funktionen der Programmiersprache C. C ist eine Unter-
menge von C++. Der Nachteil dieses Vorgehens ist die mangelnde Portabilität. So sind
gelegentlich unter Windows andere Header-Dateien einzubinden als unter Linux, wie
unten in Abschnitt [25.1.3](#) zu sehen.

- Die bereits erwähnte Boost-Library war schon oft Vorbild – große Teile sind in den C++-Standard eingeflossen. Boost bietet die gewünschten Funktionen mit dem Vorteil, dass die Quellprogramme ohne Änderung auf allen Betriebssystemen, auf denen Boost installiert ist, übersetzt werden können.

Die Bibliothek `Boost.Filesystem` ist in den Technical Report 2 (TR2) übernommen worden, der vermutlich in einigen Jahren Bestandteil des C++-Standards werden wird. In diesem Abschnitt werden die wichtigsten Dateioperationen am Beispiel gezeigt, wobei ein Teil C-basiert ist und ein anderer Teil Boost nutzt.

25.1.1 Datei oder Verzeichnis löschen

Das folgende Programm kann eine Datei oder ein leeres Verzeichnis löschen. Die C-Funktion `remove()` (Header `<cstdio>`) gibt bei Erfolg 0 zurück, ansonsten einen anderen Wert. Im C-Standard [ISOC] nicht gefordert, aber hilfreich, ist die Ablage eines Fehlercodes in der globalen Variablen `errno`. Die Funktion `strerror(int)` gibt einen zum Fehlercode passenden, implementationsabhängigen C-String zurück.

Listing 25.1: Datei oder leeres Verzeichnis löschen

```
// cppbuch/k25/files/loeschen.cpp
#include<cstdio> // remove()
#include<cerrno> // errno
#include<cstring> // strerror(int)
#include<iostream>
using namespace std;

int main(int argc, char*argv[]) {
    if(argc != 2) {
        cout << "Datei oder leeres Verzeichnis löschen\n"
              << "Gebrauch: loeschen.exe name" << endl;
    }
    else {
        if(remove(argv[1]) != 0) {
            cerr << "Löschen von " << argv[1]
                  << " fehlgeschlagen: " << strerror(errno) << endl;
        }
    }
}
```

Das Programm kann kein gefülltes Verzeichnis löschen. `Boost.Filesystem` hat ebenfalls eine Funktion `remove()`, aber um die Funktionalität zu variieren, löscht das nächste Programm ein nicht-leeres Unterverzeichnis. Im Fehlerfall wird nicht `errno` gesetzt, sondern eine Exception geworfen, wie in der Boost-Library üblich. Im Programm lernen Sie auch die Funktionen `exists()` und `is_directory()` kennen.

Listing 25.2: Verzeichnis löschen

```
// cppbuch/k25/files/boost/verzeichnisloeschen.cpp
#include<boost/filesystem/operations.hpp>
#include<iostream>
#include<string>
using namespace std;
```

```

namespace bf=boost::filesystem; // Abkürzung

int main(int argc, char*argv[]) {
    if(argc != 2) {
        cout << "Unterverzeichnis löschen\n"
              << "Gebrauch: verzeichnisloeschen.exe name" << endl;
    } else if(argv[1][0] == '.') {
        cout << ". nicht erlaubt. Nur für Unterverzeichnis aufrufen!\n";
    }
    else {
        try {
            bf::path pfad(argv[1]);
            if(bf::exists(pfad) && bf::is_directory(pfad)) {
                bf::remove_all(pfad);
            }
            else {
                cout << "Unterverzeichnis " << argv[1] << " existiert nicht!\n";
            }
        }
        catch(const exception& e) {
            cerr << "Löschen des Verzeichnisses " << argv[1]
                  << " fehlgeschlagen: " << e.what() << endl;
        }
    }
}

```

25.1.2 Datei oder Verzeichnis umbenennen

Die C-Funktion `rename()` benennt eine Datei oder ein Verzeichnis um. Wie oben dient `strerror()` zur Fehlerdokumentation.

Listing 25.3: Datei oder Verzeichnis umbenennen

```

// cppbuch/k25/files/umbenennen.cpp
#include<cstdio> // rename()
#include<cerrno> // errno
#include<cstring> // strerror(int)
#include<iostream>
using namespace std;

int main(int argc, char*argv[]) {
    if(argc != 3) {
        cout << "Datei oder Verzeichnis umbenennen\n"
              << "Gebrauch: umbenennen.exe alternname neuernamen" << endl;
    }
    else {
        if(rename(argv[1], argv[2]) != 0) {
            cerr << "Umbenennen von " << argv[1]
                  << " fehlgeschlagen: " << strerror(errno) << endl;
        }
    }
}

```

25.1.3 Verzeichnis anlegen

Das folgende Programm zum Anlegen eines Verzeichnisses zeigt betriebssystemabhängige Unterschiede. Zum einen sind die Header verschieden, zum anderen müssen unter Unix die Dateizugriffsrechte angegeben werden. Die Rechte sind schreiben (4), lesen (2), ausführen (1), wobei die Ziffern für jede Gruppe (Eigentümer, Gruppe, alle) addiert werden. Üblich ist die Darstellung als Oktalzahl. 0755 bedeutet: Der Eigentümer darf alles, die Gruppe und der Rest der Welt dürfen das Verzeichnis lesen und ausführen, aber nicht verändern (schreiben). »Ausführen« meint bei einem Verzeichnis, den Inhalt lesen zu können, wie etwa auch darunterhängende Verzeichnisse. Die Oktalzahl wird auch vom Unix-Befehl `chmod` verstanden. Windows hat eine andere Art der Rechtevergabe. Eine Angabe beim Anlegen des Verzeichnisses mit `mkdir()` ist nicht vorgesehen. Im Programm wird die unterschiedliche Behandlung durch das Makro `WIN32` gesteuert.

Listing 25.4: Verzeichnis anlegen mit `cstdio`

```
// cppbuch/k25/files/verzanlegen.cpp
#include<cstdio> // mkdir()
#include<cerrno> // errno
#include<cstring> // strerror(int)
#include<iostream>
#ifdef WIN32
    #include<direct.h>
#else
    #include<sys/stat.h>
#endif
using namespace std;

int main(int argc, char*argv[]) {
    if(argc != 2) {
        cout << "Verzeichnis anlegen\n"
              << "Gebrauch: verzanlegen.exe name" << endl;
    }
    else {
#ifdef WIN32
        int fehler = mkdir(argv[1]);
#else
        int fehler = mkdir(argv[1], 0755);
#endif
        if(fehler != 0) {
            cerr << "Anlegen des Verzeichnisses " << argv[1]
                  << " fehlgeschlagen: " << strerror(errno) << endl;
        }
    }
}
```

Die umständliche zweifache Makro-Abfrage entfällt in dem folgenden Programm, das die Boost-Funktion `create_directory()` nutzt:

Listing 25.5: Verzeichnis anlegen mit Boost.Filesystem

```
// cppbuch/k25/files/boost/verzeichnisanlegen.cpp
#include<boost/filesystem/operations.hpp>
#include<iostream>
```

```
using namespace std;

int main(int argc, char*argv[]) {
    if(argc != 2) {
        cout << "Verzeichnis anlegen\n"
              << "Gebrauch: verzeichnisanlegen.exe name" << endl;
    }
    else {
        try {
            boost::filesystem::create_directory(argv[1]);
        }
        catch(const exception& e) {
            cerr << "Anlegen des Verzeichnisses " << argv[1]
                  << " fehlgeschlagen: " << e.what() << endl;
        }
    }
}
```

25.1.4 Verzeichnis anzeigen

Das folgende Programm zeigt ein Verzeichnis an, indem ein `directory_iterator` in STL-Manier über das Verzeichnis wandert. Neben dem Namen werden auch die Größe der Dateien und das letzte Modifikationsdatum ermittelt und angezeigt.

Listing 25.6: Verzeichnis anzeigen

```
// cppbuch/k25/files/boost/verzeichnisanzeigen.cpp
#include<boost/filesystem/operations.hpp>
#include<iostream>
#include<string>
using namespace std;
namespace bf=boost::filesystem;

int main(int argc, char*argv[]) {
    if(argc > 2) {
        cout << "Verzeichnis anzeigen\n"
              << "Gebrauch: verzeichnisanzeigen.exe [name]" << endl;
    }
    else {
        string verz("."); // aktuelles Verzeichnis
        if(argc > 1) {
            verz = argv[1];
        }
        try {
            bf::path p(fad(verz);
            bf::directory_iterator di(pfad), ende;
            while(di != ende) {
                bf::path p = di->path();
                bf::file_status status = di->status();
                cout << p.file_string() << '\t';
                if(bf::is_directory(status)) {
                    cout << " (Verzeichnis)";
                }
            }
        }
    }
}
```



```

        else {
            cout << bf::file_size(p) << " Bytes";
        }
        time_t t = bf::last_write_time(p);
        cout << '\t' << ctime(&t);
        ++di;
    }
}
catch(const exception& e) {
    cerr << "Anzeige des Verzeichnisses " << argv[1]
        << " fehlgeschlagen: " << e.what() << endl;
}
}
}

```

25.1.5 Verzeichnisbaum anzeigen

Mit den obigen Beispielen liegt alles vor, um auch einen Verzeichnisbaum bearbeiten zu können. Als Beispiel dient ein Programm zur Anzeige der Baumstruktur, ähnlich wie sie das Unix-Programm `tree` liefert (vergleiche Seite 605).

Listing 25.7: Verzeichnisbaum anzeigen

```

// cppbuch/k25/files/boost/tree/main.cpp
#include<iostream>
#include<string>
#include"tree.h"
using namespace std;

int main(int argc, char*argv[]) {
    if(argc > 2) {
        cout << "Verzeichnisbaum anzeigen\n Gebrauch: tree.exe [name]" << endl;
    }
    else {
        string verz("."); // aktuelles Verzeichnis
        if(argc > 1) {
            verz = argv[1];
        }
        try {
            baumAnzeigen(verz);
        }
        catch(const exception& e) {
            cerr << argv[1] << ": Fehler: " << e.what() << endl;
        }
    }
}

```

Die Ablauflogik liegt in der Funktion `baumAnzeigen(verzeichnis)`. Die Header-Datei ist trivial:

Listing 25.8: `tree.h`

```

// cppbuch/k25/files/boost/tree/tree.h
#ifndef TREE_H

```

```
#define TREE_H
#include<string>
void baumAnzeigen(const std::string& verz);
#endif
```

Die Funktion `baumAnzeigen()` ruft eine überladene Funktion gleichen Namens auf, die sich selbst aufruft, wobei der Zähler für die Einrückungsebene (`level`) erhöht wird. Der Verzeichnisbaum wird also rekursiv durchwandert. Die Rekursion bricht ab, wenn es auf einer Ebene keine Verzeichnisse mehr gibt. Damit die rekursive Funktion nicht direkt aufgerufen werden kann, fehlt sie in *tree.h* und ist in einem anonymen Namespace angelegt.

Listing 25.9: *tree.cpp*

```
// cppbuch/k25/files/boost/tree/tree.cpp
#include<iostream>
#include<stdexcept>
#include<boost/filesystem/operations.hpp>
#include"tree.h"
namespace bf = boost::filesystem;

namespace {
    void baumAnzeigen(const bf::path& p, int level) {
        bf::directory_iterator di(p), ende;
        while(di != ende) {
            for(int i = 0; i < level; ++i) {
                std::cout << " | ";
            }
            std::cout << " |-- " << di->filename() << std::endl;
            if(bf::is_directory(di->status())) {
                baumAnzeigen(di->path(), level+1);
            }
            ++di;
        }
    }
} // anonymen Namespace

void baumAnzeigen(const std::string& verz) {
    bf::path pfad(verz);
    if(bf::is_directory(pfad)) {
        std::cout << pfad.filename() << std::endl;
        baumAnzeigen(pfad, 0);
    }
    else {
        throw std::runtime_error(" ist kein Verzeichnis!");
    }
}
```

Das Programm, ohne Argumente im Verzeichnis *cppbuch/k25/files/boost* aufgerufen, gibt aus (der Punkt am Anfang steht für das aktuelle Verzeichnis):

```

|-- verzeichnisanzeigen.cpp
|-- verzeichnisloeschen.cpp
|-- README.txt
|-- tree
|   |-- main.o
|   |-- tree.o
|   |-- README.txt
|   |-- tree.cpp
|   |-- tree.exe
|   |-- tree.h
|   |-- makefile
|   |-- main.cpp
|-- verzeichnisanlegen.cpp
|-- makefile

```

25.2 Tabelle formatiert ausgeben

Als Beispiel soll eine Tabelle von Sinus- und Kosinuswerten formatiert ausgegeben werden. Die benötigten Informationen finden Sie zum Nachlesen auf Seite 378. Durch das `fixed`-Bit wird die Darstellung mit Dezimalpunkt erreicht, und mit `precision(6)` wird die Anzahl der Nachkommastellen auf 6 festgelegt. Um ein gleichmäßiges Bild zu erzeugen, werden nachfolgende Nullen ausgegeben (`showpoint` wirkt nur in Verbindung mit `fixed`):

Listing 25.10: Formatierte Ausgabe

```

// cppbuch/k25/tabelle/tabelle.cpp
#include<iostream>
#include<cmath> // cos(), sin(), Konstante M_PI für π
// manche Compiler stellen Konstanten wie M_PI nicht zur Verfügung
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
using namespace std;

int main() {
    cout << "Grad    sin(x)    cos(x)\n";
    cout.setf(ios::showpoint|ios::fixed, ios::floatfield);
    cout.precision(6);
    for(int grad = 0; grad <= 90; grad += 10) {
        // Grad in Bogenmaß umwandeln
        const double rad = static_cast<double>(grad)/180.0*M_PI;
        cout.width(4);    cout << grad;
        cout.width(12);   cout << sin(rad);
        cout.width(12);   cout << cos(rad) << endl;
    }
}

```

Das Programm gibt aus:

Grad	$\sin(x)$	$\cos(x)$
0	0.000000	1.000000
10	0.173648	0.984808
20	0.342020	0.939693
...	usw.	

25.3 Formatierte Daten lesen

Die Art des Einlesens hängt von der Formatierung ab und auch von den Objekten, die die Daten aufnehmen sollen. Es bieten sich zwei verschiedene Möglichkeiten an, von denen die zweite vorgestellt wird:

- Einlesen einer Zeichenkette, die dann ausgewertet wird.
- Überladen des Eingabeoperators `>>` für benutzerdefinierte Datentypen.

25.3.1 Eingabe benutzerdefinierter Typen

C++ ermöglicht die Eingabe benutzerdefinierter Datentypen, indem der Eingabeoperator `>>` überladen wird. Ebenso wie beim Überladen des Ausgabeoperators muss das erste Argument eine Referenz auf den Stream (hier also `istream`) sein, um eine Hintereinanderschaltung zu erlauben. Die Analogie zum auf Seite 322 behandelten Ausgabeoperator `<<` liegt auf der Hand. An dieser Stelle wird ein Eingabeoperator für die uns schon bekannte Klasse `Datum` angegeben. Es soll möglich sein, ein Datum im Format Tag◊Monat◊Jahr einzugeben, wobei das Trennzeichen ◊ entweder ein Punkt (.) oder ein Schrägstrich (/) sein darf:

```
// Anwendung des Eingabe-Operators für Datumswerte:
Datum dat1, dat2;
cout << "Eingabe zweier Daten: ";
cin >> dat1 >> dat2;    // Verkettung von >>
cout << dat1 << endl    // Lösung der Aufgaben von Seite 337 vorausgesetzt
    << dat2 << endl;
cout << "Eingabe von Daten bis zum Fehler\n";
while(cin) {
    try {
        cout << "Datum ?";
        cin >> dat1;
        cout << dat1 << endl;
    } catch(const char* e) {
        cout << e << " Abbruch!" << endl;
    }
}
```

Um das Programm zu realisieren, wird in der aus Kapitel 9 bekannten Datei `datum.h` der `>>`-Operator als globale Funktion deklariert. Die unten gezeigte zugehörige Implementa-

tion muss in der Datei *datum.cpp* nachgetragen werden. Ferner sollte der Typumwandlungsoperator durch eine Methode *toString()* ersetzt werden, damit der Compiler nicht versucht, *operator>>(istream&, string&)* zu benutzen.

```
// Auszug aus cppbuch/k25/datum/datum.h
std::istream& operator>>(std::istream&, Datum&) throw(const char*);

// Implementierung, Auszug aus cppbuch/k25/datum/datum.cpp
std::istream& operator>>(std::istream &eingabe, Datum &d) {
// Einlese-Operator für ein Datum
// erlaubte Formate: Tag.Monat.Jahr oder Tag/Monat/Jahr
    char c = '\0';
    int tag, monat, jahr;
    eingabe >> tag >> c;           // Tag und 1. Trennzeichen
    if(c != '.' && c != '/') {
        eingabe.setstate(std::ios::failbit); // Status setzen
    }
    else {
        eingabe >> monat >> c;    // Monat und 2. Trennzeichen
        if(c != '.' && c != '/') {
            eingabe.setstate(std::ios::failbit); // Status setzen
        }
        else {
            eingabe >> jahr;
        }
        if(jahr < 100) {
            jahr += 2000;
        }
        // Datum gültig?
        if(istGueltigesDatum(tag, monat, jahr)) { // nur dann
            d.set(tag, monat, jahr);
        }
        else {
            eingabe.setstate(std::ios::failbit);
        }
    }
    if(!eingabe.good()) {
        throw "kein gültiges Datum!";
    }
    return eingabe;
}
```

Die Variable *c* wird mit 0 initialisiert, damit sie nicht zufällig einen der erlaubten Werte annimmt, wenn die Eingabe von *tag* fehlschlagen sollte. Ein Eingabefehler beim Einlesen von *tag*, *monat* oder *jahr* bewegt das C++-Laufzeitsystem zum Setzen des *failbit*, weil es sich um *int*-Werte handelt. Ein anderer Eingabefehler führt im überladenen Operator ebenfalls zu einen *failbit*-Fehler, sodass die obige Abfrage *while(cin)* bei Fehlern zum Schleifenabbruch führt.

25.4 Array als Block lesen oder schreiben

Das Beispiel auf Seite 221 zeigt bereits das Schreiben eines Arrays als Block. Es fehlt jedoch das Lesen, und die Fehlerbehandlung besteht nur im Programmabbruch. Im Folgenden werden Lesen und Schreiben zur einfacheren Verwendung in jeweils eine einfach aufzurufende Funktion gepackt. Der Template-Parameter `T` steht für den Typ eines Array-Elements. Im Fehlerfall wird eine Exception geworfen:

Listing 25.11: Funktionen zur blockweisen binären Ein- und Ausgabe

```
// cppbuch/k25/binaer/binaerIO.t
#ifndef BINAERIO_T
#define BINAERIO_T
#include<ios>
#include<fstream>
#include<IOException.h> // siehe Text

template<typename T>
void schreibeBinaer(const std::string& dateiname, const T* daten,
                   size_t anzBytes) {
    std::ofstream ziel;
    ziel.open(dateiname.c_str(), std::ios::binary|std::ios::out);
    if (!ziel) {
        throw IOException(dateiname);
    }
    ziel.write(reinterpret_cast<const char*>(daten), anzBytes);
    ziel.close();
}

template<typename T>
void liesBinaer(const std::string& dateiname, T* daten, size_t anzBytes) {
    std::ifstream quelle;
    quelle.open(dateiname.c_str(), std::ios::binary|std::ios::in);
    if (!quelle) {
        throw IOException(dateiname);
    }
    quelle.read(reinterpret_cast<char*>(daten), anzBytes);
    quelle.close();
}
#endif
```



Hinweis

Die beiden Funktionen sind nur für Arrays geeignet, deren Elemente keine Zeiger enthalten. Von den Zeigern referenzierte Objekte würden nicht mitkopiert werden.

Die Klasse `IOException` sorgt dafür, dass der Dateiname im Fehlerfall an den Aufrufer übermittelt wird. Wie das im Einzelnen vonstatten geht, lesen Sie in der Beschreibung der Klasse `IOException` auf Seite 574 sowie in der nachfolgenden Beispielanwendung.

Listing 25.12: Anwendung: zweidimensionale Matrix schreiben und lesen

```
// cppbuch/k25/binaer/binaerIO.cpp
#include<cstdlib>
#include<iostream>
#include<algorithm>    // equal()
#include"binaerIO.t"
using namespace std;

int main() {
    const string dateiname("binaerdaten.bin");
    const int ZEILEN = 10;
    const int SPALTEN = 8;
    double matrix[ZEILEN][SPALTEN];
    // Matrix mit (hier beliebigen) Werten füllen und anzeigen:
    cout << "Matrix:\n";
    for(int i = 0; i < ZEILEN; ++i) {
        for(int j = 0; j < SPALTEN; ++j) {
            matrix[i][j] = i*SPALTEN + j;
            cout << matrix[i][j] << '\t';
        }
        cout << endl;
    }
    cout << endl;
    // Matrix schreiben
    size_t anzahlBytes = ZEILEN*SPALTEN*sizeof(matrix[0][0]);
    try {
        schreibeBinaer(dateiname, matrix, anzahlBytes);
    }
    catch(const IOException& e) {
        cout << e.what() << endl;
        return 1;
    }
    // Kopie anlegen und mit 0 vorbesetzen
    double kopie[ZEILEN][SPALTEN] = {{0}, {0}};
    // Kopie von der Datei einlesen
    try {
        liesBinaer(dateiname, kopie, anzahlBytes);
    }
    catch(const IOException& e) {
        cout << e.what() << endl;
        return 2;
    }
    // Werte vergleichen zum Nachweis des korrekten Schreibens/Lesens
    if(equal(matrix[0], matrix[0] + ZEILEN*SPALTEN, kopie[0]))
        cout << "Geschriebene und gelesene Daten sind gleich." << endl;
    else
        cout << "Geschriebene und gelesene Daten sind ungleich!" << endl;
}
```



Teil V:
Die C++-Standardbibliothek

26

Aufbau und Übersicht

Dieses Kapitel behandelt die folgenden Themen:

- Aufbau der Standardbibliothek
- Auslassungen/Anmerkungen zur Darstellung
- Bezug zu den Beispielen des Buchs

Dieser Teil des Buchs ist als *Nachschlagewerk*, nicht als Einführung für die wichtigsten Elemente der C++-Standardbibliothek gedacht, sofern sie in den vorangegangenen Kapiteln noch nicht ausreichend beschrieben werden. Ein gutes C++-Verständnis, möglicherweise erworben durch den ersten Teil, ist bei der Lektüre hilfreich. Allein der sehr dichte Text des Standarddokuments umfasst mehr als 1300 Seiten. Sämtliche technischen Details hier aufzuführen, würde ebenso wie der Versuch, alle Bestandteile der Bibliothek zu erläutern, den Rahmen dieses Buchs sprengen. Der Schwerpunkt liegt daher auf den Informationen, die für den Anwender wesentlich sind. Auf Dinge, an denen eher die Hersteller von Compilern interessiert sind, wird verzichtet. Den an weitergehenden Einzelheiten interessierten Leserinnen und Lesern sei als Ergänzung [\[BeckP\]](#), [\[KL\]](#) und [\[Str\]](#) empfohlen. Die Standarddokumente [\[ISOC\]](#) und [\[ISOC++\]](#) selbst kann man natürlich auch zu Rate ziehen.

Tabelle 26.1: Aufbau der Standardbibliothek (ohne C-Header)

Schwerpunkt	Header	Hinweise/Behandlung auf Seite(n)
Hilfsfunktionen und -klassen	<utility>	750
	<functional>	753
Container	<bitset>	801
	<deque>	775
	<list>	772
	<map>	782
	<unordered_map>	793
	<queue>	777
	<set>	787
	<unordered_set>	798
	<stack>	776
	<vector>	770
Iteratoren	<iterator>	805
Algorithmen	<algorithm>	815
Ein-/Ausgabe	<fstream>	96, 220, 390
	<iomanip>	383
	<ios>	375
	<iostream>	93
	<istream>	380, 735
	<ostream>	377
	<sstream>	393
Nationale Besonderheiten	<locale>	821
Numerisches	<complex>	695
	<limits>	725
	<numerics>	Tabelle 30.4, Seite 820
	<valarray>	857
String	<string>	841
Laufzeittyperkennung	<typeinfo>	295
Fehlerbehandlung	<exception>	301, 308
	<stdexcept>	301, 308
Speicher	<memory>	855
	<new>	854

Zu jedem Compiler gibt es eine Bibliothek mit nützlichen Klassen und Routinen. Die C++-Standardbibliothek (englisch *standard library*) stellt ein erweiterbares Rahmenwerk mit folgenden Komponenten zur Verfügung: Diagnose, Strings, Container, Algorithmen, komplexe Zahlen, numerische Algorithmen, Anpassung an nationale Zeichensätze, Ein-/Ausgabe und anderes mehr.

Die C++-Standardbibliothek ist systematisch nach Schwerpunkten aufgebaut, denen verschiedene Header zum Einbinden zugeordnet sind. Die Tabellen 26.1 und 26.2 geben eine Übersicht über die wichtigsten Header und über die Seitenzahl, wo sie ggf. beschrieben werden. Dabei kann es sein, dass dort kaum Informationen stehen, nämlich dann,

wenn es sich um aus der Programmiersprache C kommende Header handelt, die durch C++ überflüssig geworden sind. Einzelheiten über Auslassungen folgen. Alle Datentypen, Klassen und Funktionen sind im Namespace `std`, sodass diese Tatsache des Weiteren nicht mehr erwähnt werden muss. In den Tabellen 26.1 und 26.2 sind Doppelnennungen vermieden worden, wie zum Beispiel `<cmath>` auch bei »Numerisches« aufzuführen. Aus historischen Gründen sind manche Dinge nicht dort zu finden, wo man sie erwartet.

Tabelle 26.2: Ausgewählte C-Header der Standardbibliothek

Schwerpunkt	Header	Seite
C-Header	<code><cassert></code>	874
	<code><cctype></code>	874
	<code><cerrno></code>	875
	<code><cmath></code>	875
	<code><cstdarg></code>	876
	<code><cstddef></code>	877
	<code><cstdlib></code>	877
	<code><cstring></code>	879
	<code><ctime></code>	881

Zum Beispiel ist die Funktion `fabs()` (Absolutbetrag von `float`-Zahlen) sinnvollerweise im Header `<cmath>` enthalten, die Funktion `abs()` (Absolutbetrag von `int`-Zahlen) jedoch im Header `<cstdlib>`. Um Wiederholungen zu vermeiden, sind allgemeine Informationen, die für viele Abschnitte gelten, nur einmal am Anfang aufgeführt. Wenn zum Beispiel etwas über Vektoren nachgeschlagen werden soll (Abschnitt 28.2.1), ist es daher empfehlenswert, auch am Anfang des Kapitels über Container (Seite 763) nachzusehen.

26.1 Auslassungen

Einige C-Header

Einige von der Programmiersprache C herrührende Header sind in diesem Buch zwar erwähnt, aber nicht beschrieben, weil sie durch C++ überflüssig geworden sind. Dazu gehört zum Beispiel der Header `<stdio>` für die Ein- und Ausgabe, dessen Funktionen wegen der `iostream`-Bibliothek nicht mehr notwendig sind. Ferner werden die C-Header für Wide-Character- und Multibyte-Zeichen weggelassen, zumal die Unterstützung durch die `String`-Klasse gewährleistet ist.

Vereinfachte Template-Schreibweise

Der einfacheren Lesbarkeit wegen werden manche Templates auf Zeichen des Typs `char` bezogen, sodass zum Beispiel die beiden Deklarationen (aus der Klasse `bitset`)

```
template<typename charT, class traits, class Allocator>
basic_string<charT, traits, Allocator> to_string() const;
```

```
template<typename charT, class traits, size_t N>
basic_ostream<charT, traits>&
operator<<((basic_ostream<charT, traits>& os, const bitset<N>& x);
```

vereinfacht werden zu

```
string to_string() const;

template <size_t N>
ostream& operator<<((ostream& os, const bitset<N>& x);
```

Die Standardbibliothek stellt einen Standard-Allokator (Klasse `allocator`) zur Speicherbeschaffung zur Verfügung. Die Speicherverwaltung mit eigenen Allokatoren ist möglich, aber relativ speziell und nur in seltenen Fällen notwendig, sodass hier stets vom Standard-Allokator ausgegangen wird. Daher werden im Folgenden nur die vereinfachten Deklarationen verwendet, wie gezeigt:

```
// vollständige Deklaration
template<typename T, class Allocator = allocator<T> > class vector;
// vereinfachte Deklaration
template<typename T> class vector;
```

Es ergeben sich keinerlei Probleme dadurch, weil der Vorgabeparameter für den Allokator stets zuletzt kommt und deswegen bei Bedarf eingefügt werden kann.

Container-Methoden für R-Wert-Referenzen

Manche Container-Methoden haben ein Gegenstück für R-Wert-Referenzen. Zum Beispiel fügt die Methode `push_front(const T& x)` der Klasse `list` das Element `x` des Typs `T` am Anfang ein, indem eine Kopie von `x` in der Liste angelegt wird. Falls ein R-Wert übergeben wird, ist eine Kopie nur Zeitverschwendung; er kann dann ohne Kopie direkt in die Liste eingetragen werden. Dafür gibt es die überladene Methode `push_front(T&& x)`. Die Bedeutung, nämlich ein Objekt am Anfang der Liste einzutragen, ist dieselbe. Aus diesem Grund werden überladene Methoden mit `T&&`-Parametertypen, obwohl vorhanden, zur Abkürzung bei den Containern in Kapitel 28 nicht mit aufgeführt.

Type Traits

In [ISOC++] gibt es über 50 Template-Klassen mit Typnamen, die zur Compilierzeit ausgewertet werden können. Diese Template-Klassen werden »traits« genannt. Auf einige davon und auf die Wirkungsweise wird in Kapitel 29 ab Seite 805 eingegangen, weswegen auf die Auflistung aller übrigen verzichtet wird. Abschnitt 29.1.1 zeigt ein Beispiel. [BeckP] und [ISOC++] bieten weitere Informationen.

forward_list

`forward_list` ist eine einfach verkettete Liste. Weil die doppelt verkettete Liste `list` ebensogut oder besser einsetzbar ist, wird auf eine nähere Beschreibung verzichtet.

26.2 Beispiele des Buchs und die C++-Standardbibliothek

Der Sinn dieses Buchs ist es, nicht nur als Nachschlagewerk zu dienen, sondern auch eine Einführung in die Sprache C++ zu geben. Dazu gehört die Vermittlung des Verständnisses, wie Templates, Strings, Container, smarte Pointer und anderes funktionieren und zusammenwirken. Wenn die Beispiele verstanden sind, ist es jedoch sinnvoll, sie nicht weiter zu entwickeln, sondern die vorgefertigten Komponenten der Standardbibliothek zu benutzen, weil Letztere mehr bieten und zudem durch die Compilerhersteller gepflegt werden.

Ein ebenso wichtiger Aspekt ist die Wartbarkeit von Programmen: Standardkonforme Programme können durch andere mit weniger Aufwand gepflegt werden, weil sich niemand in Spezialbibliotheken einarbeiten muss, wobei die Kenntnis des Standards natürlich vorausgesetzt wird. Aus diesem Grund sind in der Tabelle 26.3 die Beispielklassen dieses Buchs den in etwa vergleichbaren Komponenten der C++-Standardbibliothek gegenübergestellt.

Tabelle 26.3: Beispielklassen und die Standardbibliothek

Beispielklasse dieses Buchs	Abschnitt, Seite	etwa vergleichbare Standardkomponente	Standard-Header	siehe Seite
MeinString	6.1, 233	string	<string>	841
Liste	11.3, 404	list	<list>	772
Warteschlange	7.12, 297	queue	<queue>	777
Stack	6.3.2, 249	stack	<stack>	776
SmartPointer	9.5, 339	shared_ptr	<memory>	855
Vektor	9.2, 323	vector	<vector>	770

Die Vergleichbarkeit ist nicht nur beschränkt durch die umfangreichere und teilweise etwas andere Funktionalität der Standardbibliothek. Insbesondere wird in den Klassen der Standardbibliothek aus Geschwindigkeitsgründen häufig auf Prüfungen zur Laufzeit verzichtet. Zum Beispiel gibt es keine Exception bei der Klasse `shared_ptr`, wenn ein nicht-initialisierter Zeiger dereferenziert werden soll. Es gibt auch keine Laufzeitfehlermeldung, wenn ein außerhalb des zulässigen Bereichs liegender Index beim Zugriff auf ein `vector`-Objekt verwendet wird. Es steht dem Benutzer der Bibliothek jedoch frei, `vector::at(size_t)` aufzurufen oder Erweiterungen vorzunehmen. So kann eine Vektor-Klasse entworfen werden, deren Indexoperator eine Prüfung vornimmt, wie in Abschnitt 9.2 gezeigt.



Tipp

Eine gute Referenz zur C++-Standardbibliothek finden Sie unter <http://stdcxx.apache.org/doc/stdlibref/>.

27

Hilfsfunktionen und -klassen

Dieses Kapitel behandelt die folgenden Themen:

- Relationale Operatoren
- Unterstützung der Referenzsemantik für R-Werte
- Template pair für Wertepaare
- Tupel
- Funktionsobjekte
- Binden von Argumentwerten
- Standard-Templates für rationale Zahlen, Zeit und Dauer
- Standard-Template für Wrapper- oder Hüll-Klassen

27.1 Relationale Operatoren

Die folgenden relationalen Operatoren werden im Header `<utility>` im Namespace `rel_ops`, der innerhalb des Namespaces `std` liegt, definiert.

```
namespace rel_ops {                                // Rückgabe:
template<class T>
    bool operator!=(const T& x, const T& y); // !(x == y)
template<class T>
    bool operator>(const T& x, const T& y); // y < x
template<class T>
    bool operator<=(const T& x, const T& y); // !(y < x)
template<class T>
    bool operator>=(const T& x, const T& y); // !(x < y)
}
```


Wie die rechte Seite zeigt, reicht es für die Anwendung der Operatoren aus, wenn die beteiligten Typen selbst den `==`-Operator im ersten Fall (`!=`) und den `<`-Operator in den anderen Fällen zur Verfügung stellen. Wenn der Namespace `rel_ops` eingebunden wird, werden die anderen Operatoren daraus abgeleitet.

27.2 Unterstützung der Referenzsemantik für R-Werte

In diesem Abschnitt geht es um die Unterstützung der Referenzsemantik für R-Werte durch die Standardfunktionen `move()` und `forward()`. Die Grundlagen dazu finden Sie in Abschnitt 22.2 ab Seite 591.

`move()`

`move()` (Header `<utility>`) ist eine Funktion, die eine Referenz auf einen R-Wert zurückgibt (`&&`). Damit erleichtert sie manche Optimierungen, wie das folgende Beispiel zeigt. Zur Klasse `StringType` siehe Abschnitt 22.3 (Seite 596). Für interne Zwecke hat diese Klasse eine private Funktion `swap()` zur Vertauschung zweier Strings. Nun soll die Möglichkeit gegeben werden, auch außerhalb der Klasse eine Vertauschung zweier `StringType`-Objekte vorzunehmen. Weil es bereits eine Standardfunktion `swap()` gibt und es hier weniger um das Vertauschen, sondern um die Anwendung und Erklärung zu `move()` geht, heißt diese Funktion `void tauschen(StringType& a, StringType& b)`. Eine Implementierung könnte wie folgt aussehen:

```
void tauschen0(StringType& a, StringType& b) { // Version 1
    StringType tmp(a);           // Kopierkonstruktor
    a = b;                       // operator=()
    b = tmp;                     // operator=()
}
```

Wie im Kommentar angegeben, werden einmal der Kopierkonstruktor und zweimal der Zuweisungsoperator aufgerufen. Jede dieser Operationen verursacht `new` und `delete` – aufwendige Operationen im Vergleich zum Tauschen von Adressen. Tatsache ist, dass der Inhalt des Objekts `a` nach Kopie in der ersten Zeile gar nicht mehr gebraucht wird. Dasselbe gilt für `b` in der zweiten Zeile. Hier kommt nun `move()` ins Spiel!

```
void tauschen(StringType& a, StringType& b) { // Version 2
    if(&a != &b) {
        StringType tmp(move(a)); // moving Konstruktor
        a = move(b);             // moving operator=()
        b = move(tmp);           // moving operator=()
    }
}
```

Der Vergleich `if (&a != &b)` muss hier sein, weil `move()` die Ressourcen »stiehlt«. Wenn `&a == &b` wäre, würde sich ohne die Abfrage `a` mit der Anweisung `a = move(b)`; selbst die Ressourcen entziehen.

Weil `move()` eine Referenz auf einen R-Wert zurückgibt, können die erheblich effizienteren Varianten `StringType(StringType&&)` und `operator=(StringType&&)` aufgerufen werden. Anstelle der `move()`-Funktion hätte man alternativ eine statische Typumwandlung einsetzen können (`static_cast<StringType&&>(a)`). Die `move()`-Funktion hat den Vorteil, dass sie auch von Compilern, die Referenzen auf R-Werte noch nicht unterstützen, akzeptiert wird. `move()` tut dann einfach nichts, außer direkt das Argument zurückzugeben. Anstelle des `moving`-Konstruktors würde der normale Kopierkonstruktor treten. Mit `move()` geschriebener Code muss nach einem Update des Compilers, der dann möglicherweise eine geänderte Implementation von `move()` enthält, nicht geändert werden. Mit einer weiteren Modifikation könnten auch temporäre Objekte getauscht werden:

```
StringTyp a("einA");  
tauschen(a, StringTyp("temporaer")); // oder  
tauschen(StringTyp("temporaer"), a);
```

Dazu muss der R-Wert per `&&` übergeben werden, das heißt, es muss noch die folgenden überladenen Varianten geben:

Listing 27.1: move()

```
// vollständiges Beispiel siehe cppbuch/k27/move/move.cpp
void tauschen(StringTyp& a, StringTyp&& b) {
    StringTyp tmp(move(a)); // moving (bewegender) Konstruktor
    a = move(b);            // moving operator=()
    b = move(tmp);          // moving operator=()
}

void tauschen(StringTyp&& a, StringTyp& b) {
    StringTyp tmp(move(a)); // moving Konstruktor
    a = move(b);            // moving operator=()
    b = move(tmp);          // moving operator=()
}
```

move() für Container-Bereiche

Um bequem ganze Bereiche mit `move()` bewegen zu können, gibt es die Funktion

```
template<class InputIterator, class OutputIterator>
OutputIterator move(InputIterator first,
                   InputIterator last,
                   OutputIterator result);
```

Der Header ist `<algorithm>`. Die Funktion bewegt die Elemente aus dem Bereich `[first, last)` in den Bereich `result, result + (last-first)`, indem für jedes `n < (last-first)` die Anweisung `*(result+n) = std::move(*(first+n));` ausgeführt wird. Die Funktion

[illegible]

bewegt die Elemente aus dem Bereich `[first, last)` in den Bereich `result - (last-first), result)`, wobei am Ende `(last-1)` angefangen wird. `result` darf nicht im Bereich `[first, last)` liegen, damit nicht noch zu bewegendende Daten zerstört werden.

forward()

`forward()` ist eine Funktion, die das sogenannte »perfect forwarding« unterstützt. »Perfect forwarding« ist dann gefragt, wenn man eine Template-Funktion mit Referenz-Parametern schreiben will, und wenn beabsichtigt ist, diese Parameter an eine andere Funktion weiterzuleiten. Das Problem dabei ist, dass zum Beispiel ein temporäres Objekt zwar zum Parametertyp `const&` passt, die Eigenschaft »temporär« aber nicht an die andere Funktion weitergeleitet werden kann. `forward()` hilft dabei und ermöglicht es, viele überladene Funktionen durch ein Template zu ersetzen. Weil `forward()` hauptsächlich in Bibliotheksfunktionen zum Tragen kommt, wird es hier nicht weiter diskutiert. Sie finden ein Beispiel im Verzeichnis `cppbuch/k27/forward`, und eine gute Darstellung mit weiteren Beispielen unter <http://msdn.microsoft.com/en-us/library/dd293668.aspx>.

27.3 Paare

Das Template `pair` erlaubt die Kombination heterogener Wertepaare:

Listing 27.2: Template `pair` (vereinfachter Auszug)

```
template<class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair(const T1& x, const T2& y);
    template<class U, class V> pair(const pair< U, V>& p);
    template<class U, class V> pair(pair< U, V>&& p);
    template<class U, class V> pair& operator=(pair< U, V>&& p);
    void swap(pair& p);
    // ...
};
```

Der (nicht aufgeführte) Standardkonstruktor initialisiert die Bestandteile mit ihrem jeweiligen Standardkonstruktor. Gleichheit und der Vergleich sind ebenfalls definiert:

```
// gibt x.first == y.first && x.second == y.second zurück
template<class T1, class T2>
bool operator==(const pair<T1, T2>& x,
                const pair<T1, T2>& y);

template<class T1, class T2>
bool operator<(const pair<T1, T2>& x,
              const pair<T1, T2>& y);
```

Die anderen relationalen Operatoren werden hier aus Platzgründen nicht aufgeführt; sie werden entsprechend Abschnitt 27.1 abgeleitet. Was bedeutet nun der Rückgabewert für `operator<>()`? Antwort:

```
true  : falls x.first < y.first
false : falls y.first < x.first
```

Falls beide Bedingungen nicht zutreffen, wird `x.second < y.second` zurückgegeben, mit anderen Worten: Es liegt ein lexikografischer Vergleich vor. Die Hilfsfunktion

```
template<class T1, class T2>
pair<T1, T2> make_pair(const T1& x, const T2& y);
```

erzeugt ein Paar aus den Parametern. Die beteiligten Typen werden aus den Parametern abgeleitet. Zum Beispiel gibt der Aufruf

```
return make_pair(1, 2.782);
```

ein `pair<int, double>`-Objekt zurück. Alternativ kann der gewünschte Typ angegeben werden:

```
return make_pair<int, double>(1, 2.782);
```

Das Beispiel zeigt `make_pair()` und die Anwendung des `<`-Operators.

Listing 27.3: Vergleich zweier `pair`-Objekte

```
// cppbuch/k27/pair/pair.cpp
#include <utility>
#include <iostream>
using namespace std;

int main() {
    pair<string, string> p1 = make_pair("Donald", "Knuth");
    pair<string, string> p2 = make_pair("Donald", "Duck");
    cout << "Vergleich zweier Paare" << endl;
    if(p1 > p2) { // Anwendung von Operator >
        cout << p2.first << " " << p2.second
              << " liegt alphabetisch vor "
              << p1.first << " " << p1.second << endl;
    }
    else {
        cout << p1.first << " " << p1.second
              << " liegt alphabetisch vor "
              << p2.first << " " << p2.second << endl;
    }
}
```

Um `pair` kompatibel zu `tuple` zu machen, stellt der Header `<utility>` die Funktionen `get<0>(const pair<T1, T2>& p)` und `get<1>(const pair<T1, T2>& p)` für den Zugriff auf die `pair`-Elemente `first` bzw. `second` zur Verfügung. Einzelheiten finden Sie im folgenden Abschnitt.

27.4 Tupel

Das Template `tuple` ist eine Verallgemeinerung des obigen `pair`-Templates auf weniger oder mehr als zwei heterogene Elemente, wobei mindestens 10 möglich sein sollen. Diese Kombination heißt *Tupel* und wird durch das Template `tuple` realisiert. Es ist nicht im Header `<utility>` angesiedelt, sondern in `<tuple>`.

```
tuple<> nichts;           // enthält kein Element
tuple<float> eins;        // enthält ein float-Element
tuple<int, float> zwei;    // enthält ein int- und ein float-Element
tuple<int, float, string> drei; // enthält zusätzlich ein string-Element
// usw.
```

Bei mehreren Elementen wäre der Zugriff über `first`, `second` usw. wie bei `pair` zu umständlich. Deswegen gibt es spezielle Template-Funktionen. Der Aufruf `get<n>(einTupel)` gibt das `n`-te Element des Tupels `einTupel` zurück. Das folgende Beispiel zeigt Tupel mit zwei und drei Elementen, `make_tuple()` und die Extraktion der Elemente mit `get<n>()`.

Listing 27.4: `Tupel`

```
// cppbuch/k27/tuple/tuple.cpp
#include<iostream>
#include<string>
#include<tuple>
using namespace std;

int main() {
    tuple<string, string> t2("Donald", "Knuth");
    tuple<string, string, int> t3("Donald", "Duck", 17);
    // get<>() gibt Referenzen zurück
    cout << "Zugriff mit get<>: "
         << get<0>(t2) << " " << get<1>(t2) // Donald Knuth
         << endl;
    get<0>(t3) = "Dagobert";                // Änderung
    cout << get<0>(t3) << " " << get<1>(t3) // Dagobert Duck
         << endl;
    // Abfrage der Anzahl der Elemente mit tuple_size<T>::value
    typedef tuple<string, string> tupeltyp;
    cout << "tupeltyp hat "
         << tuple_size<tupeltyp>::value // 2
         << " Elemente" << endl;
}
```

Eine umfangreichere Anwendung von `tuple` finden Sie im Abschnitt [24.10.1](#). Dort werden Tupel zum »Aufsammeln« von Operanden benutzt, um Operatoren möglichst effizient zu überladen.

27.5 Funktionsobjekte

Im Header `<functional>` sind Klassen definiert, die dem Erzeugen verschiedener Funktionsobjekte dienen. Funktionsobjekte haben alle einen `operator()()` und werden ausführlich ab Seite 344 beschrieben.

Im nächsten Abschnitt sind einige Klassen für Funktionsobjekte aufgeführt. Dieser Programmausschnitt zeigt eine einfache Anwendung:

```
// Auszug aus cppbuch/k27/funktionsobjekt/addierer.cpp
std::plus<double> addierer;
double d1 = 3.1415926;
double d2 = 2.718;
std::cout << d1 << " + " << d2 << " = "
          << addierer(d1, d2) << std::endl;
```

27.5.1 Arithmetische, vergleichende und logische Operationen

In der Rückgabespalte der folgenden Klassen für Funktionsobjekte meint `x` das erste und `y` das zweite Argument.

```
// arithmetische Operationen:
template<class T> struct plus;      // x + y
template<class T> struct minus;    // x - y
template<class T> struct multiplies; // x * y
template<class T> struct divides;  // x / y
template<class T> struct modulus;  // x % y
template<class T> struct negate;   // -x
```

```
// Vergleiche:
template<class T> struct equal_to;  // x == y
template<class T> struct not_equal_to; // x != y
template<class T> struct greater;   // x > y
template<class T> struct less;      // x < y
template<class T> struct greater_equal; // x >= y
template<class T> struct less_equal;  // x <= y
```

```
// logische Operationen:
template<class T> struct logical_and; // x && y
template<class T> struct logical_or;  // x || y
template<class T> struct logical_not; // !x
```

27.5.2 Funktionsobjekte zum Negieren logischer Prädikate

`not1()` und `not2()` sind Funktionen, die ein Funktionsobjekt zurückgeben, dessen Aufruf ein Prädikat negiert. Das Prädikat kann einen (`not1()`) oder zwei (`not2()`) Parameter haben. Die zurückgegebenen Funktionsobjekte sind vom Typ `unary_negate` bzw. `binary_negate`:

```
template<class Predicate>
class unary_negate {
```

```

public:
    explicit unary_negate(const Predicate& pred)
        : P(pred) {
    }

    bool operator()(const typename Predicate::argument_type& x) const {
        return !P(x);
    }

    typedef typename Predicate::argument_type argument_type;
    typedef bool result_type;
protected:
    Predicate P;
};

```

```

template<class Predicate>
class binary_negate {
public:
    explicit binary_negate(const Predicate& pred)
        : P(pred) {
    }

    bool operator()(
        const typename Predicate::first_argument_type& x,
        const typename Predicate::second_argument_type& y) const {
        return !P(x, y);
    }

    typedef typename Predicate::first_argument_type first_argument_type;
    typedef typename Predicate::second_argument_type second_argument_type;
    typedef bool result_type;
protected:
    Predicate P;
};

```

Wenn *P* ein Prädikat des Typs *Predicate* ist, gibt der Aufruf `not1(P)` das Objekt `unary_negate<Predicate>(P)` zurück. `not2(P)` liefert dementsprechend das Objekt `binary_negate<Predicate>(P)`. Eine mögliche Anwendung sei hier am Beispiel einer Sortierfunktion demonstriert, die ein Funktionsobjekt zum Vergleich der `int`-Elemente der Tabelle benötigt:

```

sortieren(Tabelle, Anzahl, less<int>());
// umgekehrte Reihenfolge:
sortieren(Tabelle, Anzahl, not2(less<int>()));

```

27.5.3 Binden von Argumentwerten

Die Template-Funktion `bind()` bindet Parameter oder Werte an eine Funktion. Das von `bind()` zurückgegebene Objekt ist ein Funktionsobjekt und kann wie eine Funktion aufgerufen werden (überladener `operator()()`). Das folgende Beispiel zeigt die anschließend genauer diskutierte Wirkungsweise.

Listing 27.5: Binden von Argumentwerten mit `bind()`

```
// cppbuch/k27/bind/bind1.cpp
#include <functional>
#include <iostream>
using std::bind;
using namespace std::placeholders;

template<class T>          // T muss ein Typ für Funktionsobjekte sein
void aufrufen(T funcObj) {
    int i1 = 1;
    int i2 = 3;
    int i3 = 5;
    int doppel = funcObj(i1, i2, i3); // Funktionsaufruf über Objekt
    std::cout << doppel << std::endl;
}

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

int main() {
    aufrufen(bind(verdoppeln, 9)); // Funktion an Wert binden
    aufrufen(bind(verdoppeln, _1)); // Funktion an Parameter 1 binden
    aufrufen(bind(verdoppeln, _2)); // Funktion an Parameter 2 binden
    aufrufen(bind(verdoppeln, _3)); // Funktion an Parameter 3 binden
}
```

1. In der `main()`-Funktion wird zuerst der Zahlenwert 9 an die Funktion `verdoppeln()` gebunden. Der Aufruf des zurückgegebenen Funktionsobjekts geschieht innerhalb der Funktion `aufrufen()`, wobei die Parameter `i1`, `i2` und `i3` alle ignoriert werden, weil ja schon ein Wert an die Funktion gebunden ist. Es wird das Doppelte von 9, also 18, ausgegeben.
2. Im nächsten Schritt wird die Funktion `verdoppeln()` an den ersten Parameter gebunden. Das bedeutet: Ruft man das Funktionsobjekt mit mehreren Parametern auf, wird nur der erste ausgewählt. Der Wert des ersten Parameters (`i1`) ist 1, also wird 2 ausgegeben. Dabei ist `_1` eine vordefinierte Größe, die im Namespace `std::placeholders` definiert ist.
3. Entsprechendes gilt für die nächsten Anweisungen und `_2` bzw. `_3`.

Dem Funktionsobjekt können mehrere Parameter übergeben werden. Wie viele es maximal sein dürfen, ist implementationsabhängig, aber es sollen mindestens 10 sein. Die Funktion `verdoppeln()` benötigt nur einen Parameter, `bind()` ist aber für mehrere Parameter geeignet, was im Folgenden anhand einer Funktion, die zwei Parameter benötigt, gezeigt werden soll. Der bekannte Funktor `less` vergleicht zwei Werte miteinander und gibt `true` zurück, wenn der erste Wert kleiner ist. Wenn der zweite Wert festgelegt ist, zum Beispiel auf 103, genügt ein Funktionsobjekt, das mit `bind` erzeugt wird, um den ersten Wert, der kleiner ist (hier: 101), herauszufinden. Der verwendete Algorithmus `find_if()` wird weiter unten in Abschnitt 30 beschrieben.

Listing 27.6: Prädikat mit bind() verwenden

```
// cppbuch/k27/bind/bind2.cpp
#include <functional>
#include <vector>
#include <iostream>
#include <algorithm>

using std::bind;
using namespace std::placeholders;

int main() {
    std::vector<int> v;
    v.push_back(111);
    v.push_back(107);
    v.push_back(101);
    v.push_back(90);
    v.push_back(106);

    int wert = 103;
    std::cout << "Das erste Element < " << wert << " ist: " <<
        *find_if(v.begin(), v.end(),
            bind(std::less<int>(), _1, wert)) << std::endl;
}
```

27.5.4 Funktionen in Objekte umwandeln

Funktionsobjekte, auch Funktoren genannt, sind flexibler als Funktionszeiger. Von vielen Algorithmen der Standardbibliothek gibt es zwei Varianten: eine mit einem Standardverhalten, und eine, die als Parameter ein Funktionsobjekt nimmt, um das Verhalten des Algorithmus zu ändern. Die übergebene Funktion heißt Callback-Funktion, weil sie aus dem Algorithmus heraus aufgerufen wird.

Das Klassen-Template `function` erzeugt Funktionsobjekte aus Funktionen, Zeigern auf Funktionen und anderen Funktionsobjekten. Das folgende Programm zeigt, wie eine Tabelle aus Funktoren mit verschiedenen Funktionen aufgebaut wird, die als Callback-Funktion in der Funktion `berechnen()` dienen. `berechnen()` verknüpft die Elemente zweier Vektoren `a` und `b` mit der übergebenen Funktion und legt das Ergebnis in einem anderen Vektor ab. Die jeweilige Funktion wird in einer Schleife über den Tabellenindex ausgewählt.

Listing 27.7: Tabelle mit Callback-Funktionen

```
// cppbuch/k27/funktionsobjekt/function.cpp
#include <algorithm>
#include <cassert>
#include <functional> // function
#include <iostream>
#include <vector>
#include <showSequence.h> // cppbuch/include

void berechnung(const std::vector<double>& v1,
               const std::vector<double>& v2,
```

```

        std::vector<double>& ergebnis,
        const std::function<double (double, double)>& f) {
    assert((v1.size() == v2.size()) && (v1.size() == ergebnis.size()));
    for(size_t i = 0; i < v1.size(); ++i) {
        ergebnis[i] = f(v1[i], v2[i]);    // Funktionsaufruf
    }
}

double differenzquadrat(double arg1, double arg2) {
    return (arg1 - arg2)*(arg1 - arg2);
}

using namespace std;

int main() {
    // als Variable verwenden
    function<double (double, double)> malnehmen = multiplies<double>();
    cout << malnehmen(4, 42) << endl;

    vector<double> a = {1.00, 2.00, 3.00, 4.00, 5.00};
    vector<double> b = {1.01, 2.02, 3.03, 4.04, 5.05};
    vector<double> erg(b.size());

    // Tabelle mit Funktionen
    vector<function<double (double, double)>> funktionen;
    funktionen.push_back(plus<double>()); // std::-Funktork
    funktionen.push_back(differenzquadrat); // Funktion
    double (*fptr)(double, double) = differenzquadrat;
    funktionen.push_back(fptr);           // Funktionszeiger

    // Zugriff auf verschiedene Funktionen per Index i
    for(size_t i = 0; i < funktionen.size(); ++i) {
        berechnung(a, b, erg, funktionen[i]);
        showSequence(erg);
    }
}

```

Elementfunktionen in Objekte umwandeln

Gelegentlich möchte man für alle Objekte einer Klasse eine Elementfunktion ausführen. So wäre es denkbar, alle in einem Vektor abgelegten graphischen Objekte zeichnen zu lassen. Mit Bezug auf die Klasse `GraphObj` des Abschnitts 7.6.2 könnte die Anweisung wie folgt lauten:

```
for_each(objekte.begin(), objekte.end(), &GraphObj::zeichnen); // Fehler!
```

Hier besteht das Problem, dass der Standard-Algorithmus `for_each()` ein Funktionsobjekt oder eine Funktion erwartet – aber keine Methode einer Klasse. Für solche Zwecke gibt es die Funktion `mem_fn()`, die eine Elementfunktion in ein Funktionsobjekt umwandelt. Das Programm zeigt, dass damit auch die polymorphe Nutzung funktioniert:

Listing 27.8: Elementfunktion als Funktionsobjekt nutzen

```
// cppbuch/k27/funktionsobjekt/memfn.cpp
#include<algorithm>
#include<functional>
#include<vector>
#include"../k7/abstrakt/quadrat.h"

using namespace std;

int main() {
    Quadrat q(Ort(100, 20), 50);
    Rechteck r(Ort(30, 40), 20, 20);
    vector<GraphObj*> objekte = {&q, &r};
    // alle Objekte zeichnen (polymorph)
    for_each(objekte.begin(), objekte.end(), mem_fn(&GraphObj::zeichnen));
}
```

Man könnte hier auch `function` nehmen, es wäre aber etwas umständlicher:

```
function<void (GraphObj*)> f = &GraphObj::zeichnen;
for_each(objekte.begin(), objekte.end(), f);
```

27.6 Templates für rationale Zahlen

Der Header `<ratio>` definiert die Klasse `ratio` für rationale Zahlen, die zur Compilationszeit festliegen. Die Klasse ist wie folgt definiert:

```
template<intmax_t N, intmax_t D = 1> //
class ratio {
public:
    typedef ratio<num, den> type;
    static const intmax_t num; // numerator (Zähler)
    static const intmax_t den; // denominator (Nenner)
};
```

Zähler und Nenner sind Bestandteile des Typs. `intmax_t` ist ein im C-Standard [ISO C] festgelegter `int`-Datentyp für große Zahlen, der im Header `<stdint>` definiert ist. Auf meinem System ist `sizeof(intmax_t)` gleich 8 (Bytes). Für Zehnerpotenzen gibt es vordefinierte Namen entsprechend dem Internationalen Einheitensystem (SI). So sind zum Beispiel `micro` und `mega` definiert:

```
typedef ratio<1, 1000000> micro;
typedef ratio<1000000, 1> mega;
```

Es gibt ergänzend Templates zur Unterstützung von Rechenoperationen, die den Typ des Ergebnisses zurückgeben. Zum Beispiel ist der Typ `ratio_multiply<micro, mega>::type` identisch mit `ratio<1, 1>`.

Kürzen des Bruchs, also die Division durch den größten gemeinsamen Teiler, wird zur Compilationszeit vorgenommen (sonst wäre der Typ `ratio<1000000, 1000000>`). Falls Sie rätseln sollten, wie das funktioniert, bitte ich Sie, einen Blick in den Abschnitt 6.4 (Seite 251) zu werfen. Abschließend sei ein Programm gezeigt, das Zähler und Nenner einiger `ratio`-Typen ausgibt und auch die Arithmetik zur Compilationszeit zeigt. Die Funktion `printRatio()` ist eine Hilfsfunktion zur Anzeige mit einem Bruchstrich. Die Funktion benötigt keine Argumente; nur ihr Typ ist entscheidend. Weitere Einzelheiten finden Sie in [ISOC++].

Listing 27.9: `ratio`-Arithmetik

```
// cppbuch/k27/ratio/main.cpp
#include<ratio>
#include<iostream>

template<typename T>
void printRatio() {
    std::cout << T::num << '/' << T::den << std::endl;
}

using namespace std;

int main() {
    cout << mega::num << endl; // Zähler
    cout << mega::den << endl; // Nenner
    cout << micro::num << endl; // Zähler
    cout << micro::den << endl; // Nenner

    // Arithmetik zur Compilationszeit
    cout << ratio<7, -21>::num << endl; // -1
    cout << ratio<7, -21>::den << endl; // 3

    cout << ratio_add<mega, micro>::type::num << endl; // 10000000000001
    cout << ratio_add<mega, micro>::type::den << endl; // 1000000

    cout << ratio_multiply<ratio<7,-21>, ratio<9,-33>>::type::num << endl;
    cout << ratio_multiply<ratio<7,-21>, ratio<9,-33>>::type::den << endl;

    cout << "micro = ";
    printRatio<micro>();

    cout << "7/-21 = ";
    printRatio<ratio<7,-21>>();

    cout << "(7/-21)*(9/-33) = ";
    printRatio<ratio_multiply<ratio<7,-21>, ratio<9,-33>>::type>();

    cout << "kilo (=1000)*milli(=1/1000) = ";
    printRatio<ratio_multiply<kilo, milli>::type>();
}
```

27.7 Zeit und Dauer

In Ergänzung zu den in Abschnitt 35.10 behandelten C-Funktionen zur Bearbeitung und Auswertung der Systemzeitinformation stellt der Header `<chrono>` Klassen im Namespace `std::chrono` zur Verarbeitung von Zeitinformationen und Berechnung von Zeitdauern zur Verfügung. Das Beispiel zeigt, wie mit Zeiten gerechnet werden kann und wie Zeiten gemessen werden können. `sleep_for(dauer)` lässt das Programm die angegebene Dauer untätig sein, während `sleep_until(zeitpunkt)` das Programm bis zu einem bestimmten Zeitpunkt schlafen legt. Weitere Einzelheiten finden Sie in [ISOC++].

Listing 27.10: Zeitfunktionen

```
// cppbuch/k27/chrono/main.cpp
#include<chrono>
#include<iostream>
#include <thread>

using namespace std;
using namespace std::chrono;

int main() {
    seconds fastEinTag = hours(23) + minutes(59) + seconds(59);
    cout << "23h 59m 59s sind "
         << fastEinTag.count() << " Sekunden" << endl;

    // Zeitmessung
    auto anfang = system_clock::now();
    // Rechnung, um Zeit vergehen zu lassen ....
    cout << "Rechnung läuft ..." << endl;
    double summe = 0.0;
    for(long i = 0; i < 2000000000L; ++i) {
        summe += i/10000.0;
    }
    // Rechnung beendet, Dauer ausgeben:
    nanoseconds ns = system_clock::now() - anfang;
    cout << "Dauer [ns] = " << ns.count() << endl;
    cout << "Dauer [us] = "
         << duration_cast<microseconds>(ns).count() << endl;
    cout << "Dauer [ms] = "
         << duration_cast<milliseconds>(ns).count() << endl;
    cout << "Dauer [s] = "
         << duration_cast<seconds>(ns).count() << endl;

    cout << "1500 ms warten ..." << endl;
    this_thread::sleep_for(milliseconds(1500));

    cout << "warten bis (jetzt + 5 Sekunden) ..." << endl;
    auto dann = system_clock::now() + seconds(5);
    this_thread::sleep_until(dann);
}
```

27.8 Hüllklasse für Referenzen

Es kann vorkommen, dass ein Objekt an eine Funktion, die eine Übergabe per Wert erwartet, übergeben werden soll, obwohl das Erzeugen einer Kopie des Objekts nicht sinnvoll oder nicht möglich ist. Der erste Fall tritt auf, wenn der Aufrufer der Funktion das übergebene Objekt direkt beeinflussen will – die Kopie in der Funktion ist aber unerreichbar. Ein Beispiel dafür finden Sie in Abschnitt 13.3 auf Seite 429, wo es darum geht, die Kopie eines dem Konstruktor der Klasse `thread` übergebenen Objekts zu verhindern.

Um ein nicht zu kopierendes Objekt dennoch an eine Funktion, die eine Übergabe per Wert erwartet, übergeben zu können, bietet C++ die Hüllklasse (englisch *wrapper*) `reference_wrapper` an; der Header ist `<functional>`. Im Folgenden nehme ich zur Vereinfachung an, dass das folgende Template mit einer Klasse instanziiert wird, für deren Objekte `operator()` (`ArgTypes&&...`) aufgerufen werden kann. Damit ergibt sich die im Vergleich zu [ISOC++] etwas vereinfachte Schnittstelle für `reference_wrapper`:

```
template <class T>
class reference_wrapper {
public:
    typedef T type;
    reference_wrapper(T&);
    reference_wrapper(const reference_wrapper<T>& x);
    reference_wrapper& operator=(const reference_wrapper<T>& x);
    operator T& () const;
    T& get() const;           // gibt gespeicherte Referenz zurück
    // Aufruf
    template <class... ArgTypes>
    typename result_of<T(ArgTypes...)>::type
    operator() (ArgTypes&&...) const;
};
```

Der Rückgabetyt ergibt sich aus dem Aufruf `get().operator() (ArgTypes&&...)`. `result_of` ist ein Hilfstemplate, das den Rückgabetyt ermittelt, und dessen Interna nicht bekannt sein müssen. Die Klasse `call_wrapper` im Listing auf Seite 430 ist ein einfaches Äquivalent zu `reference_wrapper`.



Tip

Es ist einfacher, die Klasse `reference_wrapper` nicht direkt zu benutzen, sondern die Funktion `ref()`, die ein `reference_wrapper`-Objekt zurückgibt.

Beispiel mit `reference_wrapper`:

```
Worker worker;
std::reference_wrapper<Worker> aufrufer(worker);
thread t(aufrufer); // Thread anlegen und starten
```

Beispiel mit `ref()`:

```
Worker worker;
thread t(std::ref(worker)); // Thread anlegen und starten
```


28

Container

Dieses Kapitel behandelt die folgenden Themen:

- Eigenschaften der C++-Container
- `vector`, `list`, `deque`, `stack` und `queue`
- Sortierte assoziative Container
- Hash-Container

Ein Container ist ein Objekt, das der Verwaltung anderer Objekte dient, die hier Elemente des Containers genannt werden. Die mit Containern arbeitenden Algorithmen verlassen sich auf eine definierte Schnittstelle von Datentypen und Methoden. Tabelle 28.1 zeigt eine Übersicht der verschiedenen Container der C++-Standardbibliothek.

Sequenzen

bedeutet, dass die Elemente im Container eine lineare Ordnung haben. `list`, `stack` und `vector` sind aus vorhergehenden Kapiteln bekannt. `array` ist eine Hüllklasse (englisch *wrapper*) für C-Arrays. `deque` ist eine Abkürzung für »double ended queue«, also eine Datenstruktur für eine Warteschlange, aber mit Zugriff von zwei Seiten (Anfang und Ende).

Tabelle 28.1: Container-Übersicht

Container-Art	Header	Klassen-Template
Sequenzen	<code><array></code> <code><deque></code> <code><list></code> <code><queue></code> <code><stack></code> <code><vector></code>	<code>array</code> <code>deque</code> <code>list</code> <code>queue</code> <code>stack</code> <code>vector</code>
Sortierte Assoziative Container	<code><map></code> <code><set></code>	<code>map</code> <code>multimap</code> <code>set</code> <code>multiset</code>
Ungeordnete Assoziative Container	<code><unordered_map></code> <code><unordered_set></code>	<code>unordered_map</code> <code>unordered_multimap</code> <code>unordered_set</code> <code>unordered_multiset</code>
Spezialfälle	<code><bitset></code> <code><vector></code> <code><queue></code>	<code>bitset</code> <code>vector<bool></code> <code>priority_queue</code>

Assoziative Container

erlauben den schnellen Zugriff auf Daten anhand eines Schlüssels. Es gibt zwei verschiedene Konzepte:

1. Eine Abbildung (englisch *map*) beschreibt eine Beziehung zwischen Elementen zweier Mengen. `map`-Container unterstützen dieses Konzept. So könnte die englische Bezeichnung für ein Buch durch Übergabe des Strings »Buch« als Schlüssel an eine geeignete Methode des Containers ermittelt werden. In diesem Fall ist dem Schlüssel »Buch« das Datum »book« zugeordnet. Es wird also eine Menge von Schlüsseln auf eine Menge von zugeordneten Daten abgebildet. Die Elemente eines solchen Containers sind Paare von Schlüsseln und Daten. Der Typ `map` kennzeichnet eine eindeutige Abbildung, weil ein Schlüssel genau *einem* Datum zugeordnet ist (Ausnahme: `multimap`).
2. Eine Menge (englisch *set*) ist eine Ansammlung unterscheidbarer Objekte, Elemente genannt, die gemeinsame Eigenschaften haben. $\mathcal{N} = \{0; 1; 2; 3; \dots\}$ bezeichnet zum Beispiel die Menge der natürlichen Zahlen. `set`-Container unterstützen die Umsetzung dieses Konzepts im Computer. Weil die Elemente unterscheidbar sein müssen, kann es keine zwei gleichen Elemente in einer Menge geben (Ausnahme: `multiset`).

C++ unterscheidet zusätzlich – je nach den zugrunde liegenden Datenstrukturen – zwei verschiedene Arten:

■ Sortierte Assoziative Container

In der zugrunde liegenden Datenstruktur liegen die Elemente sortiert vor, weil sie bereits beim Einfügen richtig einsortiert werden. Beim GNU C++-Compiler (und vermutlich auch allen anderen) ist die Datenstruktur ein sogenannter Rot-Schwarz-Baum [CLR], eine Variante des binären Suchbaums.

■ Ungeordnete Assoziative Container

Diese Art von Containern bietet sich an, wenn eine Sortierung nicht erforderlich ist. Die zugrunde liegende Datenstruktur basiert auf einem Streuspeicher-Verfahren. Dabei werden die Elemente nach einem (Pseudo-) Zufallsprinzip auf den Speicherbereich »verstreut« (englisch *hashing*). Besonderes Merkmal: Die Zugriffszeit zu einem Element ist bei geeigneter Auslegung von Speicher und Zufallsfunktion konstant, also *unabhängig* von der Anzahl der gespeicherten Elemente.

28.1 Gemeinsame Eigenschaften

Ehe ich auf die einzelnen Container-Klassen eingehe, beschreibe ich in diesem Abschnitt gemeinsame Eigenschaften. Tabelle 28.2 zeigt die von den Containern der C++-Standardbibliothek zur Verfügung gestellten und von selbst gebauten Containern zu fordernden Datentypen eines Containers. Dabei ist X der Datentyp des Containers, zum Beispiel `vector<int>`, und T der Datentyp eines Container-Elements, zum Beispiel `int`. Der Typ `vector<int>::value_type` ist dann identisch mit `int`. Diese Datentypen sind in allen Container-Klassen der Standardbibliothek enthalten. Mit `const_reference` und `const_iterator` können Container-Elemente nicht verändert werden, nur ein lesender Zugriff ist möglich.

Jeder Container stellt einen öffentlichen Satz von Methoden zur Verfügung, die in einem Programm eingesetzt werden können. Die Methoden `begin()` und `end()` für Vektoren wurden bereits erwähnt (Seite 399). Tabelle 28.3 zeigt die von den Containern der C++-Standardbibliothek zur Verfügung gestellten Methoden. X sei wieder die Bezeichnung des Container-Typs. Diese Methoden sind in allen Container-Klassen der Standardbibliothek enthalten.

Tabelle 28.2: Container-Datentypen

Datentyp	Bedeutung
<code>X::value_type</code>	Typ der gespeicherten Elemente
<code>X::reference</code>	Referenz auf Container-Element
<code>X::const_reference</code>	dito, aber nur lesend verwendbar
<code>X::iterator</code>	Iterator
<code>X::const_iterator</code>	nur lesend verwendbarer Iterator
<code>X::difference_type</code>	vorzeichenbehafteter integraler Typ
<code>X::size_type</code>	integraler Typ ohne Vorzeichen für Größenangaben

X = Typ des Containers, zum Beispiel `vector`

Die Funktion `swap()` ist sehr schnell, weil intern nur die Verweise auf den Speicherbereich vertauscht werden. Ausnahme ist das Klassen-Template `array`, weil es die Daten direkt kapselt. Die für `swap()` benötigte Zeitdauer ist proportional zur Anzahl der Elemente des `array`-Objekts.

Tabelle 28.3: Container-Methoden

Rückgabetyyp Methode	Bedeutung
X()	Standardkonstruktor; erzeugt leeren Container
X(const X&)	Kopierkonstruktor
X(X&&)	bewegender Konstruktor für temporäre Parameter (R-Werte)
~X()	Destruktor; ruft die Destrukturen aller Elemente des Containers auf
X(il)	il ist eine Initialisierungsliste, siehe Seite 767.
iterator begin()	gibt Iterator zurück, der auf das erste Element des Containers, falls vorhanden, verweist; andernfalls wird end() zurückgegeben
const_iterator begin()	dito, aber mit diesem Iterator kann das Element nicht verändert werden
const_iterator cbegin()	dito (siehe Text)
iterator end()	Position nach dem letzten Element
const_iterator end()	dito
const_iterator cend()	dito (siehe Text)
size_type size()	Größe des Containers = Anzahl der aktuell gespeicherten Elemente = (end() - begin())
size_type max_size()	maximal mögliche Größe des Containers
bool empty()	(size() == 0) bzw. (begin() == end())
void swap(X&)	Vertauschen mit Argument-Container
X& operator=(const X&)	Zuweisungsoperator

X = Typ des Containers, zum Beispiel vector

begin(), end(), cbegin(), cend()

Betrachten Sie die folgende Schleife zur Ausgabe einer Liste:

```
for(auto it = einContainer.begin();
    it != einContainer.end(); ++it) {
    cout << *it << endl;
}
```

Mit der Abkürzung auto wird dem Compiler die Ermittlung des Iteratortyps überlassen, siehe Seite 90. Falls container eine const-Referenz auf einen Container ist, geben begin() und end() einen const_iterator zurück, andernfalls einen iterator. Es kann aber sein, dass container nicht const ist und dennoch wie in unserem Beispiel im Schleifenkörper nur lesende Operationen ausgeführt werden sollen. Dann empfehlen sich die neu in den Standardentwurf aufgenommenen Methoden cbegin() und cend(). Falls dann versehentlich eine verändernde Anweisung wie *it = wert; im Schleifenkörper stünde, würde schon der Compiler einen Fehler melden. Die Alternative ist also

```
for(auto it = container.cbegin();
    it != container.cend(); ++it) {
    // Der Compiler erlaubt nur lesenden Zugriff.
    cout << *it << endl;
}
```

Kurzform für for-Schleifen

Es ist mühsam, jedesmal `begin()` und `end()` schreiben zu müssen, um über einen Container zu iterieren. Der neue C++-Standard¹ erlaubt es, mit einer `for`-Schleife über einen ganzen Bereich zu iterieren. Dabei kann für den Bereich alles angegeben werden, wofür man `begin()` und `end()` definieren würde, also Container, Strings u.a. Es geht also noch besser, weil erheblich kürzer und angenehmer zu schreiben und zu lesen. Das obige Beispiel mit der Kurzform formuliert:

```
for(auto wert : container) {  
    cout << wert << endl;  
}
```

Man kann die Schleife so lesen: Führe den Schleifenkörper für alle Objekte `wert` in `container` aus. Es sind auch ändernde Operationen möglich, wenn eine Referenz angegeben wird:

```
for(double& wert : v) {  
    wert = 0.0;           // setzt alle Werte von v auf 0.0  
}
```

Relationale Operatoren

Für alle Container mit Ausnahme der `priority_queue` und der ungeordneten assoziativen Container gibt es die bekannten relationalen Operatoren `=`, `!=`, `<`, `<=`, `>` und `>=`, sodass zwei Container miteinander verglichen werden können. Das Ergebnis basiert auf einem lexikografischen Vergleich. Voraussetzung ist natürlich die Vergleichbarkeit der in den Containern gespeicherten Elemente.

Schreibweisen

Dieser Abschnitt beschreibt einige Konventionen zur Schreibweise, die für alle folgenden Container-Beschreibungen gelten.

`p` und `q` sind dereferenzierbare Iteratoren desselben Containers (Typ `iterator`).

Oft müssen Bereiche angegeben werden. Dafür wird die in der Mathematik übliche Notation für Intervalle verwendet. Eckige Klammern bezeichnen dabei Intervalle einschließlich der Grenzwerte, runde Klammern Intervalle ausschließlich der Grenzwerte. Im folgenden Text ist `[i, j)` also ein Intervall einschließlich `i` und ausschließlich `j`.

`i` und `j` sind vom Typ eines Input-Iterators. Mit `InputIterator` ist ein Iterator gemeint, der zum Lesen verwendet wird und der zu einem anderen Container, auch unterschiedlichen Typs, gehört.

Ein Parameter `n` ist von dem integralen Typ `size_type` des Containers. Ein Argument `t` ist ein Element des Container-Typs `value_type` (identisch mit dem Template-Parameter `T`).

28.1.1 Initialisierungslisten

Initialisierungslisten sind von den C-Arrays bekannt (Seite 192), und auch von der Initialisierung von Objekten (Seite 155). Das Konzept wurde erheblich ausgedehnt, um es für STL-Container nutzbar zu machen, zum Beispiel sind die Anweisungen

¹ Wird vom GNU C++-Compiler erst ab Version 4.6 unterstützt.

```
vector<string> vs = {"gute", "Idee"};
vector<int> vi = {1, 2, 3};
vi.insert(vi.end(), {4, 5, 6});
```

damit möglich. Zur Realisierung wurde ein eigener Typ `initializer_list` entworfen, mit dem Konstruktoren entworfen werden können, zum Beispiel (Zitat aus [ISOC++], Abschnitt 8.5.4):

```
struct S {
    S(std::initializer_list<double>); // #1
    S(std::initializer_list<int>); // #2
    // ...
};
S s1 = { 1.0, 2.0, 3.0 }; // invoke #1
S s2 = { 1, 2, 3 }; // invoke #2
```

Es kann eine beliebige Anzahl von Elementen übergeben werden – für einen Container also ideal.

28.1.2 Konstruktion an Ort und Stelle

Eine weitere Neuerung ist das Einfügen mit der Methode `emplace()`, die eine Konstruktion an Ort und Stelle bewirkt, um eine Kopie zu vermeiden. `emplace()` entspricht insofern grob dem Placement-new aus Abschnitt 33.2. Der Vergleich mit der Vektor-Methode `push_back()` zeigt den Effekt.

```
// cppbuch/k28/emplace/vec.cpp
#include<vector>
class A {
public:
    A(int i, int j) : i_(i), j_(j) { }
private:
    int i_;
    int j_;
};
using namespace std;

int main() {
    vector<A> v;
    A a1(1,2);
    v.push_back(a1); // echte Kopie (default-Kopierkonstruktor)
    // v.push_back(3, 4); nicht erlaubt!
    v.push_back({3, 4}); // Initialisierungsliste: erlaubt (entspricht A{3, 4})
    v.emplace(v.end(), 3, 4); // nur Konstruktoraufruf A(3, 4),
    // kein nachfolgender Aufruf des Kopierkonstruktors
}
```

28.1.3 Reversible Container

Eine einfach verkettete Liste kann nur in einer Richtung durchlaufen werden, eine doppelt verkettete Liste in zwei Richtungen, nämlich auch vom Ende zum Anfang. Alle Container mit dieser Eigenschaft heißen Reversible Container. Die Iteratoren für solche

Container sind bidirektional, das heißt, sie können mit dem ++-Operator einen Schritt vorwärts und mit dem ---Operator einen Schritt rückwärts gehen.

Für solche Container werden zusätzliche Iteratoren bereitgestellt, die vom Ende her mit dem ++-Operator bis an den Anfang laufen können. Diese Iteratoren haben den Typ `reverse_iterator` und können mit den entsprechenden Methoden ermittelt werden:

```
const_reverse_iterator rbegin() const,
const_reverse_iterator crbegin() const  und
reverse_iterator rbegin()
```

geben einen Reverse-Iterator zurück, der auf das letzte Element zeigt.

```
const_reverse_iterator rend() const,
const_reverse_iterator crend() const  und
reverse_iterator rend()
```

geben einen Reverse-Iterator zurück, der auf eine fiktive Position vor dem ersten Element zeigt. Anwendungsbeispiel für einen Container `container` des Typs `vector<int>`:

```
// Mit Hilfe eines Iterators Daten ausgeben, dabei am Ende beginnen:
vector<int>::reverse_iterator iter = container.rbegin();
while(iter != container.rend()) {
    cout << (*iter) << endl;
    ++iter;
}
```

28.2 Sequenzen

Die grundlegenden Sequenz-Typen sind `list`, `vector` und `deque`. Ein `stack`-Objekt kann mit einem dieser Typen gebildet werden. Eine Liste (`list`) ist empfehlenswert, wenn oft Elemente in der Mitte eingefügt oder gelöscht werden sollen. Eine »double ended queue« (`deque`) sollte man nehmen, wenn Einfügen und Löschen am Anfang oder Ende gefragt sind. Wenn keine besonderen Anforderungen vorliegen, ist ein `vector` die richtige Wahl. Zusätzlich zu den Methoden der Tabelle 28.3 gelten für Sequenzen die in der Tabelle 28.4 aufgeführten Methoden.



Tipp

iterator-Rückgabewerte der Methoden nutzen! Grund: Die übergebenen Iteratoren und Iteratoren auf nachfolgende Elemente werden ungültig.

Die Zeitkomplexität (siehe Glossar Seite 957) der Methoden hängt von der Art der Sequenz ab. So ist die Zeitkomplexität des Einfügens eines Elements in die Mitte eines Vektors $O(n)$, weil im Mittel $n/2$ Elemente verschoben werden. Der Zugriff auf ein Element in der Mitte mit dem []-Operator ist dagegen schnell ($O(1)$). Bei einer Liste ist es gerade umgekehrt: Einfügen geht schnell ($O(1)$), aber der Zugriff auf ein Element in der Mitte dauert ($O(n)$), weil die Liste durchlaufen werden muss, um das Element zu finden.

Tabelle 28.4: Sequenz-Methoden

Rückgabotyp Methode	Bedeutung
<code>X(n, t)</code>	Erzeugt eine Sequenz mit <code>n</code> Kopien von <code>t</code> .
<code>X(i, j)</code>	Erzeugt eine Sequenz aus den Elementen des Intervalls <code>[i, j)</code> .
<code>iterator emplace(p, args)</code>	Fügt ein Objekt, das mit den Parametern <code>args</code> konstruiert wird, vor <code>p</code> ein (siehe Seite 768). Der zurückgegebene Iterator zeigt auf die eingefügte Kopie.
<code>iterator insert(p, t)</code>	Fügt eine Kopie von <code>t</code> vor <code>p</code> ein. Der zurückgegebene Iterator zeigt auf die eingefügte Kopie.
<code>void insert(p, n, t)</code>	Fügt <code>n</code> Kopien von <code>t</code> vor <code>p</code> ein.
<code>void insert(p, i, j)</code>	Fügt Kopien der Elemente im Bereich <code>[i, j)</code> vor <code>p</code> ein. Die Iteratoren <code>i</code> und <code>j</code> dürfen nicht in den aufnehmenden Container verweisen.
<code>void insert(p, il)</code>	<code>il</code> ist eine Initialisierungsliste.
<code>iterator erase(q)</code>	Löscht das Element, auf das <code>q</code> zeigt. Der zurückgegebene Iterator zeigt anschließend auf das <code>q</code> folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (<code>end()</code>) zurückgegeben.
<code>iterator erase(p, q)</code>	Löscht alle Elemente des Bereichs <code>[p, q)</code> . Der zurückgegebene Iterator zeigt anschließend auf das <code>q</code> folgende Element, wenn es existiert, ansonsten wird ein End-Iterator zurückgegeben.
<code>void clear()</code>	Löscht alle Elemente.
<code>void assign(i, j)</code>	Alle Elemente löschen, danach Kopie von <code>[i, j)</code> einfügen. <code>i</code> und <code>j</code> dürfen nicht in den Container verweisen.
<code>void assign(il)</code>	<code>il</code> ist eine Initialisierungsliste.
<code>void assign(n, t)</code>	Alle Elemente löschen, danach <code>n</code> Kopien von <code>t</code> einfügen. Dabei darf <code>t</code> keine Referenz auf ein Element des Containers sein.

X = array, deque, list, vector

28.2.1 vector

Ein `vector`, eingebunden durch den Header `<vector>`, stellt öffentliche Typen und Methoden zur Verfügung, die nachfolgend kurz beschrieben werden. Vorab sei auf die schon genannten vorhandenen Datentypen und Methoden hingewiesen, um sie nicht zu wiederholen:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)
- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)
- Konstruktoren und Methoden der Tabelle 28.4 (Sequenzen, Seite 770)

Ein Vektor hat noch weitere öffentliche Datentypen, die in der Tabelle 28.5 aufgelistet sind. Die Deklaration der Klasse ist `template<typename T> class vector;`

Tabelle 28.5: Zusätzliche Datentypen für `vector`

Datentyp	Bedeutung
<code>pointer</code>	Zeiger auf Vektor-Element
<code>const_pointer</code>	dito, aber nur lesend verwendbar

Die folgende Aufstellung zeigt die Methoden eines Vektors, die zusätzlich zu denen, auf die am Abschnittsanfang verwiesen wird, benutzbar sind. Die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben.

- `const_reference front() const` und `reference front()`
liefern eine Referenz auf das erste Element.
- `const_reference back() const` und `reference back()`
liefern eine Referenz auf das letzte Element.
- `const_pointer data() const` und `pointer data()`
liefern einen Zeiger auf das erste Element. Es gilt also `data() == &front()`, falls der Vektor nicht-leer ist. Der Zeiger erlaubt den Zugriff auf die Daten des Vektors wie bei einem C-Array.
- `const_reference operator[](size_type n) const` und `reference operator[](size_type n)`
geben eine Referenz auf das n -te Element zurück.
- `const_reference at(size_type n) const` und `reference at(size_type n)`
geben eine Referenz auf das n -te Element zurück. Der Unterschied zum vorhergehenden `operator[]()` besteht in der Prüfung, ob n in einem gültigen Bereich liegt. Falls nicht, d.h. n ist $\geq \text{size}()$, wird eine `out_of_range`-Exception geworfen.
- `void emplace_back(args)` fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Ende ein.
- `void push_back(const T& t)` fügt t am Ende ein.
(`void push_back(T&& t)` für R-Werte)
- `void pop_back()`
löscht das letzte Element.
- `void resize(size_type n, T t = T())`
Vektorgröße ändern. Dabei werden $n - \text{size}()$ Elemente t am Ende hinzugefügt bzw. $\text{size}() - n$ Elemente am Ende gelöscht, je nachdem, ob n kleiner oder größer als die aktuelle Größe ist. Die Zeitkomplexität ist $O(|\text{size}() - n|)$.
- `void reserve(size_type n)`
Speicherplatz reservieren, sodass der verfügbare Platz (Kapazität) größer als der aktuell benötigte ist. Zweck: Vermeiden von Speicherbeschaffungsoperationen während der Benutzung des Vektors. Die Zeitkomplexität ist $O(n)$.
- `size_type capacity() const`
gibt den Wert der Kapazität zurück (siehe `reserve()`). `size()` ist immer kleiner oder gleich `capacity()`.

28.2.2 `vector<bool>`

Ein Bitset (Seite 801) alloziert festen Speicher, ein Vektor kann seinen Speicher dynamisch ändern. Falls dies auch für die Speicherung von Bits gewünscht ist und gleichzeitig nicht ein Byte pro Bit verschwendet werden soll, bietet sich eine Spezialisierung der

Klasse `vector` an: die Klasse `vector<bool>`. Sie hat dieselben öffentlichen Datentypen wie die Vektor-Klasse mit Ausnahme des Datentyps `reference`, der für Manipulationen an einzelnen Bits gedacht ist:

```
// Typ vector<bool>::reference
class reference {
    friend class vector;
    reference();
public:
    ~reference();
    reference& operator=(const bool x); // für b[i] = x;
    reference& operator=(const reference&); // für b[i] = b[j];
    operator bool() const; // für x = b[i];
    void flip(); // für b[i].flip();
};
```

Ein `bool`-Vektor hat die Methoden eines Vektors `vector<T>`, wenn anstelle des Platzhalters `T` für den Typ `bool` eingesetzt wird. Die Spezialisierung `vector<bool>` bietet zusätzlich die Methode `void flip()` an, die alle Elemente negiert. Das folgende kleine Programm gibt *false true false false* aus:

Listing 28.1: Bool-Vektor

```
// cppbuch/k28/vector/vectorbool.cpp
#include<vector>
#include<iostream>
#include<showSequence.h> // siehe Seite 648
using namespace std;

int main() {
    vector<bool> vecBool(4, true); // alles true
    vecBool.flip(); // alles false
    vecBool[1].flip(); // Bit 1 wird true
    cout.setf(ios_base::boolalpha);
    showSequence(vecBool);
}
```



Hinweis

`vector<bool>` speichert die Werte als ein Bit pro `bool` ab, also sehr kompakt, und verhält sich daher nicht ganz wie andere Vektoren. So kann nicht die Adresse eines Elements genommen werden. Wenn die Anzahl der Bits nicht veränderbar sein muss, empfiehlt sich `bitset` als Alternative (Seite 801).

28.2.3 list

Die Liste dieses Abschnitts ist eine doppelt-verkettete Liste, die das Hinzufügen und Entnehmen von Elementen an jeder Stelle mit konstanter Zeit erlaubt, also unabhängig von der Zahl bereits gespeicherter Elemente. Die Klasse `list`, eingebunden durch den Header `<list>`, stellt öffentliche Typen und Methoden zur Verfügung, die nachfolgend kurz beschrieben werden. Vorab sei auf die schon genannten vorhandenen Datentypen und Methoden hingewiesen, um sie hier nicht im Einzelnen zu wiederholen:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)
- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)
- Konstruktoren und Methoden der Tabelle 28.4 (Sequenzen, Seite 770)

Eine Liste hat wie ein Vektor noch die Datentypen `pointer` und `const_pointer` für Zeiger auf Listenelemente (siehe Tabelle 28.5). Die Deklaration der Klasse ist

```
template<typename T> class List;
```

Die folgende Aufstellung zeigt die Methoden einer Liste, die zusätzlich zu den oben erwähnten benutzbar sind. Die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben. Das n in $O(n)$ ist die Anzahl der Elemente, d.h. identisch mit dem Wert von `size()`.

- `const_reference front() const` und `reference front()`
liefern eine Referenz auf das erste Element einer Liste.
- `const_reference back() const` und `reference back()`
liefern eine Referenz auf das letzte Element einer Liste.
- `void emplace_front(args)` fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Anfang ein.
- `void emplace_back(args)` fügt ein Objekt, dessen Konstruktor mit den Parametern `args` aufgerufen wird, am Ende ein.
- `void push_front(const T& t)` fügt `t` am Anfang ein. (`void push_front(T&& t)` für R-Werte)
- `void pop_front()` löscht das erste Element.
- `void push_back(const T& t)` fügt `t` am Ende ein. (`void push_back(T&& t)` für R-Werte)
- `void pop_back()`
löscht das letzte Element.
- `void remove(const T& t)`
entfernt alle Elemente, die gleich dem übergebenen Element `t` sind. Die Zeitkomplexität ist $O(n)$.
- `template<class Praedikat> void remove_if(Praedikat P)`
entfernt alle Elemente, auf die das Prädikat zutrifft. Die Zeitkomplexität ist $O(n)$.
- `void resize(size_type neu, T t = T())`
Listengröße ändern. Dabei werden `neu - size()` Elemente `t` am Ende hinzugefügt bzw. `size() - neu` Elemente am Ende gelöscht, je nachdem, ob `neu` kleiner oder größer als die aktuelle Größe ist. Die Zeitkomplexität ist $O(|n - neu|)$.
- `void reverse()`
kehrt die Reihenfolge der Elemente in der Liste um (Zeitkomplexität $O(n)$).
- `void sort()`
sortiert die Elemente in der Liste. Die Zeitkomplexität ist $O(n \log n)$. Sortierkriterium ist der für die Elemente definierte Operator `<`.

- `template<class Compare> void sort(Compare cmp)`
wie `sort()`, aber mit dem Sortierkriterium des Compare-Objekts `cmp`. Das Compare-Objekt ist ein Funktionsobjekt (siehe Seite 344).
- `void unique()`
löscht gleiche aufeinanderfolgende Elemente bis auf das erste (Zeitkomplexität $O(n)$). Anwendung auf eine sortierte Liste bedeutet, dass danach kein Element mehrfach auftritt.
- `template<class binaeresPraedikat> void unique(binaeresPraedikat P)`
dito, nur dass statt des Gleichheitskriteriums ein anderes binäres Prädikat genommen wird.
- `void merge(List& L)`
Verschmelzen zweier sortierter Listen (Zeitkomplexität $O(n1 + n2)$). Die aufgerufene Liste `L` ist anschließend leer, die aufrufende enthält nachher alle Elemente.
- `template<class Compare> void merge(List& L, Compare cmp)`
wie vorher, aber für den Vergleich von Elementen wird ein Compare-Objekt genommen.
- `void splice(iterator pos, List& x)`
fügt den Inhalt von Liste `x` vor `pos` ein. `x` ist anschließend leer.
- `void splice(iterator p, List& x, iterator i)`
Fügt Element `*i` aus `x` vor `p` ein und entfernt `*i` aus `x`.
- `void splice(iterator pos, List& x, iterator first, iterator last)`
fügt Elemente im Bereich `[first, last)` aus `x` vor `pos` ein und entfernt sie aus `x`. Bei dem Aufruf für dasselbe Objekt (das heißt `&x == this`) wird konstante Zeit benötigt, ansonsten ist der Aufwand von der Ordnung $O(last - first)$. `pos` darf nicht im Bereich `[first, last)` liegen.

Das folgende kleine Programm zeigt, wie zwei Listen mit `merge()` verschmolzen werden.

Listing 28.2: Verschmelzen zweier Listen

```
// cppbuch/k28/list/merge.cpp
#include<list>
#include<showSequence.h> // siehe Seite 648
using namespace std;

int main() {
    list<int> L1, L2;
    // Listen mit sortierten Zahlen füllen:
    for(int i = 0; i < 5; ++i) {
        L1.push_back(2*i);    // gerade Zahlen
        L2.push_back(2*i+1);  // ungerade Zahlen
    }
    showSequence(L1); // 0 2 4 6 8
    showSequence(L2); // 1 3 5 7 9
    L1.merge(L2);      // verschmelzen
    showSequence(L1);  // 0 1 2 3 4 5 6 7 8 9
    showSequence(L2);  // (leere Liste)
}
```

28.2.4 deque

Der Name Deque ist die Abkürzung für *double ended queue*, also eine Warteschlange, die das Hinzufügen und Entnehmen von Elementen sowohl am Anfang als auch am Ende erlaubt. Die Klasse deque, eingebunden durch den Header `<deque>`, stellt öffentliche Typen und Methoden zur Verfügung, die nachfolgend kurz beschrieben werden. Vorab sei auf die schon genannten vorhandenen Datentypen und Methoden hingewiesen, um sie hier nicht im Einzelnen zu wiederholen:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)
- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)
- Konstruktoren und Methoden der Tabelle 28.4 (Sequenzen, Seite 770)

Eine Deque hat wie ein Vektor noch die Datentypen `pointer` und `const_pointer` für Zeiger auf Deque-Elemente (siehe Tabelle 28.5). Die Deklaration der Klasse ist

```
template<typename T> class deque;
```

Die folgende Aufstellung zeigt die Methoden einer Deque, die zusätzlich zu den eben erwähnten benutzbar sind.

- `const_reference front() const` und `reference front()`
liefern eine Referenz auf das erste Element eines Deque-Objekts.
- `const_reference back() const` und `reference back()`
liefern eine Referenz auf das letzte Element eines Deque-Objekts.
- `const_reference operator[](size_type n) const` und `reference operator[](size_type n)`
geben eine Referenz auf das n-te Element zurück.
- `const_reference at(size_type n) const` und `reference at(size_type n)`
geben eine Referenz auf das n-te Element zurück. Der Unterschied zum vorhergehenden `operator[]()` besteht in der Prüfung, ob n in einem gültigen Bereich liegt. Falls nicht, d.h. $n \geq \text{size}()$, wird eine `out_of_range`-Exception geworfen.
- `void emplace_front(args)` fügt ein Objekt, dessen Konstruktor mit den Parametern args aufgerufen wird, am Anfang ein.
- `void emplace_back(args)` fügt ein Objekt, dessen Konstruktor mit den Parametern args aufgerufen wird, am Ende ein.
- `void push_front(const T& t)` fügt t am Anfang ein.
(`void push_front(T&& t)` für R-Werte)
- `void pop_front()` löscht das erste Element.
- `void push_back(const T& t)` fügt t am Ende ein.
(`void push_back(T&& t)` für R-Werte)
- `void pop_back()` löscht das letzte Element.

■ `void resize(size_type n, T t = T())`

Deque-Größe ändern. Dabei werden $n - \text{size}()$ Elemente `t` am Ende hinzugefügt bzw. $\text{size}() - n$ Elemente am Ende gelöscht, je nachdem, ob n kleiner oder größer als die aktuelle Größe ist.

28.2.5 stack

Ein Stack (Header `<stack>`) ist ein Container, der Ablage und Entnahme nur von einer Seite erlaubt. Zuerst abgelegte Objekte werden zuletzt entnommen. Ein Stack benutzt intern einen anderen Container, der die Operationen `back()`, `push_back()` und `pop_back()` unterstützt. Die Stack-Methoden delegieren ihre Aufgabe an die entsprechenden Methoden des Containers, etwa wie auf Seite 299 beschrieben. Es kommen die Container `vector`, `list` und `deque` in Frage. Falls nichts anderes angegeben wird, wird eine Deque (siehe Seite 775) verwendet. Ein Stack stellt zusätzlich den Datentyp `container_type` bereit.

```
// mit deque realisierter Stack (Voreinstellung)
stack<double> Stack1;
// mit vector realisierter Stack
stack<double, vector<double> > Stack2;
```

Die Deklaration der Klasse ist

```
template<typename T, class Container = deque<T> > class stack;
```

Einem Stack stehen die relationalen Operatoren (Seite 767) sowie die folgenden Methoden zur Verfügung. Die Zeitkomplexität ist jeweils $O(1)$, wenn nicht anders angegeben.

- `stack(const Container& cont = Container())`
Konstruktor. Ein Stack kann mit einem bereits vorhandenen Container initialisiert werden. Container ist der Typ des Containers. In diesem Fall ist die Zeitkomplexität $O(\text{cont.size}())$.
- `bool empty() const`
gibt zurück, ob der Stack leer ist.
- `size_type size() const`
gibt die Anzahl der im Stack befindlichen Elemente zurück.
- `const value_type& top() const` und `value_type& top()`
geben das oberste Element zurück.
- `void push(const value_type& x)`
legt das Element `x` auf dem Stack ab.
- `void pop()`
entfernt das oberste Element vom Stack.

Die Typen `size_type` und `value_type` sind dem Container, der intern zur Realisierung des Stacks verwendet wird, entlehnt. Das folgende Programm zeigt eine `stack`-Anwendung.

Listing 28.3: Stack

```
// cppbuch/k28/stack.cpp
#include<stack>
#include<iostream>
using namespace std;
```

```
int main() {
    int numbers[] = {1, 5, 6, 0, 9, 1, 8, 7, 2};
    const int COUNT = sizeof(numbers)/sizeof(int);
    stack<int> einStack;
    cout << "Zahlen auf dem Stack ablegen:" << endl;
    for(int i = 0; i < COUNT; ++i) {
        cout.width(6);
        cout << numbers[i]; // protokollieren
        einStack.push(numbers[i]);
    }
    cout << endl;
    cout << "Zahlen vom Stack holen (umgekehrte Reihenfolge!),"
        " anzeigen und löschen:" << endl;
    while(!einStack.empty()) {
        cout.width(6);
        cout << einStack.top(); // obersten Wert anzeigen
        einStack.pop();         // Wert löschen
    }
    cout << endl;
}
```

28.2.6 queue

Eine Queue (Header `<queue>`) oder Warteschlange erlaubt die Ablage von Objekten auf einer Seite und ihre Entnahme von der anderen Seite. Die Objekte an den Enden der Queue können ohne Entnahme gelesen werden. Sowohl `List` als auch `deque` sind geeignete Datentypen zur Implementierung. Falls nichts anderes angegeben wird, wird eine Deque (siehe Seite 775) verwendet. Eine Queue stellt zusätzlich den Datentyp `container_type` bereit.

```
// mit deque realisierte Queue (Voreinstellung)
queue<double> Queue1;
// mit list realisierte Queue
queue<double, list<double> > Queue2;
```

Die Deklaration der Klasse ist

```
template<typename T, class Container = deque<T> > class queue;
```

Einer Queue stehen die relationalen Operatoren (Seite 767) sowie die folgenden Methoden zur Verfügung. Die Zeitkomplexität ist jeweils $O(1)$, wenn nicht anders angegeben.

- `queue(const Container& cont = Container())`
Konstruktor. Eine Queue kann mit einem bereits vorhandenen Container initialisiert werden. Container ist der Typ des Containers. In diesem Fall ist die Zeitkomplexität $O(cont.size())$.
- `size_type size() const`
gibt die Anzahl der in der Queue befindlichen Elemente zurück.
- `const value_type& front() const` und
`value_type& front()`
geben das erste Element zurück.

- `const value_type& back() const` und `value_type& back()`
geben das letzte Element zurück.
- `void push(const value_type& x)`
fügt das Element `x` am Ende der Queue ein.
- `void pop()`
entfernt das erste Element der Queue.
- `bool empty() const`
gibt zurück, ob die Queue leer ist.

Die Typen `size_type` und `value_type` sind dem Container, der intern zur Realisierung der Queue verwendet wird, entlehnt. Das folgende Programm zeigt eine `queue`-Anwendung.

Listing 28.4: Warteschlange

```
// cppbuch/k28/queue.cpp
#include<queue>
#include<iostream>
using namespace std;

int main() {
    int numbers[] = {1, 5, 6, 0, 9, 1, 8, 7, 2};
    const int COUNT = sizeof(numbers)/sizeof(int);
    queue<int> eineQueue;
    cout << "Zahlen in die Warteschlange schreiben:" << endl;
    for(int i = 0; i < COUNT; ++i) {
        cout.width(6);
        cout << numbers[i];          // protokollieren
        eineQueue.push(numbers[i]);
    }
    cout << endl;
    cout << "Zahlen aus der Warteschlange holen, "
         "anzeigen und löschen:" << endl;
    while(!eineQueue.empty()) {
        cout.width(6);
        cout << eineQueue.front(); // ersten Wert anzeigen
        eineQueue.pop();          // Wert löschen
    }
    cout << endl;
}
```

28.2.7 priority_queue

Eine Priority-Queue ist eine prioritätsgesteuerte Warteschlange. Jedem Element ist eine Priorität zugeordnet, die den Platz innerhalb der Priority-Queue schon beim Einfügen bestimmt. Die relative Priorität wird durch Vergleich jeweils zweier Elemente bestimmt, indem entweder der `<`-Operator oder wahlweise ein Funktionsobjekt zum Vergleich herangezogen wird. Sowohl `vector` als auch `deque` sind geeignete Datentypen zur Implementierung. Falls nichts anderes angegeben ist, wird `deque` (Seite 775) verwendet.

Die genannten Container, als Sequenz eingesetzt, wären nicht gut geeignet, denn das einfache Einfügen eines Elements in die Mitte eines Vektors (das Element habe mittlere

Priorität) hat die Zeitkomplexität $O(n)$, weil ja etwa $n/2$ Elemente nach hinten geschoben werden müssten. Weil aber ein wahlfreier Zugriff mit der Methode `at(int i)` oder dem Index-Operator `[]` möglich ist, wird der implementierende Container intern nicht als Sequenz, sondern als *Heap* aufgebaut. Der Zeitaufwand, ein Element hinzuzufügen, sinkt damit auf $O(\log n)$, eine drastische Verbesserung bei großen Werten von n . Einzelheiten zur Heap-Struktur finden Sie auf Seite 688. Die Deklaration der Klasse ist

```
template<class T, class Container = deque<T>,
        class Compare = less<typename Container::value_type> >
class priority_queue;
```

Eine Priority-Queue stellt dieselben Datentypen wie die Queue und die folgenden Methoden zur Verfügung. Die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben.

```
■ priority_queue(const Compare& cmp = Compare(),
               const Container& = Container())
```

Konstruktor. Eine Priority-Queue kann mit einem bereits vorhandenen Container initialisiert werden. Container ist der Typ des Containers. In diesem Fall ist die Zeitkomplexität $O(\text{cont.size}())$. `cmp` ist das Funktionsobjekt, mit dem verglichen wird. `less<Container::value_type>` wird angenommen, falls kein Typ angegeben ist.

```
■ template<class InputIterator>
priority_queue(InputIterator first, InputIterator last,
               const Compare& cmp = Compare(),
               const Container& cont = Container())
```

Dieser Konstruktor unterscheidet sich von dem vorhergehenden, indem die Elemente im Bereich `[first, last)` eines anderen Containers bei der Konstruktion zusätzlich eingefügt werden. Die Zeitkomplexität ist $O(\text{last} - \text{first} + \text{cont.size}())$.

```
■ bool empty() const
```

gibt zurück, ob die Priority-Queue leer ist.

```
■ size_type size() const
```

gibt die Anzahl der in der Priority-Queue befindlichen Elemente zurück.

```
■ const value_type& top() const
```

gibt das erste Element zurück, d.h. das mit der größten Priorität.

```
■ void push(const value_type& x)
```

fügt das Element `x` ein. Die Zeitkomplexität ist $O(\log n)$.

```
■ void pop()
```

entfernt das erste Element. Die Zeitkomplexität ist $O(\log n)$.

Für die `priority_queue`-Klasse gibt es keine relationalen Operatoren. Das folgende Programm zeigt eine `priority_queue`-Anwendung.

Listing 28.5: Prioritätsgesteuerte Warteschlange

```
// cppbuch/k28/priorityqueue.cpp
#include<queue>
#include<vector>
#include<iostream>
using namespace std;

int main() {
```



```

int numbers[] = {1, 5, 6, 0, 9, 1, 8, 7, 2};
const int COUNT = sizeof(numbers)/sizeof(int);
// Standard-Implementierung mit deque und less
priority_queue<int> einePrioQueue;
cout << "Zahlen in die Prioritäts-Warteschlange schreiben:"
    << endl;
for(int i = 0; i < COUNT; ++i) {
    cout.width(6);
    cout << numbers[i];          // protokollieren
    einePrioQueue.push(numbers[i]);
}
cout << endl;
cout << "Zahlen aus der Prioritäts-Warteschlange holen,"
    " anzeigen und löschen:" << endl;
while(!einePrioQueue.empty()) {
    cout.width(6);
    cout << einePrioQueue.top(); // 'wichtigsten' Wert anzeigen
    einePrioQueue.pop();        // Wert löschen
}
cout << endl;
}

```

Im Beispiel werden die größeren Zahlen zuerst ausgegeben. Wenn kleinen Zahlen eine hohe Priorität zugeordnet werden soll, ist `greater` statt `less` einzusetzen. Weil dieser vordefinierte Template-Parameter an dritter Stelle steht, ist auch der zweite Parameter, der Container für die interne Implementierung, anzugeben. Beispiel mit `vector`:

```
priority_queue<int, vector<int>, greater<int> > einePrioQueue;
```

28.2.8 array

Die Algorithmen der C++-Standardbibliothek, die auf Containern arbeiten, benutzen oft die Methoden `begin()`, `end()` und `size()`. Ein C-Array ist keine Klasse und hat diese Methoden daher nicht. Die Anzahl der Elemente eines C-Arrays kann mit `sizeof` nur unter bestimmten Voraussetzungen ermittelt werden (siehe Seite 191). Das Klassen-Template `array` (Header `<array>`) ist eine Art Hüllklasse (englisch *wrapper*), die ein C-Array kapselt und die entsprechenden Methoden bereitstellt. Im Unterschied zu einem Vektor hat ein `array` eine fixe, zur Compilationszeit festgelegte Größe. Ein `array` ist zwar eine Sequenz, die Methoden unterscheiden sich aber von denen der Tabelle 28.4 (Seite 770). Die Deklaration eines `array` ist

```
template<typename T, size_t N> class array;
```

Dabei gibt `T` den Typ der abzuspeichernden Elemente an. `N` ist die Anzahl der Elemente. Es gibt nur den (vorgegebenen) Standardkonstruktor. Ein `array` hat strukturelle Ähnlichkeit mit dem Beispiel eines Stacks festgelegter Größe aus Abschnitt 6.3.2 (Seite 249). Ein `array` stellt die folgenden öffentlichen Typen und Methoden zur Verfügung:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)

- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)

Darüber hinaus gibt es die folgenden Methoden. Die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben.

- `void assign(const T& t)`
weist allen Elementen des Arrays eine Kopie von `t` zu. Die Zeitkomplexität ist $O(N)$.
- `const_reference front() const` und `reference front()`
liefern eine Referenz auf das erste Element.
- `const_reference back() const` und `reference back()`
liefern eine Referenz auf das letzte Element.
- `const_pointer data() const` und `pointer data()`
liefern einen Zeiger auf das erste Element. Es gilt also `data() == &front()`, falls das Array nicht-leer ist.
- `const_reference operator[](size_type n) const` und `reference operator[](size_type n)`
geben eine Referenz auf das n -te Element zurück.
- `const_reference at(size_type n) const` und `reference at(size_type n)`
geben eine Referenz auf das n -te Element zurück. Der Unterschied zum vorhergehenden `operator[]()` besteht in der Prüfung, ob n in einem gültigen Bereich liegt. Falls nicht, d.h. n ist $\geq \text{size}()$, wird eine `out_of_range`-Exception geworfen.

Falls das Array leer ist (d.h. $N == 0$), ist das Ergebnis der Funktionen `data()`, `front()` und `back()` undefiniert. `size()` liefert natürlich stets den unveränderlichen Wert N . Ein Array dieser Art kann wie ein Tupel (siehe Abschnitt 27.4 auf Seite 752) mit N Elementen aufgefasst werden. Um kompatibel zu `tuple` zu sein, besitzt `array` die entsprechenden `get()`-Methoden, für deren Aufruf die Voraussetzung ist $0 \leq i < N$ gelten muss.

```
template <int I, class T, size_t N>
T& get(array<T, N>& a);
```

```
template <int I, class T, size_t N>
const T& get(const array<T, N>& a);
```

Das Beispiel unten zeigt einige Verwendungsmöglichkeiten eines `array`-Objekts. Wie zu sehen ist, kann es mit einer durch geschweifte Klammern begrenzten Liste initialisiert werden (vgl. Abschnitt 5.2.3 auf Seite 192).

Listing 28.6: Klasse `Array`

```
// cppbuch/k28/array.cpp
#include<array>
#include<iostream>
using namespace std;

int main() {
    const size_t ANZAHL = 3;
    array<int, ANZAHL> tabelle = {{ 9, -10, 5 }}; // Initialisierung;
```

```

for(auto i = tabelle.begin();
    i != tabelle.end(); ++i) {           // Benutzung mit Iterator
    cout << *i << endl;
}
for(size_t i = 0; i < tabelle.size(); ++i) {
    cout << tabelle[i] << endl;         // Benutzung wie ein Vektor
}
// Benutzung wie ein Tupel
cout << get<0>(tabelle) << endl;
cout << get<1>(tabelle) << endl;
cout << get<2>(tabelle) << endl;
}

```

28.3 Sortierte assoziative Container

Wegen der intern verwendeten Baumstruktur für die Daten ist die Zugriffszeit zu einem Element proportional zu $\log n$, wenn n die Anzahl der gespeicherten Elemente ist. `map` ist die programmtechnische Umsetzung einer Relation oder diskreten Abbildung im mathematischen Sinn. Das mathematische Konzept einer Menge wird mit `set` realisiert. `multimap` und `multiset` erlauben mehrfache identische Schlüssel.

28.3.1 `map`

Die Klasse `map<Key, T>`, eingebunden durch den Header `<map>`, speichert *Paare* von Schlüsseln und zugehörigen Daten. Dabei ist der Schlüssel eindeutig, es kann also keine zwei Datensätze zu demselben Schlüssel geben. `map` ist ein assoziativer Container: Die Daten werden durch direkte Angabe des Schlüssels gefunden.

Damit ist ein `map`-Objekt eine Abbildung der enthaltenen Schlüssel auf die zugehörigen Daten. Obwohl die Schlüssel für einen assoziativen Zugriff nicht sortiert sein müssen, liegen sie in der Klasse `map` sortiert vor. Das ergibt den Vorteil der sortierten Ausgabe zum Beispiel beim Durchwandern vom ersten bis zum letzten Element. Dem steht der Nachteil einer längeren Zugriffszeit im Vergleich zu Datenstrukturen gegenüber, die auf der gestreuten Speicherung basieren (sogenannte Hash-Maps, siehe Abschnitt 28.4.1 weiter unten). Der Typ eines `map`-Elements ist `pair<const Key, T>`. Die Konstanz des Schlüssels ist notwendig, damit ein Schlüssel, der ja dank der Sortierung einer Position im Container entspricht, nicht geändert werden kann – dies würde die Sortierung zerstören. Die Deklaration der Klasse ist

```

template<class Key,           // Typ der Schlüssel
        class T,             // Typ der Daten
        class Compare = less<Key> > // Standardvergleich für Sortierung
class map;

```

Mit der `map`-Klasse lässt sich leicht ein Wörterbuch realisieren:

```

map<string, string> woerterbuch;

```

```
// Eintrag aufnehmen
woerterbuch.insert(pair<string, string>("Buch", "book"));
// hier lässt sich auch make_pair von Seite 751 einsetzen:
woerterbuch.insert(make_pair("Ziffer", "digit"));
// ... weitere Einträge aufnehmen

// assoziativer Zugriff auf einen Eintrag:
cout << woerterbuch["Buch"] << endl;    // book

// sortierte Ausgabe des ganzen Wörterbuchs
auto i = woerterbuch.begin();
while(i != woerterbuch.end()) {
    cout << (*i).first << " " << (*i).second << endl;
    ++i;
}
```

Falls die Sortierung nicht mit dem <-Operator (das heißt `less<Key>`) hergestellt werden soll, kann eine Klasse für Funktionsobjekte angegeben werden:

```
class Vergleich { ... }; // Klasse für Funktionsobjekt

// Funktionsobjekt Vergleich():
map<string, string, Vergleich> woerterbuch;

Vergleich einVergleich;
// Funktionsobjekt einVergleich:
map<string, string, Vergleich> woerterbuch(einVergleich);
```

Ein `map`-Container stellt unter anderem die folgenden öffentlichen Datentypen und Methoden bereit:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)
- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)

Es gibt weitere öffentliche Datentypen, die in der Tabelle 28.6 aufgelistet sind.

Tabelle 28.6: Zusätzliche Datentypen für `map<Key, T, Compare>`

Datentyp	Bedeutung
<code>pointer</code>	Zeiger auf Map-Element
<code>const_pointer</code>	nur lesend verwendbarer Zeiger
<code>key_type</code>	entspricht Key
<code>value_type</code>	entspricht <code>pair<const Key, T></code>
<code>mapped_type</code>	entspricht T
<code>key_compare</code>	<code>Compare</code>
<code>value_compare</code>	Klasse für Funktionsobjekte, siehe Funktion <code>value_comp()</code> auf Seite 785

Außer den oben angegebenen Methoden gibt es noch die folgenden (die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben):

- `map(const Compare& cmp = Compare())`
Konstruktor, der ein Compare-Objekt akzeptieren kann. Falls keines angegeben ist, wird `less<Key>()` genommen.
- `template<class InputIterator>`
`map(InputIterator first, InputIterator last,`
 `const Compare& cmp = Compare())`
Eine Map kann mit Daten initialisiert werden, wenn dem Konstruktor ein Bereich von Schlüssel/Wert-Paaren in Form von Iteratoren übergeben wird. Die Zeitkomplexität ist $O(n \log n)$ mit $n = last - first$.
- `T& operator[] (const key_type& k)`
gibt eine Referenz auf die zum Schlüssel `k` gehörenden Daten zurück. Falls der Schlüssel nicht existiert, wird zu diesem Schlüssel ein Objekt mit dem Konstruktor `T()` für den Wertteil angelegt. Die Zeitkomplexität ist $O(\log n)$.
- `const T& at(const key_type& k) const` und
 `T& at(const key_type& k)`
geben eine Referenz auf die zum Schlüssel `k` gehörenden Daten zurück. Falls der Schlüssel nicht existiert, wird eine `out_of_range`-Exception geworfen. Die Zeitkomplexität ist $O(\log n)$.
- `pair<iterator, bool> emplace(args)`
fügt ein Objekt, das mit den Parametern `args` konstruiert wird, ein (zu `emplace` siehe Seite 768). Zum Rückgabewert siehe `insert()`.
- `pair<iterator, bool> insert(const value_type& x)`
fügt das Schlüssel/Daten-Paar `x` ein, sofern noch kein Element mit dem entsprechenden Schlüssel vorhanden ist. Der Iterator des zurückgegebenen Paares zeigt auf das eingefügte Element bzw. auf das Element mit demselben Schlüssel wie `x`. Der Wahrheitswert zeigt, ob überhaupt ein Einfügen stattgefunden hat – es könnte ja sein, dass der Schlüssel bereits existiert. Die Zeitkomplexität ist $O(\log n)$.
- `void insert(InputIterator i, InputIterator j)`
fügt die Elemente aus dem Iteratorbereich `[i, j)` ein. Die Zeitkomplexität ist $O(n \log n)$ mit $n = j - i$.
- `iterator insert(iterator p, const value_type& x)`
wie `insert(const value_type& x)`, wobei der Iterator `p` ein Hinweis sein soll, wo die Suche zum Einfügen beginnen soll. Der zurückgegebene Iterator zeigt auf das eingefügte Element bzw. auf das mit demselben Schlüssel wie `x`. Die Zeitkomplexität ist praktisch $O(1)$, wenn mit `p` die richtige Stelle getroffen wird, ansonsten $O(\log n)$.
- `void insert(il)` fügt die Elemente aus der Initialisierungsliste `il` ein.
- `size_type erase(const key_type& k)`
alle Elemente mit einem Schlüssel gleich `k` löschen. Es wird die Anzahl N der gelöschten Elemente (hier: 0 oder 1, bei einer `multimap` auch mehr) zurückgegeben. Die Zeitkomplexität ist $O(\log n)$, für eine `multimap`: $O(\log n + N)$.
- `iterator erase(iterator p)`
das Element löschen, auf das der Iterator `p` zeigt. Der zurückgegebene Iterator zeigt anschließend auf das `p` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist $O(\log n)$.

- `iterator erase(iterator p, iterator q)`
alle Elemente im Iteratorbereich `[p, q)` löschen. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist $O((q - p)\log n)$.
- `void clear()`
löscht alle Elemente. Ein Aufruf entspricht `erase(begin(), end())`. Die Zeitkomplexität ist $O(n\log n)$.
- `key_compare key_comp() const`
gibt eine Kopie des Vergleichsobjekts zurück, das zur Konstruktion der Map benutzt 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 zur Konstruktion der map benutzt wurde.
- `const_iterator find(const key_type& k) const` und `iterator find(const key_type& k)`
geben einen Iterator auf ein Element mit dem Schlüssel `k` zurück, falls vorhanden. Andernfalls wird `end()` zurückgegeben. Die Zeitkomplexität ist $O(\log n)$.
- `size_type count(const key_type& k) const`
gibt die Anzahl N der Elemente (hier: 0 oder 1, bei einer `multimap` auch mehr) mit dem Schlüssel `k` zurück. Die Zeitkomplexität ist $O(\log n)$, für eine `multimap`: $O(\log n + N)$.
- `const_iterator lower_bound(const key_type& k) const` und `iterator lower_bound(const key_type& k)`
zeigen auf das erste Element, dessen Schlüssel nicht kleiner als `k` ist. Die Zeitkomplexität ist $O(\log n)$.
- `const_iterator upper_bound(const key_type& k) const` und `iterator upper_bound(const key_type& k)`
geben einen Iterator auf das erste Element zurück, dessen Schlüssel größer als `k` ist. Die Zeitkomplexität ist $O(\log n)$.
- `pair<const_iterator, const_iterator> equal_range(const key_type& k) const` und `pair<iterator, iterator> equal_range(const key_type& k)`
geben ein Paar von Iteratoren zurück, zwischen denen die Schlüssel gleich `k` sind. Die Differenz der Iteratoren kann bei einer `map` nur 0 oder 1 sein. Die Zeitkomplexität ist $O(\log n)$. Bei einer `multimap` (siehe unten) können die Differenz und die Komplexität größer sein.

Das Programm unten zeigt eine Anwendung, in der zu jedem Namen eine Telefonnummer abgespeichert wird. Das Einfügen geschieht unsortiert, die Ausgabe ist wegen der internen Implementierung als Suchbaum automatisch sortiert.

Die Suche mit `operator[]` ist trotz der einfacheren Schreibweise unvorteilhaft, weil bei Eingabe eines nicht-existierenden Namens automatisch ein Eintrag dieses Namens mit der Nummer 0 angelegt wird.

```
cout << aMap[derName] << endl; // nicht empfehlenswert!
```

Aus diesem Grund gibt es die Funktion `at()`. In Analogie zur gleichnamigen Funktion bei anderen Containern wirft `at()` eine Exception, wenn der Eintrag nicht vorhanden ist:

```
try {
    cout << aMap.at(derName) << endl;    // O(logN)
} catch(const exception& e) {
    cout << "Nicht gefunden! Exception:" << e.what() << endl;
}
```

Listing 28.7: Sortierte Abbildung (map)

```
// cppbuch/k28/sortedmap.cpp
#include<map>
#include<string>
#include<iostream>
using namespace std;

// Zwei typedefs zur Abkürzung
typedef map<string, long> MapType; // Vergleichsobjekt ist less<string>()
typedef MapType::value_type ValuePair;

int main() {
    MapType aMap;
    aMap.insert(ValuePair("Thomas", 5192835));
    aMap.insert(ValuePair("Werner", 24439404));
    aMap.insert(ValuePair("Manfred", 535353));
    aMap.insert(ValuePair("Heiko", 635352723));
    aMap.insert(ValuePair("Andreas", 42536347));
    aMap.insert(ValuePair("Karin", 9273539));
    // Das zweite Einfügen von Heiko mit einer anderen Nummer wird
    // nicht ausgeführt, weil der Schlüssel schon existiert.
    aMap.insert(ValuePair("Heiko", 1000000));
    cout << "Ausgabe:\n"; // sortiert
    auto iter = aMap.begin();
    while(iter != aMap.end()) {
        cout << (*iter).first << ':' // Name
              << (*iter).second << endl; // Nummer
        ++iter;
    }
    cout << "Ausgabe der Nummer nach Eingabe des Namens\n"
          << "Name: ";
    string derName;
    cin >> derName;
    cout << "Suche mit Iterator: ";
    iter = aMap.find(derName); // O(log N)
    if(iter != aMap.end()) {
        cout << (*iter).second << endl;
    }
    else {
        cout << "Nicht gefunden!" << endl;
    }
    try {
        cout << "Suche mit at(): " << aMap.at(derName) << endl;
    } catch(const exception& e) {
```

```

    cout << "Nicht gefunden! Exception: " << e.what() << endl;
}
}

```

28.3.2 multimap

Die Klasse `multimap` unterscheidet sich von der Klasse `map` dadurch, dass *mehrfache* Einträge von Elementen mit identischen Schlüsseln möglich sind. Die Datentypen und Methoden entsprechen denen der Klasse `map` mit den folgenden wenigen Unterschieden:

- Es gibt keinen Index-Operator `T& operator[] (const key_type& x)`.
- Die `insert()`-Methoden fügen das Schlüssel/Daten-Paar ein, auch wenn ein Element mit dem entsprechenden Schlüssel schon vorhanden ist.
- Die Methode `emplace()` hat einen anderen Rückgabetypp, die Signatur ist `iterator emplace(args)`. Der Iterator zeigt auf das eingefügte Objekt,

28.3.3 set

Die Klasse `set<Key, Compare>` für Mengen, eingebunden durch den Header `<set>`, entspricht der Klasse `map` von Seite 782, nur dass Schlüssel und Daten zusammenfallen. Das heißt, es werden nur Schlüssel gespeichert. Dabei ist der Schlüssel eindeutig, es kann also keine zwei Datensätze zu demselben Schlüssel geben. `set` ist ein assoziativer Container: Die direkte Angabe des Schlüssels sagt, ob er vorhanden ist oder nicht. Obwohl für den assoziativen Zugriff die Schlüssel nicht sortiert sein müssen, liegen sie in der Klasse `set` sortiert vor. Das ergibt den Vorteil der sortierten Ausgabe zum Beispiel beim Durchwandern vom ersten bis zum letzten Element. Dem steht der Nachteil einer längeren Zugriffszeit im Vergleich zu Datenstrukturen gegenüber, die auf der gestreuten Speicherung basieren (sogenannte Hash-Sets, siehe Abschnitt 28.4.3 weiter unten). Die Deklaration der Klasse ist

```

template<class Key,                // Schlüssel
         class Compare = less<Key> > // Standardvergleich für Sortierung
class set;

```

Falls die Sortierung nicht mit dem `<`-Operator (das heißt `less<Key>`) hergestellt werden soll, kann eine Klasse für Funktionsobjekte angegeben werden:

```

class Vergleich { ... }; // Klasse für Funktionsobjekt
// Funktionsobjekt Vergleich():
set<string, Vergleich> Woertermenge;
Vergleich einVergleich; // Funktionsobjekt
set<string, Vergleich> Woertermenge(einVergleich);

```

Ein `set`-Container stellt unter anderem die folgenden öffentlichen Datentypen und Methoden bereit:

- Datentypen der Tabelle 28.2 (Seite 765)
- Konstruktoren und Methoden der Tabelle 28.3 (Seite 766)
- relationale Operatoren (Seite 767)
- Datentypen/Methoden des Abschnitts 28.1.3 (Reversible Container, Seite 768)

Es gibt weitere öffentliche Datentypen, die in der Tabelle 28.7 aufgelistet sind.

Tabelle 28.7: Zusätzliche Datentypen für `set<Key, Compare>`

Datentyp	Bedeutung
<code>pointer</code>	Zeiger auf Set-Element
<code>const_pointer</code>	dito, aber nur lesend verwendbar
<code>key_type</code>	entspricht Key
<code>value_type</code>	entspricht Key (im Gegensatz zur <code>map</code>)
<code>key_compare</code>	<code>Compare</code>
<code>value_compare</code>	<code>Compare</code> (im Gegensatz zur <code>map</code>)

Zusätzlich zu den oben angegebenen Methoden gibt es die folgenden (die Zeitkomplexität ist $O(1)$, wenn nicht anders angegeben):

- `set(const Compare& cmp = Compare())`
Konstruktor, der ein `Compare`-Objekt akzeptieren kann. Falls kein Typ angegeben ist, wird `less<Key>` genommen.
- `template<class InputIterator>`
`set(InputIterator first, InputIterator last, const Compare& cmp = Compare())`
Ein Set kann mit Daten initialisiert werden, wenn dem Konstruktor ein Bereich in Form von Iteratoren übergeben wird. Die Zeitkomplexität ist $O(n \log n)$ mit $n = last - first$.
- `pair<iterator, bool> emplace(args)`
fügt ein Objekt ein, das mit den Parametern `args` konstruiert wird (siehe Seite 768). Zum Rückgabewert siehe `insert()`.
- `pair<iterator, bool> insert(const value_type& k)`
fügt den Schlüssel `k` ein, sofern er noch nicht vorhanden ist. Der Iterator des zurückgegebenen Paares zeigt auf den eingefügten Schlüssel bzw. auf den schon vorhandenen Schlüssel mit demselben Wert wie `k`. Der Wahrheitswert zeigt, ob überhaupt ein Einfügen stattgefunden hat – es könnte ja sein, dass der Schlüssel bereits existiert. Die Zeitkomplexität ist $O(\log n)$.
- `iterator insert(iterator p, const value_type& k)`
wie `insert(const key_type& k)`, wobei der Iterator `p` ein Hinweis ist, wo die Suche zum Einfügen beginnen soll. Falls `p` die richtige Stelle trifft, ist die Zeitkomplexität $O(1)$, ansonsten $O(\log n)$. Der zurückgegebene Iterator zeigt auf das eingefügte Element bzw. auf das schon vorhandene.
- `void insert(InputIterator i, InputIterator j)`
fügt die Elemente aus dem Iteratorbereich `[i, j)` ein. Mit $N = j - i$ ist die Zeitkomplexität $O(N \log N)$.
- `size_type erase(const key_type& k)`
alle Schlüssel gleich `k` löschen. Es wird die Anzahl N der gelöschten Elemente (hier: 0 oder 1) zurückgegeben. Die Zeitkomplexität ist $O(\log n)$ bzw. $O(\log n + N)$ bei einer `multimap`.
- `iterator erase(iterator p)`
löscht das Element, auf das der Iterator `p` zeigt. Der zurückgegebene Iterator zeigt anschließend auf das `p` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist $O(\log n)$.

- `void erase(iterator p, iterator q)`
alle Elemente im Iteratorbereich `[p, q)` löschen. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist $O((q - p)\log n)$.
- `void clear()`
löscht alle Elemente. Der Aufruf entspricht `erase(begin(), end())`. Die Zeitkomplexität ist $O(n\log n)$.
- `key_compare key_comp() const`
gibt eine Kopie des Vergleichsobjekts zurück, das zur Konstruktion des Sets benutzt wurde.
- `value_compare value_comp() const`
dasselbe wie `key_comp()`. Diese Funktion ist nur der Vollständigkeit halber aufgeführt, weil sie auch in der Klasse `map` vorkommt, dort mit anderer Bedeutung.
- `const_iterator find(const key_type& k) const` und
`iterator find(const key_type& k)`
geben einen Iterator auf den Schlüssel `k` zurück, falls vorhanden. Andernfalls wird `end()` zurückgegeben. Die Zeitkomplexität ist $O(\log n)$.
- `size_type count(const key_type& k) const`
gibt die Anzahl N der Elemente (hier: 0 oder 1, bei einem `multiset` auch mehr) mit dem Schlüssel `k` zurück. Die Zeitkomplexität ist $O(\log n)$, für einen `multiset`: $O(\log n + N)$.
- `const_iterator lower_bound(const key_type& k) const` und
`iterator lower_bound(const key_type& k)`
zeigen auf den ersten Schlüssel, der nicht kleiner als `k` ist. Die Zeitkomplexität ist $O(\log n)$.
- `const_iterator upper_bound(const key_type& k) const` und
`iterator upper_bound(const key_type& k)`
geben einen Iterator auf das erste Element zurück, dessen Wert größer als `k` ist. Die Zeitkomplexität ist $O(\log n)$.
- `pair<const_iterator, const_iterator>`
`equal_range(const key_type& k) const` und
`pair<iterator, iterator> equal_range(const key_type& k)`
geben ein Paar von Iteratoren zurück, zwischen denen die Schlüssel gleich `k` sind. Die Differenz der Iteratoren kann hier nur 0 oder 1 sein. Die Zeitkomplexität ist $O(\log n)$. Bei einem `multiset` (siehe unten) können die Differenz und die Komplexität größer sein.

Das folgende Programm zeigt die Verwendung eines Sets. Die Ausgabe zeigt, dass die Elemente trotz zweifachen `insert()`-Aufrufs nur einmal vorkommen. Anschließend wird ein Element gelöscht, einmal über einen Iterator und einmal über seinen Wert.

Listing 28.8: Set

```
// cppbuch/k28/setm.cpp
#include<set>
#include"<showSequence.h>"
using namespace std;

int main() {
    set<int> einSet; // vordefinierter Vergleich: less<int>()
    for(int i = 0; i < 10; ++i) einSet.insert(i);
    for(int i = 0; i < 10; ++i) einSet.insert(i); // keine Wirkung
    showSequence(einSet); // 0 1 2 3 4 5 6 7 8 9

    cout << "Löschen mit Iterator\n Welches Element? (0..9)" ;
    int i;
    cin >> i;
    auto iter = einSet.find(i);
    if(iter == einSet.end()) {
        cout << i << " nicht gefunden!\n";
    }
    else {
        // count() kann nur 0 oder 1 ergeben.
        cout << "Das Element " << i // 1
            << " existiert " << einSet.count(i) << " mal."
            << endl;
        einSet.erase(iter);
        cout << i << " gelöscht!\n";
        cout << "Das Element " << i // 0
            << " existiert " << einSet.count(i) << " mal."
            << endl;
    }
    showSequence(einSet);

    cout << "Löschen durch Suche nach dem Wert\n"
        "Welches Element? (0..9)" ;
    cin >> i;
    int anzahl = einSet.erase(i);
    if(anzahl == 0) {
        cout << i << " nicht gefunden!\n";
    }
    showSequence(einSet);
}
```

Ein set kann nicht nur mit insert() initialisiert werden, sondern auch durch Angabe eines Iterator-Paars. Alle Elemente des Bereichs werden dabei eingefügt. Das funktioniert auch für ein C-Array. Dubletten werden eliminiert:

```
int tabelle[] = { 1, 2, 2, 4, 9, 13, 1, 0, 5}; // 2 und 1 doppelt
size_t count = sizeof(tabelle)/sizeof(tabelle[0]);
set<int> nochEinSet(tabelle, tabelle + count);
showSequence(nochEinSet); // 0 1 2 4 5 9 13
```

28.3.4 multiset

Die Klasse `multiset` unterscheidet sich von der Klasse `set` dadurch, dass *mehrfache* Einträge identischer Schlüssel möglich sind. Die Datentypen und Methoden entsprechen denen der Klasse `set` mit folgenden Ausnahmen:

- `iterator insert(const key_type& k)`
fügt den Schlüssel `k` ein, auch wenn er schon vorhanden ist. Der zurückgegebene Iterator zeigt auf das eingefügte Element.
- Die Methode `emplace()` hat einen anderen Rückgabetypp, die Signatur ist `iterator emplace(args)`. Der Iterator zeigt auf das eingefügte Objekt.

28.4 Hash-Container

Die Sortierung der assoziativen Container wird manchmal nicht benötigt. Die Reihenfolge der Elemente einer Menge oder Abbildung muss durchaus nicht definiert sein. Der Verzicht auf die Sortierung erlaubt es, aus dem Schlüssel die Adresse eines gesuchten Elements direkt zu berechnen. Zum Beispiel baut ein Compiler eine Symboltabelle auf, auf deren Elemente sehr schnell zugegriffen werden soll. Die Zeitkomplexität des Zugriffs ist $O(1)$, unabhängig von der Anzahl N der Elemente in der Tabelle. Voraussetzung ist, dass die Adresse in konstanter Zeit durch eine einfache Formel berechnet werden kann, ausreichend Speicher zur Verfügung steht und dass die Adressberechnung eine gleichmäßige Verteilung der Elemente im Speicher liefert.

Diese Art der Ablage wird Streuspeicherung genannt. Sie ist immer dann geeignet, wenn die tatsächliche Anzahl zu speichernder Schlüssel klein ist, verglichen mit der Anzahl der möglichen Schlüssel. Ein Compiler kann eine Symboltabelle mit 10000 Einträgen vorsehen; die Anzahl der möglichen Variablennamen mit zum Beispiel nur 10 Zeichen ist sehr viel größer. Wenn wir der Einfachheit halber annehmen, dass nur die 26 Kleinbuchstaben verwendet werden sollen, ergeben sich bereits $26^{10} = \text{ca. } 1,4 \cdot 10^{14}$ Möglichkeiten. Dasselbe Problem stellt sich bei der Speicherung riesiger Matrizen, deren Elemente nur zu einem kleinen Prozentsatz ungleich Null sind.

Die Funktion $h(k)$ zur Transformation des Schlüssels k in die Adresse heißt *Hash-Funktion* (vom englischen *to hash* = hacken, haschieren, durcheinanderbringen usw.), weil alle N Möglichkeiten der Schlüssel auf M Speicherplätze abgebildet werden müssen, indem Informationen zerhackt und verwürfelt werden. Dabei sei M sehr viel kleiner als N , woraus sofort ein Problem resultiert: Zwei verschiedene Schlüssel ergeben möglicherweise dieselbe Adresse. Solchen Kollisionen muss Rechnung getragen werden. Die Funktion $h(k); 0 \leq k < N$ darf nur Werte zwischen 0 und $M - 1$ annehmen. Eine sehr einfache Hash-Funktion für Zahlenschlüssel ist die Modulo-Funktion

$$h(k) = k \bmod M$$

Dabei wird für die Tabellengröße M eine Primzahl gewählt, um eine gleichmäßige Verteilung zu erreichen. Dennoch hängt die Verteilung sehr von der Art und dem Vorkommen der Schlüssel ab, und es ist manchmal schwierig, eine Funktion zu finden, die zu nur we-

nigen Kollisionen führt. Eine Hash-Funktion für Zeichenketten sollte dafür sorgen, dass ähnliche Zeichenketten nicht zu Ballungen in der Streutabelle führen. Am besten ist es, wenn die Belegung anhand von »realen« Daten kontrolliert wird, um die Hash-Funktion vor dem produktiven Einsatz einer Software geeignet anpassen zu können.

Kollisionsbehandlung

Was tun, wenn zwei Schlüssel auf derselben Adresse landen? Der zweite hat das Nachsehen, wenn der Platz schon besetzt ist. Zur Lösung dieses Problems legt ein übliches Verfahren die Schlüssel nicht direkt ab. Vielmehr besteht jeder Eintrag in der Tabelle aus einem Verweis auf eine verkettete Liste oder eine andere geeignete Datenstruktur, in der alle Schlüssel mit demselben Hash-Funktionswert abgelegt werden. So ein Tabelleneintrag wird *Bucket* (deutsch Eimer) genannt. Das Verfahren heißt »Streuspeicherung mit Kollisionsauflösung durch Verketten« und ist in Abbildung 28.1 dargestellt. Sie zeigt die mögliche innere Struktur eines `unordered_map`-Containers.

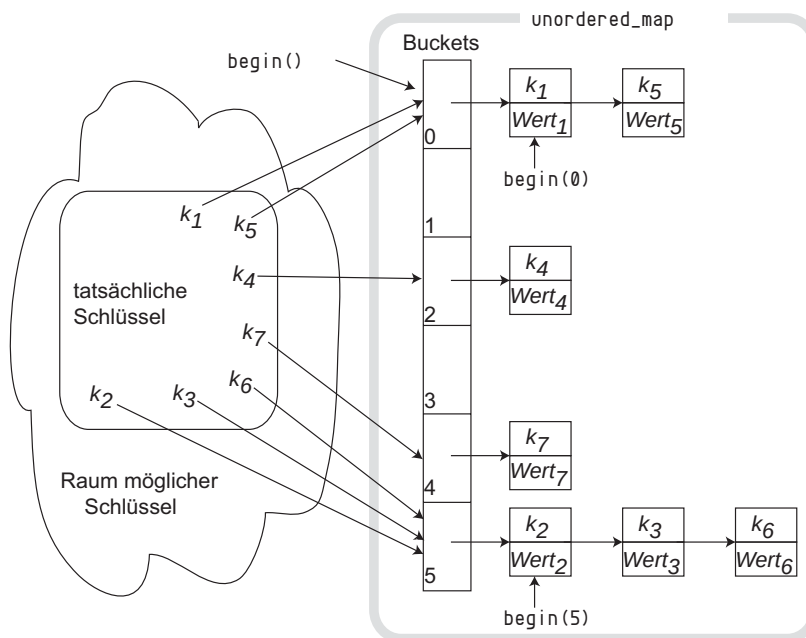


Abbildung 28.1: Streuspeicherung mit Kollisionsauflösung durch Verketten

Ein Tabellenelement `bucket[i]` verweist auf eine Liste aller Schlüssel, deren Hash-Funktionswert $= i$ ist. In Abbildung 28.1 gelten $h(k_1) = h(k_5) = 0$; $h(k_2) = h(k_3) = h(k_6) = 5$; $h(k_4) = 2$ und $h(k_7) = 4$. Der Belegungsgrad (englisch *load factor*) ist das Verhältnis der Anzahl der gespeicherten Elemente zur Anzahl der Buckets. Er sollte erfahrungsgemäß kleiner als 0,7 sein. Das Löschen eines Elements ist einfach, und es können dank der fast beliebigen Länge einer Liste sogar mehr Elemente abgelegt werden, als die Tabelle Positionen hat. Das damit verbundene Anwachsen des Belegungsgrades auf einen Wert

> 1 ist natürlich mit einer Leistungseinbuße verbunden, weil die Such- oder Einfügedauer im schlimmsten Fall proportional zur Länge der längsten Liste ist. Im Fall eines `unordered_set`-Containers sind Schlüssel und Daten dasselbe, sodass die Listenelemente nur Schlüssel enthalten.

Wie üblich, können auch diese Container mithilfe der von `begin()` bzw. `end()` gelieferten Iteratoren durchwandert werden. Um alle Elemente, die auf denselben Hash-Funktionswert abgebildet werden, erreichen zu können, gibt es die beiden Funktionen `begin(size_t n)` bzw. `end(size_t n)`, die einen lokalen Iterator auf die zum Bucket mit der Nummer `n` gehörende Liste zurückgeben (siehe `begin(0)` und `begin(5)` in Abbildung 28.1).

28.4.1 unordered_map

Die Klasse `unordered_map`, eingebunden durch den Header `<unordered_map>`, speichert wie `map` von Seite 782 Paare von Schlüsseln und zugehörigen Daten. Im Gegensatz zur `map` ist die Speicherung nicht sortiert. Die Deklaration der Klasse ist

```
template<class Key,           // Typ der Schlüssel
        class T,             // Typ der Daten
        class Hash = hash<Key>, // Hash-Funktion
        class Pred = std::equal_to<Key> > // für Elementvergleich,
                                           // siehe Seite 753
class unordered_map;
```

`hash<Key>` ist der Typ für ein Funktionsobjekt, das eine Zahl des Typs `size_t` als Hash-Wert zurückgibt. Für alle Grunddatentypen wie `int`, `double`, `char` usw. sowie für Zeiger und `string` liegen Spezialisierungen vor, für andere Typen muss man die Hash-Funktion nach dem folgenden Muster selbst schreiben:

```
// eigene Hash-Funktionsklasse für den Typ meinKey
struct meinHash {
    std::size_t operator()(meinKey wert) const {
        // hier Hash-Wert aus wert berechnen und zurückgeben
    }
}
```

Dieser Typ kann dann bei der Anlage einer `unordered_map` übergeben werden:

```
unordered_map<meinKey, TypDerDaten, meinHash> eineAbbildung;
```

Wenn der Schlüssel selbst mehrere Attribute hat, für die es die vordefinierten Funktionen gibt, kann man sie natürlich nutzen. Zum Beispiel sollen Objekte einer Klasse `Name`, die die Attribute `vorname` und `nachname` hat, als Schlüssel dienen. Der Hash-Wert könnte dann wie folgt für ein Objekt `einName` dieser Klasse berechnet werden:

```
hash<string> hashFunktion; // hash<string> ist vordefiniert.
return hashFunktion(einName.getVorname())
    + hashFunktion(einName.getNachname());
```

Ein `unordered_map`-Container stellt unter anderem die folgenden öffentlichen Datentypen und Methoden bereit:

- Datentypen der Tabelle 28.2 (Seite 765)
- Methoden der Tabelle 28.3 (Seite 766)

Relationale Operatoren sind nicht definiert. Es gibt weitere öffentliche Datentypen, die in der Tabelle 28.8 aufgelistet sind.

Tabelle 28.8: Zusätzliche Datentypen für `unordered_map<Key, T>`

Datentyp	Bedeutung
<code>pointer</code>	Zeiger auf Map-Element
<code>const_pointer</code>	nur lesend verwendbarer Zeiger
<code>key_type</code>	entspricht Key
<code>value_type</code>	entspricht <code>pair<const Key, T></code>
<code>mapped_type</code>	entspricht T
<code>hasher</code>	Hash-Funktionsobjekt
<code>key_equal</code>	Vergleichs-Funktionsobjekt, entspricht <code>Pred</code> in der Deklaration oben
<code>local_iterator</code>	lokaler Iterator für ein Bucket
<code>const_local_iterator</code>	dito (siehe Text)

Die in einer `unordered_map` abgelegten Elemente sind Schlüssel/Wert-Paare. Weil der Schlüssel zur Berechnung der Adresse verwendet wird, kann mit einem Iterator, ob `const` oder nicht, auf das erste Element des Pairs (den Schlüssel) *nur lesend* zugegriffen werden. Andernfalls wäre die Adressberechnung hinfällig. Das zweite Element, z.B. `(*einIterator).second`, ist veränderbar, wenn `einIterator` vom nicht-`const`-Typ `iterator` oder `local_iterator` ist. Das Beispiel unten ist eine Variation des Programms von Seite 786 für eine `unordered_map`.

Der Belegungsfaktor und die Anzahl der Buckets lassen sich mit den entsprechenden Funktionen ermitteln. Mit Hilfe eines `local_iterator`-Objekts können alle Schlüssel mit demselben Hash-Wert angezeigt werden. Das Programmfragment zeigt dies, indem es den Inhalt aller nicht-leeren Buckets anzeigt und nur nach jedem Bucket eine neue Zeile anfängt. Mehrfache Einträge pro Bucket stehen also in einer Zeile:

Listing 28.9: Hash-Map

```
// cppbuch/k28/unorderedmap.cpp
#include<unordered_map>
#include<string>
#include<iostream>
using namespace std;

// Zwei typedefs zur Abkürzung
typedef unordered_map<string, long> MapType;
typedef MapType::value_type ValuePair;

int main() {
    MapType aMap;
    aMap.insert(ValuePair("Thomas", 5192835));
    aMap.insert(ValuePair("Werner", 24439404));
    aMap.insert(ValuePair("Manfred", 535353));
    aMap.insert(ValuePair("Heiko", 635352723));
    aMap.insert(ValuePair("Andreas", 42536347));
    aMap.insert(ValuePair("Karin", 9273539));
    // 2. Einfügen von Heiko mit einer anderen Telefonnummer wird
    // NICHT ausgeführt, weil der Schlüssel schon existiert.
```

```

aMap.insert(ValuePair("Heiko", 1000000));

// Wegen der internen Hash-Struktur ist die Ausgabe unsortiert.
cout << "Ausgabe:\n";
auto iter = aMap.begin();
while(iter != aMap.end()) {
    cout << (*iter).first << ':' // Name
        << (*iter).second << endl; // Nummer
    ++iter;
}
cout << "Ausgabe der Nummer nach Eingabe des Namens\n"
    << "Name: ";
string derName;
cin >> derName;
cout << "Suche mit Iterator: ";
iter = aMap.find(derName); // O(1)
if(iter != aMap.end()) {
    cout << (*iter).second << endl;
}
else {
    cout << "Nicht gefunden!" << endl;
}
cout << "Belegungsfaktor = " << aMap.load_factor() << endl;
cout << "Anzahl der Buckets = " << aMap.bucket_count()
    << ". Belegt sind:" << endl;
for(size_t i=0; i < aMap.bucket_count(); ++i) {
    if(aMap.bucket_size(i) > 0) {
        cout << "Bucket " << i << ": ";
        auto locIter = aMap.begin(i); // auto: MapType::const_local_iterator
        while(locIter != aMap.end(i)) {
            cout << (*locIter).first << ' '
                << (*locIter).second << " ";
            ++locIter;
        }
        cout << endl;
    }
}
}
}

```

Hier ist MapType wie oben definiert. Außer den Methoden der Tabelle 28.3 auf Seite 766 gibt es noch die folgenden – wenn nicht anders angegeben, ist die Zeitkomplexität $O(1)$ und mit $O(n)$ ist $O(\text{size}())$ gemeint:

■ `unordered_map(size_type n = X, const hasher& hf = hasher(),
const key_equal& eql = key_equal())`

Konstruktor. Die vorgegebenen Parameter haben folgende Bedeutung:

- `n` ist die Anzahl der Buckets. Wenn für `n` nichts angegeben wird, ist `X` eine implementationsabhängige Zahl.
- `hf`: Hash-Funktionsobjekt
- `eql`: Funktionsobjekt zum Prüfen zweier Elemente auf Gleichheit.

Wenn der Schlüssel einer der Grunddatentypen, ein Zeiger oder vom Typ `string` ist, ist die einfachste Erzeugung nur durch Angabe von Schlüssel- und Datentyp möglich, wie oben im Beispiel gezeigt.

■ `template<class InputIterator>`

```
unordered_map(InputIterator first, InputIterator last,
size_type n = X, const hasher& hf = hasher(),
const key_equal& eql = key_equal())
```

Eine Hash-Map kann mit Daten initialisiert werden, wenn dem Konstruktor ein Bereich von Schlüssel/Wert-Paaren in Form von Iteratoren übergeben wird. Die Zeitkomplexität ist im Durchschnitt $O(n)$ mit $n = \text{last} - \text{first}$, sie kann im schlechtesten Fall $O(n^2)$ sein. Ursache wäre eine für die reale Schlüsselverteilung gänzlich ungeeignete Hash-Funktion, die alle vorkommenden Schlüssel demselben Bucket zuordnet.

■ `hasher hash_function() const`

gibt eine Kopie des Funktionsobjekts zurück, das zur Hash-Berechnung verwendet wird.

■ `key_equal key_eq() const`

gibt eine Kopie des Funktionsobjekts zurück, das zum Test auf Gleichheit zweier Elemente verwendet wird.

■ `pair<iterator, bool> insert(const value_type& x)`

fügt das Schlüssel/Daten-Paar `x` ein, sofern ein Element mit dem entsprechenden Schlüssel noch nicht vorhanden ist. Der Iterator des zurückgegebenen Paares zeigt auf das eingefügte Element bzw. auf das Element mit demselben Schlüssel wie `x`. Der Wahrheitswert zeigt, ob überhaupt ein Einfügen stattgefunden hat – es könnte ja sein, dass der Schlüssel bereits existiert. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.

■ `iterator insert(iterator p, const value_type& x)`

wie `insert(const value_type& x)`, wobei der Iterator `p` ein Hinweis sein soll, wo die Suche zum Einfügen beginnen soll. Der zurückgegebene Iterator zeigt auf das eingefügte Element bzw. auf das mit demselben Schlüssel wie `x`. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.

■ `void insert(InputIterator i, InputIterator j)`

fügt die Elemente aus dem Iteratorbereich `[i, j)` ein. Die Zeitkomplexität ist im Durchschnitt $O(j - i)$, im schlechtesten Fall $O(n(j - i))$.

■ `size_type erase(const key_type& k)`

alle Elemente mit einem Schlüssel gleich `k` löschen. Es wird die Anzahl `N` der gelöschten Elemente (hier: 0 oder 1, bei einer `unordered_multimap` auch mehr) zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.

■ `iterator erase(iterator p)`

das Element löschen, auf das der Iterator `p` zeigt. Der zurückgegebene Iterator zeigt anschließend auf das `p` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben.

■ `iterator erase(iterator p, iterator q)`

alle Elemente im Iteratorbereich `[p, q)` löschen. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(q - p)$.

- `const_iterator find(const key_type& k) const` und
`iterator find(const key_type& k)`
 geben einen Iterator auf ein Element mit dem Schlüssel `k` zurück, falls vorhanden. Andernfalls wird `end()` zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `size_type count(const key_type& k) const`
 gibt die Anzahl der Elemente (hier: 0 oder 1, bei einer `unordered_multimap` auch mehr) mit dem Schlüssel `k` zurück. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `pair<const_iterator, const_iterator>`
`equal_range(const key_type& k) const` und
`pair<iterator, iterator> equal_range(const key_type& k)`
 geben ein Paar von Iteratoren zurück, zwischen denen die Schlüssel gleich `k` sind. Die Differenz der Iteratoren kann (anders als bei `unordered_multimap`) nur 0 oder 1 sein. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `T& operator[] (const key_type& k)`
 gibt eine Referenz auf die zum Schlüssel `k` gehörenden Daten zurück. Falls der Schlüssel nicht existiert, wird zu diesem Schlüssel ein Objekt mit dem Konstruktor `T()` für den Wertteil angelegt. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `size_type bucket_count() const`
 gibt die aktuelle Anzahl der Buckets zurück.
- `size_type max_bucket_count() const`
 gibt die maximal mögliche Anzahl der Buckets zurück.
- `size_type bucket_size(size_type n) const`
 gibt die Anzahl der im Bucket `n` gespeicherten Elemente zurück. Die Funktion kann hilfreich sein, um eine geeignete Hash-Funktion zu finden.
- `size_type bucket(const key_type& k) const`
 gibt die Nummer des Buckets zurück, in dem die Elemente mit dem Schlüssel `k` gespeichert sind.
- `const_local_iterator begin(size_type n) const` und
`local_iterator begin(size_type n)`
 geben einen Iterator auf den Bucket mit der Nummer `n` zurück.
- `const_local_iterator end(size_type n) const` und
`local_iterator end(size_type n)`
 geben den End-Iterator des Buckets mit der Nummer `n` zurück.
- `float load_factor() const`
 gibt den Belegungsfaktor zurück.
- `void max_load_factor(float f) const`
 setzt den maximal gewünschten Belegungsfaktor. Falls dieser Faktor überschritten zu werden droht, wird die Anzahl der Buckets erhöht.
- `float max_load_factor() const`
 gibt den maximal gewünschten Belegungsfaktor zurück.

- `void rehash(size_type n)`
reorganisiert den Container, sodass er danach mindestens n Buckets hat und der Belegungsfaktor unter dem maximal gewünschten liegt. Die Zeitkomplexität ist im Durchschnitt $O(n)$, im schlechtesten Fall $O(n^2)$.

28.4.2 unordered_multimap

Die Klasse `unordered_multimap` unterscheidet sich von der Klasse `unordered_map` dadurch, dass *mehrfache* Einträge von Elementen mit identischen Schlüsseln möglich sind. Die Datentypen und Methoden entsprechen denen der oben dargestellten Klasse `unordered_map` mit folgenden wenigen Unterschieden:

- Es gibt keinen Index-Operator `T& operator[](const key_type& x)`.
- Die `insert()`-Methoden fügen Schlüssel/Daten-Paare ein, auch wenn ein Element mit dem entsprechenden Schlüssel schon vorhanden ist.
- Die Zeitkomplexität mancher Methoden (wie zum Beispiel `equal_range(const key_type& k)`) erhöht sich im Vergleich zu einer `unordered_map`, wenn es mehrere Einträge mit identischen Schlüsseln gibt.

28.4.3 unordered_set

Die Klasse `unordered_set` für Mengen entspricht der Klasse `unordered_map`, nur dass Schlüssel und Daten zusammenfallen. Das heißt, es werden nur Schlüssel gespeichert. Dabei ist der Schlüssel eindeutig, es kann also keine zwei Datensätze zu demselben Schlüssel geben. Im Gegensatz zu `set` ist die Speicherung nicht sortiert. Die Deklaration ist

```
template<class Value,                // Typ der Schlüssel
        class Hash = hash<Key>,      // Hash-Funktion
        class Pred = std::equal_to<Key> > // für Elementvergleich
        // siehe Seite 753
class unordered_set;
```

`hash<Key>` ist ein Typ für ein Funktionsobjekt, das eine Zahl des Typs `size_t` als Hash-Wert zurückgibt, vgl. Seite 793. Ein `unordered_set`-Container stellt unter anderem die folgenden öffentlichen Datentypen und Methoden bereit:

- Datentypen der Tabelle 28.2 (Seite 765)
- Methoden der Tabelle 28.3 (Seite 766)

Relationale Operatoren sind nicht definiert. Es gibt weitere öffentliche Datentypen, die in der Tabelle 28.9 aufgelistet sind. Außer den Methoden der Tabelle 28.3 gibt es noch die folgenden – wenn nicht anders angegeben, ist die Zeitkomplexität $O(1)$ und mit $O(n)$ ist $O(\text{size}())$ gemeint:

- `unordered_set(size_type n = X, const hasher& hf = hasher(),
const key_equal& eql = key_equal())`

Konstruktor. Die vorgegebenen Parameter haben die auf Seite 795 angegebene Bedeutung.

- `template<class InputIterator>
unordered_set(InputIterator first, InputIterator last,
size_type n = X, const hasher& hf = hasher(),
const key_equal& eql = key_equal())`

Tabelle 28.9: Zusätzliche Datentypen für `unordered_set<Key>`

Datentyp	Bedeutung
<code>pointer</code>	Zeiger auf Set-Element
<code>const_pointer</code>	nur lesend verwendbarer Zeiger
<code>key_type</code>	entspricht <code>Value</code>
<code>value_type</code>	dito
<code>hasher</code>	Hash-Funktionsobjekt
<code>key_equal</code>	Vergleichs-Funktionsobjekt, entspricht <code>Pred</code> in der Deklaration oben
<code>local_iterator</code>	lokaler Iterator für ein Bucket
<code>const_local_iterator</code>	dito (vgl. Seite 794)

Ein Hash-Set kann mit Daten initialisiert werden, wenn dem Konstruktor ein Bereich von Schlüsseln in Form von Iteratoren übergeben wird. Die Zeitkomplexität ist im Durchschnitt $O(n)$ mit $n = last - first$, sie kann wie bei der `unordered_map` im schlechtesten Fall $O(n^2)$ sein.

- `hasher hash_function() const`
gibt eine Kopie des Funktionsobjekts zurück, das zur Hash-Berechnung verwendet wird.
- `key_equal key_eq() const`
gibt eine Kopie des Funktionsobjekts zurück, das zum Test auf Gleichheit zweier Schlüssel verwendet wird.
- `pair<iterator, bool> insert(const value_type& x)`
fügt den Schlüssel `x` ein, sofern er noch nicht vorhanden ist. Der Iterator des zurückgegebenen Paares zeigt auf das eingefügte Element bzw. auf den ggf. schon vorhandenen Schlüssel. Der Wahrheitswert zeigt, ob überhaupt ein Einfügen stattgefunden hat – es könnte ja sein, dass der Schlüssel bereits existiert. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `iterator insert(iterator p, const value_type& x)`
wie `insert(const value_type& x)`, wobei der Iterator `p` ein Hinweis ist, wo die Suche zum Einfügen beginnen soll. Der zurückgegebene Iterator zeigt auf das eingefügte Element bzw. auf den schon vorhandenen Schlüssel. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `void insert(InputIterator i, InputIterator j)`
fügt die Schlüssel aus dem Iteratorbereich `[i, j)` ein. Die Zeitkomplexität ist im Durchschnitt $O(j - i)$, im schlechtesten Fall $O(n(j - i))$.
- `size_type erase(const key_type& k)`
alle Elemente mit einem Schlüssel gleich `k` löschen. Es wird die Anzahl `N` der gelöschten Elemente (hier: 0 oder 1, bei einem `unordered_multiset` auch mehr) zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `iterator erase(iterator p)`
das Element löschen, auf das der Iterator `p` zeigt. Der zurückgegebene Iterator zeigt anschließend auf das `p` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben.

- `iterator erase(iterator p, iterator q)`
alle Elemente im Iteratorbereich `[p, q)` löschen. Der zurückgegebene Iterator zeigt anschließend auf das `q` folgende Element, wenn es existiert, ansonsten wird ein End-Iterator (`end()`) zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(q - p)$.
- `const_iterator find(const key_type& k) const` und `iterator find(const key_type& k)`
geben einen Iterator auf den Schlüssel `k` zurück, falls vorhanden. Andernfalls wird `end()` zurückgegeben. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `size_type count(const key_type& k) const`
gibt die Anzahl der Schlüssel (hier: 0 oder 1, bei einem `unordered_multiset` auch mehr) mit dem Wert `k` zurück. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `pair<const_iterator, const_iterator> equal_range(const key_type& k) const` und `pair<iterator, iterator> equal_range(const key_type& k)`
geben ein Paar von Iteratoren zurück, zwischen denen die Schlüssel gleich `k` sind. Die Differenz der Iteratoren kann (anders als bei `unordered_multiset`) nur 0 oder 1 sein. Die Zeitkomplexität ist im Durchschnitt $O(1)$, im schlechtesten Fall $O(n)$.
- `size_type bucket_count() const`
gibt die aktuelle Anzahl der Buckets zurück.
- `size_type max_bucket_count() const`
gibt die maximal mögliche Anzahl der Buckets zurück.
- `size_type bucket_size(size_type n) const`
gibt die Anzahl der im Bucket `n` gespeicherten Elemente zurück.
- `size_type bucket(const key_type& k) const`
gibt die Nummer des Buckets zurück, in dem die Elemente mit dem Schlüssel `k` gespeichert sind.
- `const_local_iterator begin(size_type n) const` und `local_iterator begin(size_type n)`
geben einen Iterator auf den Bucket mit der Nummer `n` zurück.
- `const_local_iterator end(size_type n) const` und `local_iterator end(size_type n)`
geben den End-Iterator des Buckets mit der Nummer `n` zurück.
- `float load_factor() const`
gibt den Belegungsfaktor zurück.
- `void max_load_factor(float f) const`
setzt den maximal gewünschten Belegungsfaktor. Falls dieser Faktor überschritten zu werden droht, wird die Anzahl der Buckets erhöht.
- `float max_load_factor() const`
gibt den maximal gewünschten Belegungsfaktor zurück.

- `void rehash(size_type n)`
reorganisiert den Container, sodass er danach mindestens `n` Buckets hat und der Belegungsfaktor unter dem maximal gewünschten liegt. Die Zeitkomplexität ist im Durchschnitt $O(n)$, im schlechtesten Fall $O(n^2)$.

28.4.4 unordered_multiset

Die Klasse `unordered_multiset` unterscheidet sich von der Klasse `unordered_set` dadurch, dass *mehrfache* Einträge identischer Schlüssel möglich sind. Die Datentypen und Methoden entsprechen denen der Klasse `set` mit folgender Ausnahme:

- Die `insert()`-Methoden fügen Schlüssel/Daten-Paare ein, auch wenn ein Element mit dem entsprechenden Schlüssel schon vorhanden ist.

28.5 bitset

Der Header `<bitset>` definiert eine Template-Klasse und zugehörige Funktionen zur Darstellung und Bearbeitung von Bitfolgen fester Größe. Die Deklaration der Klasse ist

```
template<size_t N> class bitset;
```

Wie bei dem Beispiel eines Stacks mit statisch festgelegter Größe in Abschnitt 6.3.2 ab Seite 249 ist `N` die Anzahl der zu speichernden Bits. Es gibt die folgenden `std`-globalen Operatoren, die den Bitoperationen mit `int`-Zahlen entsprechen:

```
template <size_t N>
    bitset<N> operator&(const bitset<N>&, const bitset<N>&);
template <size_t N>
    bitset<N> operator|(const bitset<N>&, const bitset<N>&);
template <size_t N>
    bitset<N> operator^(const bitset<N>&, const bitset<N>&);
```

Dazu kommen Operatoren zur Ein- und Ausgabe:

```
template <size_t N>
    istream& operator>>(istream& is, bitset<N>& x);
template <size_t N>
    ostream& operator<<(ostream& os, const bitset<N>& x);
```

Die beiden letzten Deklarationen sorgen dafür, dass die Eingabe und die Ausgabe mit Streams auf die bekannte einfache Art möglich sind, etwa:

```
bitset<13817> B; // 13817 Bits
cin >> B;       // Eingabe
cout << B;      // Ausgabe
```

Die Klasse `bitset` stellt den öffentlichen Datentyp `reference` für Manipulationen an einzelnen Bits bereit, ähnlich wie die Klasse `vector<bool>`:

```

class reference {
    friend class bitset;
    reference();
public:
    ~reference();
    reference& operator=(bool x);          // für b[i] = x;
    reference& operator=(const reference&); // für b[i] = b[j];
    bool operator~() const;                // negiert das Bit
    operator bool() const;                 // für x = b[i];
    reference& flip();                      // für b[i].flip();
};

```

Die Klasse `bitset` kennt die folgenden Konstruktoren und Methoden:

- `bitset()` ist der Standardkonstruktor. Alle Bits werden zu 0 initialisiert.
- `bitset(unsigned long val)`
Dieser Konstruktor initialisiert die ersten m Positionen entsprechend der Bitwerte im Parameter `val`. m ist $8 \cdot \text{sizeof}(\text{unsigned long})$, falls ein Byte 8 Bits entspricht. Falls $m < N$ ist, werden die restlichen Bits zu 0 initialisiert. Falls $m > N$ ist, werden nur N Positionen initialisiert.
- `bitset(const string& str, size_t pos = 0, size_t n = string::npos)`
Der Bitset wird mit dem String `str`, beginnend an der Position `pos`, initialisiert. Der String darf nur aus den Zeichen '0' und '1' bestehen. Der Parameter `n` bestimmt die maximale Anzahl der auszuwertenden Zeichen. Auch hier werden restliche Positionen zu 0 gesetzt, und es werden insgesamt nicht mehr als N Positionen initialisiert.
- `bitset<N>& operator&=(const bitset<N>& B)`
löscht jedes Bit, dessen Entsprechung im Bitset `B` nicht gesetzt ist. Die anderen Bits bleiben unverändert. Es wird das aufrufende Objekt zurückgegeben (`*this`).
- `bitset<N>& operator|=(const bitset<N>& B)`
setzt jedes Bit, dessen Entsprechung im Bitset `B` gesetzt ist. Die anderen Bits bleiben unverändert. Es wird `*this` zurückgegeben.
- `bitset<N>& operator^=(const bitset<N>& B)`
negiert jedes Bit, dessen Entsprechung im Bitset `B` gesetzt ist. Die anderen Bits bleiben unverändert. Es wird `*this` zurückgegeben.
- `bitset<N>& operator<<=(size_t n)`
verschiebt alle Bits um n Positionen nach links. Dabei werden von rechts Null-Bits nachgezogen. Es wird `*this` zurückgegeben.
- `bitset<N> operator<<(size_t n) const`
gibt einen Bitset zurück, in dem relativ zum aufrufenden Objekt alle Bits um n Positionen nach links verschoben sind. Im zurückgegebenen Ergebnis sind n Bits rechts auf Null gesetzt.
- `bitset<N>& operator>>=(size_t n)`
verschiebt alle Bits um n Positionen nach rechts. Dabei werden von rechts Null-Bits nachgezogen. Es wird `*this` zurückgegeben.

- `bitset<N> operator>>(size_t n) const`
gibt einen Bitset zurück, in dem relativ zum aufrufenden Objekt alle Bits um `n` Positionen nach rechts verschoben sind. Im zurückgegebenen Ergebnis sind `n` Bits links auf Null gesetzt.
- `bitset<N>& set()` setzt alle Bits. Es wird `*this` zurückgegeben.
- `bitset<N>& set(size_t n, int val = 1)`
setzt das Bit an der Position `n`, falls `val` nicht angegeben wird oder ungleich Null ist. Ist `val` gleich 0, wird das Bit auf 0 zurückgesetzt. Es wird `*this` zurückgegeben.
- `bitset<N>& reset()` löscht alle Bits. Es wird `*this` zurückgegeben.
- `bitset<N>& reset(size_t n)`
löscht das Bit an der Position `n`. Es wird `*this` zurückgegeben.
- `bitset<N>& flip()` negiert alle Bits. Es wird `*this` zurückgegeben.
- `bitset<N>& flip(size_t n)`
negiert das Bit an der Position `n`. Es wird `*this` zurückgegeben.
- `bitset<N> operator~() const`
gibt eine Kopie des aufrufenden Objekts zurück, in der alle Bits negiert sind.
- `reference operator[](size_t n)`
gibt das Bit Nr. `n` als Objekt der Klasse `bitset<N>::reference` zurück (siehe Seite 801).
- `unsigned long to_ulong() const`
gibt das aufrufende `bitset`-Objekt als `unsigned long`-Zahl zurück. Dabei darf `N` nicht größer als die Anzahl der Bits sein, die eine `unsigned long`-Zahl aufnehmen kann.
- `string to_string() const`
gibt das aufrufende `bitset`-Objekt als String, bestehend aus Nullen und Einsen, zurück. Das erste Zeichen des Strings (Position 0) entspricht dem höchstwertigen Bit.
- `size_t count() const` gibt die Anzahl der gesetzten Bits zurück.
- `size_t size() const` gibt die Anzahl aller Bits (d.h. `N`) zurück.
- `bool operator==(const bitset<N>& B) const`
gibt zurück, ob alle Bits mit denen von `B` übereinstimmen.
- `bool operator!=(const bitset<N>& B) const`
gibt zurück, ob wenigstens ein Bit mit denen von `B` nicht übereinstimmt.
- `bool test(size_t n) const` gibt zurück, ob das Bit `n` gesetzt ist.
- `bool any() const` gibt zurück, ob wenigstens ein Bit gesetzt ist.
- `bool none() const` gibt zurück, ob alle Bits 0 sind.

Das folgende kleine Programm gibt `11111111101111` und `00001111111110` aus:

Listing 28.10: Bitset

```
// cppbuch/k28/bitset.cpp
#include<bitset>
#include<iostream>
using namespace std;
int main() {
    bitset<15> einBitset;
    einBitset.set();           // alles 1
    einBitset[4].flip();       // Bit 4 wird 0
    cout << einBitset << endl;
```



```
einBitset >>= 4; // Rechtsverschiebung um 4 Positionen
cout << einBitset << endl;
}
```



Übungen

28.1 Wenn eine Funktion aufgerufen wird, werden die lokalen Variablen auf dem C++-Laufzeitstack abgelegt. Nach Rückkehr der Funktion werden die Variablen wieder vom Stack geholt, und damit wird die Umgebung des Aufrufers wiederhergestellt. Daraus ergibt sich, dass jeder Funktionsaufruf auch mithilfe eines eigenen Stacks simuliert werden kann: 1. lokale Variablen auf dem Stack sichern; 2. Code mit Variablen der Funktion ausführen; 3. lokale Variablen restaurieren. Also lassen sich auch jegliche rekursive Aufrufe einer Funktion simulieren und damit ersetzen. Aufgabe: Eliminieren Sie die noch verbliebene Rekursion in der Lösung zu Aufgabe 3.2 von Seite 111, indem Sie ein Objekt der Klasse `stack` zur Verwaltung der aktuellen Variablen verwenden.

28.2 Tragen Sie Prominente als Rang/Name-Kombination in eine Priority-Queue `promis` ein, etwa

```
promis.push(make_pair(8, "Peter Jackson")); // oder
promis.push(pair<int, string>(10, "Tina Turner")); // ... usw.
```

Zu `make_pair()` und `pair` siehe Abschnitt 27.3. Die Zahl soll den Rang oder die vermutete Wichtigkeit des jeweiligen Stars in den einschlägigen Illustrierten bedeuten: je größer, desto wichtiger. Leeren Sie die Queue und zeigen Sie dabei die Prominenten auf dem Bildschirm an, geordnet nach ihrem Rang, und die wichtigsten zuerst.

28.3 Wie muss die Deklaration der Priority-Queue `promis` lauten, wenn zuerst die weniger wichtigen Promis ausgegeben werden sollen?

28.4 Lösen Sie das Problem der beiden vorstehenden Aufgaben, indem Sie anstelle einer Priority-Queue ein Objekt der Klasse `multimap` verwenden.

29

Iteratoren

Dieses Kapitel behandelt die folgenden Themen:

- Iterator-Kategorien
- Typinformation mit Traits
- Abstand von Iteratoren
- Iteratoren zum Einfügen
- Stream-Iteratoren

Iteratoren werden in Abschnitt [11.2](#) besprochen und an Beispielen gezeigt. Hier geht es um die vordefinierten Iteratortypen im Header `<iterator>` der C++-Standardbibliothek, die wie die Standard-Container als Templates realisiert sind und über traits-Klassen (*traits*, dt. etwa Eigenschaft) bestimmte öffentliche Typnamen zur Verfügung stellen. Natürlich könnten Typnamen auch direkt von einer Iteratorklasse veröffentlicht werden, aber man geht einen anderen Weg, weil die Algorithmen der C++-Standardbibliothek nicht nur auf Containern, die Typnamen bereitstellen, sondern auch auf einfachen C-Arrays arbeiten können sollen. Die damit arbeitenden Iteratoren sind aber nichts anderes als Zeiger, möglicherweise auf Grunddatentypen wie `int`. Ein Iterator des Typs `int*` kann sicher keine Typnamen zur Verfügung stellen. Aus diesem Grund gibt es eine Spezialisierung der traits-Klassen speziell für Zeiger.

```
// vom Iterator abgeleitete öffentliche Typen:  
template<class Iterator>  
struct iterator_traits {
```

```
typedef typename Iterator::difference_type difference_type;
typedef typename Iterator::value_type value_type;
typedef typename Iterator::pointer pointer;
typedef typename Iterator::reference reference;
typedef typename Iterator::iterator_category iterator_category;
};
```

Damit ein Algorithmus, der mit Zeigern arbeitet, die üblichen Typnamen verwenden kann, wird das obige Template für Zeiger spezialisiert:

```
// partielle Spezialisierung (für Zeiger)
template<class T>
struct iterator_traits<T*> {
    typedef ptrdiff_t difference_type;
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef random_access_iterator_tag iterator_category;
};
```

Die Iterator-Kategorie wird im nächsten Abschnitt erläutert, auch konkrete Anwendungsbeispiele für Traits sind dort zu finden. In der C++-Standardbibliothek wird ein Datentyp für Iteratoren angegeben, von dem jeder benutzerdefinierte Iterator erben kann:

```
namespace std {
    template<class Category, class T, class Distance = ptrdiff_t,
            class Pointer = T*, class Reference = T&>
    struct iterator {
        typedef Distance difference_type;
        typedef T value_type;
        typedef Pointer pointer;
        typedef Reference reference;
        typedef Category iterator_category; // siehe Abschnitt 29.1
    };
}
```

Durch public-Vererbung sind diese Namen in allen abgeleiteten Klassen sicht- und verwendbar.

29.1 Iterator-Kategorien

Es gibt verschiedene Kategorien von Iteratoren in einer hierarchischen Anordnung.

- **Input-Iterator:** Ein Input-Iterator ist zum sequenziellen Lesen von Daten gedacht, zum Beispiel aus einem Container oder aus einer Datei. Ein Zurückspringen an eine schon gelesene Stelle ist nicht möglich (der `---`Operator ist nicht definiert).
- **Output-Iterator:** Ein Output-Iterator kann sequenziell in einen Container oder in eine Datei schreiben, wobei der Dereferenzierungsoperator verwendet wird. Beispiel:

```
// »Ausgabe« ist ein Output-Iterator
*Ausgabe++ = Wert; // in die Ausgabe schreiben und weiterschalten
```

- **Forward-Iterator:** Wie Input- und Output-Iterator kann der Forward-Iterator sich vorwärts bewegen. Im Unterschied zu den vorgenannten Iteratoren können jedoch Werte des Iterators gespeichert werden, um ein Element des Containers wiederzufinden. Damit ist ein mehrfacher Durchlauf in eine Richtung möglich, zum Beispiel durch eine einfach verkettete Liste, wenn man sich den Anfang gemerkt hat.
- **Bidirectional-Iterator:** Ein Bidirectional-Iterator kann all das, was ein Forward-Iterator kann. Darüber hinaus kann er noch mit dem `--`-Operator *rückwärts* gehen, sodass er zum Beispiel für eine doppelt verkettete Liste geeignet ist.
- **Random-Access-Iterator:** Ein Random-Access-Iterator kann alles, was ein Bidirectional-Iterator kann. Zusätzlich ist ein wahlfreier Zugriff möglich, wie er für einen Vektor benötigt wird. Der wahlfreie Zugriff wird durch den Indexoperator `operator[]()` realisiert.

Tabelle 29.1 zeigt eine Übersicht über mögliche Operationen einer Kategorie.

Tabelle 29.1: Fähigkeiten der Iterator-Kategorien

Operation	Input	Output	Forward	Bidirectional	Random Access
=	•		•	•	•
==	•		•	•	•
!=	•		•	•	•
*	1)	2)	•	•	•
->	•		•	•	•
++	•	•	•	•	•
--				•	•
[]					3)
arithmetisch					4)
relational					5)

1) Dereferenzierung ist nur lesend möglich.

2) Dereferenzierung ist nur auf der linken Seite einer Zuweisung möglich.

3) `iter[n]` bedeutet `*(iter+n)` für einen Iterator `iter`

4) `+` `+=` `-` `--` in Analogie zur Zeigerarithmetik

5) `<` `>` `<=` `>=` relationale Operatoren

Um einen Iterator mit einer Marke (englisch *tag*) zu versehen, gibt es die folgenden Markierungsklassen:

```
struct input_iterator_tag {};
```

```
struct output_iterator_tag {};
```

```
struct forward_iterator_tag
: public input_iterator_tag {};
```

```
struct bidirectional_iterator_tag
: public forward_iterator_tag {};
```

```
struct random_access_iterator_tag
: public bidirectional_iterator_tag {};
```

29.1.1 Anwendung von Traits

In diesem Abschnitt wird gezeigt, wie mithilfe von Traits der Typ bestimmt und wie abhängig vom Typ der passende Algorithmus ausgewählt werden kann. Der Compiler wählt im folgenden Beispiel dazu unter überladenen Funktionen aus. Deren Parameter, die Iterator-Kategorie, wird aus dem Iterator mithilfe der Funktion `getIteratortyp()` ermittelt:

Listing 29.1: Iteratortyp bestimmen

```
// cppbuch/k29/ityp.cpp
#include<string>
#include<fstream>
#include<vector>
#include<iterator>
#include<iostream>
using namespace std;

// Funktions-Templates zur Ermittlung des Typs (iterator-tag) des Iterators
template<class Iterator>
typename iterator_traits<Iterator>::iterator_category
getIteratortyp(const Iterator&) {
    typename iterator_traits<Iterator>::iterator_category
        typeobject;
    return typeobject;
}

// überladene Funktionen
void welcherIterator(const input_iterator_tag&) {
    cout << "Input-Iterator!" << endl;
}

void welcherIterator(const output_iterator_tag&) {
    cout << "Output-Iterator!" << endl;
}

void welcherIterator(const forward_iterator_tag&) {
    cout << "Forward-Iterator!" << endl;
}

void welcherIterator(const random_access_iterator_tag&) {
    cout << "Random-Access-Iterator!" << endl;
}

int main( ) {    // Anwendung
    // Bei Grunddatentypen (hier: ein Zeiger) wird das partiell spezialisierte
    // iterator_traits<T*>- Template von Seite 806 benutzt.
    int *ip;           // Random-Access-Iterator
    // Anzeige des Iteratortyps
    welcherIterator(getIteratortyp(ip)); // oder:
    welcherIterator(iterator_traits<int*>::iterator_category());

    // Definition eines Datei-Objekts zum Lesen (eine tatsächliche Datei ist nicht
    // erforderlich, es geht nur um den Typ)
```

```

    ifstream Source;
    // Ein istream_iterator ist ein Input-Iterator
    istream_iterator<string> IPos(Source);
    // Anzeige des Iteratortyps
    welcherIterator(getIteratortyp(IPos)); // oder:
    welcherIterator(iterator_traits<istream_iterator<string>>
        ::iterator_category());

    // Definition eines Datei-Objekts zum Schreiben
    ofstream Destination;
    // Ein ostream_iterator ist ein Output-Iterator
    ostream_iterator<string> OPos(Destination);
    // Anzeige des Iteratortyps
    welcherIterator(getIteratortyp(OPos)); // oder:
    welcherIterator(iterator_traits<ostream_iterator<string>>
        ::iterator_category());

    vector<int> v(10);
    // Anzeige des Iteratortyps
    welcherIterator(getIteratortyp(v.begin())); // oder end()
    welcherIterator(iterator_traits<vector<int>::iterator>
        ::iterator_category());
}

```

Im zweiten Beispiel wird der am besten passende Algorithmus automatisch zur Compilierzeit ausgewählt. Es soll das mittlere Element eines Containers zurückgegeben werden. Bei einer Liste müssen dazu $N/2$ Elemente abgeklappert werden, es ist also eine Schleife notwendig. Bei einem Vektor nimmt man einfach das Element $[N/2]$. Dabei wird einer Funktion `mittleresElement(Iterator anfang, size_t n)` der Anfang des Containers als Iterator und die Anzahl der Elemente mitgeteilt. Aus dem Typ des Iterators wird die passende überladene aufzurufende Funktion ermittelt:

Listing 29.2: Algorithmus typabhängig auswählen

```

// cppbuch/k29/algorithmenwahl.cpp
#include<iostream>
#include<list>
#include<vector>
#include<iterator>

template<class Iterator> // aufrufende Funktion
int mittleresElement(Iterator anfang, size_t n) {
    typename std::iterator_traits<Iterator>::iterator_category
        typeobject;
    return holeMittleres(anfang, n, typeobject);
}

template<class Iterator> // erste überladene Funktion
int holeMittleres(Iterator anfang, size_t n,
    std::bidirectional_iterator_tag) {
    for(size_t i=0; i < n/2; ++i) { // Schleife
        ++anfang;
    }
}

```

```

    return *anfang;
}

template<class Iterator> // zweite überladene Funktion
int holeMittleres(Iterator anfang, size_t n,
                  std::random_access_iterator_tag) {
    return *(anfang + n/2); // Arithmetik
}

using namespace std;
int main() { // Hauptprogramm
    list<int> lis; // mit Werten füllen
    for(size_t i=0; i < 10; ++i) {
        lis.push_back(i);
    }
    vector<int> vec(10); // mit Werten füllen
    for(size_t i = 0; i < vec.size(); ++i) {
        vec[i] = 10*i;
    }
    // Aufruf der ersten Implementierung für die Liste
    cout << mittleresElement(lis.begin(), lis.size()) << endl;
    // Aufruf der zweiten Implementierung für den Vektor
    cout << mittleresElement(vec.begin(), vec.size()) << endl;
}

```

29.2 distance() und advance()

Weil nur Random-Access-Iteratoren die Operationen + und - erlauben, gibt es die Funktionen distance() zum Ermitteln eines Iteratorabstands und advance() zum Weiterschalten. Diese Funktionen benutzen intern + und - für Random-Access-Iteratoren und in anderen Fällen ++ bzw. --. Die Deklarationen sind:

```

// advance() schaltet i um n Positionen vor bzw. zurück, falls n < 0 ist.
template<class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);

// distance(first, last) 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);

```

29.3 Reverse-Iteratoren

Ein Reverse-Iterator ist bei einem bidirektionalen Iterator immer möglich. Ein Reverse-Iterator durchläuft einen Container *rückwärts* mit der `++`-Operation. Beginn und Ende eines Containers für Reverse-Iteratoren werden durch `rbegin()` und `rend()` markiert. Dabei verweist `rbegin()` auf das letzte Element des Containers und `rend()` auf die (ggf. fiktive) Position *vor* dem ersten Element. Einige Container stellen Reverse-Iteratoren zur Verfügung. Sie werden mit der vordefinierten Klasse

```
template<class Iterator> class reverse_iterator;
```

realisiert. Ein Objekt dieser Klasse wird mit einem bidirektionalen oder einem Random-Access-Iterator initialisiert, entsprechend dem Typ des Template-Parameters. Ein Reverse-Iterator arbeitet intern mit diesem Iterator und legt eine Hülle (englisch *wrapper*) mit bestimmten zusätzlichen Operationen um ihn herum. Für einen existierenden Iterator wird eine neue Schnittstelle geschaffen, um sich verschiedenen Gegebenheiten anpassen (englisch *to adapt*) zu können. Deshalb werden Klassen, die eine Klasse in eine andere umwandeln, *Adapter* genannt. Ein Beispiel sehen Sie unten. Zu den in der Tabelle 29.1 auf Seite 807 angegebenen Operationen für bidirektionale oder Random-Access-Iteratoren gibt es die Methode `base()`, die den gekapselten Iterator zurückgibt.

Listing 29.3: Reverse-Iterator

```
// cppbuch/k29/reverse.cpp
#include<vector>
#include<iostream>
#include<iterator>
using namespace std;

int main() {
    vector<int> v(10);
    for(size_t i = 0; i < v.size(); ++i) {
        v[i] = i;
    }
    reverse_iterator<vector<int>::iterator> revIter(v.rbegin());
    // Alternativ: vector<int>::reverse_iterator revIter(v.rbegin());
    while(revIter != v.rend()) { // Zahlen verdoppelt in umgekehrter Folge ausgeben
        *revIter *= 2;           // Wert über den Iterator ändern
        cout << *revIter++ << ' '; // nur lesender Zugriff
    }
    cout << endl;
    // Falls Werte NICHT geändert werden sollen, kann der von der Klasse vector
    // bereitgestellte Typ const_reverse_iterator verwendet werden. Eine eigene
    // Klasse const_reverse_iterator gibt es nicht [ISOC++].
    vector<int>::const_reverse_iterator constRevIter = v.rbegin(),
                                                    constRevIterEnd(v.rend());
    while(constRevIter != constRevIterEnd) {
        cout << *constRevIter++ << ' ';
    }
}
```


29.4 Insert-Iteratoren

Mit normalen Iteratoren bewirkt der Code

```
while(first != last) {
    *result++ = *first++;
}
```

das Kopieren des Bereichs `[first, last)` an die Stelle `result`, wobei der vorherige Inhalt an der Stelle *überschrieben* wird. Derselbe Code bewirkt jedoch das *Einfügen* in einen Container, wenn `result` ein Insert-Iterator ist. Je nach gewünschter Position zum Einfügen gibt es drei Varianten:

1. `front_insert_iterator` (Beispiel siehe *cppbuch/k29/finsert.cpp*)

Dieser Insert-Iterator fügt etwas am Anfang eines Containers ein. Der Container muss die Methode `push_front()` zur Verfügung stellen. Anwendungsbeispiel:

```
List<int> dieListe;
// ....
front_insert_iterator<List<int> > frontInsIter(dieListe);
int i = 1;
while(i < 5) {
    *frontInsIter++ = i++;           // Zahlen vorne einfügen
}
```

`front_inserter()` ist eine Funktion, die einen `front_insert_iterator` zurückgibt. Ein Beispiel mit dem Standardalgorithmus `copy()` (siehe auch *cppbuch/k29/finsert.cpp*):

```
// Einfügen aller Elemente im Bereich [a, b) am Anfang von dieListe.
// a und b sind Iteratoren eines anderen Containers.
copy(a, b, front_inserter(dieListe));
```

2. `back_insert_iterator` (Beispiel siehe *cppbuch/k29/binsert.cpp*)

Dieser Insert-Iterator fügt etwas am Ende eines Containers ein. Der Container muss die Methode `push_back()` zur Verfügung stellen. Anwendungsbeispiel:

```
List<int> dieListe;
// ....
back_insert_iterator<List<int> > backInsIter(dieListe);
int i = 1;
while(i < 5) {
    *backInsIter++ = i++;           // Zahlen anhängen
}
```

`back_inserter()` ist eine Funktion, die einen `back_insert_iterator` zurückgibt. Ein Beispiel mit dem Standardalgorithmus `copy()`:

```
// Anhängen aller Elemente im Bereich [a, b) an das Ende von dieListe.
// a und b sind Iteratoren eines anderen Containers.
copy(a, b, back_inserter(dieListe));
```

3. `insert_iterator` (Beispiel siehe *cppbuch/k29/insert.cpp*)

Dieser Insert-Iterator fügt etwas an einer ausgewählten Position in den Container ein. Der Container muss die Methode `insert()` zur Verfügung stellen. Die Anwendung

ist ähnlich wie vorher, nur dass dem Insert-Iterator die gewünschte Einfügeposition mitgegeben werden muss:

```
list<int> dieListe;
list<int>::iterator pos;
// .... hier pos an die gewünschte Stelle bringen
insert_iterator<list<int> > iter(dieListe, pos);
int i = 1;
while(i < 5) {
    *iter++ = i++;          // Zahlen bei pos einfügen
}
```

`inserter()` ist eine Funktion, die einen `insert_iterator` zurückgibt. Die Anwendung wird dadurch manchmal einfacher. Ein Beispiel mit dem Standardalgorithmus `copy()` von Seite 717:

```
// Einfügen aller Elemente im Bereich [a, b) an die Stelle pos von dieListe.
// a und b sind Iteratoren eines anderen Containers.
copy(a, b, inserter(dieListe, pos));
```

29.5 Stream-Iteratoren

Stream-Iteratoren dienen zum sequenziellen Lesen und Schreiben von Strömen mit den bekannten Operatoren `<<` und `>>`. Der `Istream`-Iterator ist ein Input-Iterator, der `Ostream`-Iterator ein Output-Iterator. Beispiele:

```
// Anzeige aller durch Zwischenraumzeichen getrennten Zeichenfolgen:
ifstream quelle("Datei.txt");
istream_iterator<string> pos(quelle), ende;
while(pos != ende) {
    cout << *pos++ << endl;
}
```

Die Dereferenzierung von `pos` in der Schleife gibt den gelesenen Wert zurück. Durch Erben von der Klasse `istream_iterator` und Redefinieren einiger Methoden lassen sich eigene `Istream`-Iteratoren mit besonderen Eigenschaften schreiben. Dem Konstruktor eines `Ostream`-Iterators kann wahlweise eine Zeichenkette zur Trennung von Elementen mitgegeben werden.

```
// Anzeige aller durch Zwischenraumzeichen getrennten Zeichenfolgen,
// wobei in der Ausgabe jede Zeile mit einem * versehen wird:
ifstream quelle("Datei.txt");
ofstream ziel("Ergebnis.txt");
istream_iterator<string> iPos(quelle), Ende;
ostream_iterator<string> oPos(ziel, "*\n");
while(iPos != Ende) {
    *oPos++ = *iPos++;
}
```

IStream-Iterator für Bezeichner

Im Folgenden wird als konkretes Beispiel ein IStream-Iterator, der die Bezeichner (englisch *identifier*) einer Datei liest. Weil der Eingabeoperator >> für die Klasse `string` schon existiert, kann ihm keine neue Bedeutung zugewiesen werden. Deshalb wird eine Klasse `Identifier` angelegt, die von `string` erbt und für die `operator>>()` definiert wird:

Listing 29.4: Leere Klasse `Identifier`

```
// cppbuch/k29/bezeichnerlesen/identifier.h
#ifndef IDENTIFIER_H
#define IDENTIFIER_H
#include <cctype>
#include <iostream>
#include <string>

class Identifier : public std::string { };

inline std::istream& operator>>(std::istream& is, Identifier& identifier) {
    identifier.assign(""); // alten Inhalt löschen
    char c = '\0';
    while(is && !(isalpha(c) || '_' == c)) { // Wortanfang finden
        is.get(c);
    }
    identifier += c;
    // Wenn der Anfang gefunden wurde, werden alle folgenden Unterstriche und alphanu-
    // merische Zeichen gesammelt. Whitespace oder Sonderzeichen beenden das Einlesen.
    while(is && (isalnum(c) || '_' == c)) {
        is.get(c);
        if(isalnum(c) || '_' == c)
            identifier += c;
    } // Das letzte Zeichen gehört nicht zum Identifier und wird deshalb
    is.putback(c); // in den Stream zurückgeschrieben.
    return is;
}
#endif
```

In der Anwendung wird ein `istream_iterator` für diesen Typ erzeugt, der mit dem `ifstream` initialisiert wird. Das folgende Programm gibt die Bezeichner auf der Konsole aus:

Listing 29.5: Anwendung für IStream-Iterator

```
// cppbuch/k29/bezeichnerlesen/main.cpp
#include <iterator>
#include <fstream>
#include "identifier.h"
using namespace std;

int main( ) {
    ifstream datei("main.cpp"); // Eingabedatei öffnen
    istream_iterator<Identifier> iter(datei), end;
    while(iter != end) {
        cout << *iter++ << endl;
    }
}
```

30

Algorithmen

Dieses Kapitel behandelt die folgenden Themen:

- Zusammenarbeit mit Iteratoren und Containern
- Algorithmen mit Prädikat
- Algorithmen mit binärem Prädikat
- Übersicht

Alle im Header `<algorithm>` vorhandenen Algorithmen sind unabhängig von der speziellen Implementierung der Container, auf denen sie arbeiten. Sie kennen nur Iteratoren, über die auf die Datenstrukturen in Containern zugegriffen werden kann. Die Iteratoren müssen nur wenigen Kriterien genügen (siehe Kapitel [11.2](#)). Dadurch bedingt können Iteratoren sowohl komplexe Objekte als auch einfache Zeiger sein. Bei der Übergabe zweier Iteratoren gilt die übliche, auf Seite [767](#) beschriebene Definition für Intervalle.

Alle Algorithmen sind im Namespace `std`. Sie sind von der speziellen Implementierung der Container, auf denen sie arbeiten, vollständig getrennt. Sie kennen nur Iteratoren, über die auf die Datenstrukturen in Containern zugegriffen werden kann. Manche Algorithmen tragen denselben Namen wie Container-Methoden. Durch die Art des Gebrauchs tritt jedoch keine Verwechslung auf. Die vollständige Trennung kann aber auch Nachteile haben: Ein sehr allgemeiner Algorithmus `find()` wird einen Container gelegentlich vom Anfang bis zum Ende durchsuchen müssen. Die Komplexität ist $O(N)$, wobei N die Anzahl der Elemente des Containers ist. Bei Kenntnis der Container-Struktur könnte `find()`

sehr viel schneller sein. Zum Beispiel ist die Komplexität der Suche in einem sortierten Set-Container nur $O(\log N)$. Deshalb gibt es einige Algorithmen, die unter demselben Namen sowohl als allgemeiner Algorithmus als auch als Elementfunktion eines Containers auftreten. Normalerweise ist die maßgeschneiderte Elementfunktion vorzuziehen.

30.1 Algorithmen mit Prädikat

Mit Prädikat ist ein Funktionsobjekt gemeint, das einem Algorithmus mitgegeben wird, und das einen Wert vom Typ `bool` zurückgibt, wenn es auf einen dereferenzierten Iterator angewendet wird. Anstelle des Funktors kann es auch eine Funktion sein. Der dereferenzierte Iterator ist nichts anderes als eine Referenz auf ein Objekt, das im Container abgelegt ist. Das Funktionsobjekt soll ermitteln, ob dieses Objekt eine bestimmte Eigenschaft hat. Nur wenn diese Frage mit `true` beantwortet wird, findet der Algorithmus auf dieses Objekt Anwendung. Ein allgemeines Schema dafür ist:

```
template <class InputIterator, class Predicate>
void algorithm(InputIterator first,
               InputIterator last,
               Predicate pred) {
    while (first != last) {
        if(pred(*first)) {           // gilt Prädikat?
            show_it(*first);        // ... oder andere Funktion
        }
        ++first;
    }
}
```

Die Klasse `Predicate` darf ein Objekt nicht verändern. Einige Algorithmen, die Prädikate benutzen, haben eine Endung `_if` im Namen, andere nicht. Allen gemeinsam ist, dass ein Prädikat in der Parameterliste erwartet wird.



Ein Beispiel für die Anwendung eines unären Prädikats sehen Sie auf der Seite 661.

30.1.1 Algorithmen mit binärem Prädikat

Ein binäres Prädikat verlangt zwei Argumente. Damit kann eine Bedingung für zwei Objekte im Container formuliert werden, zum Beispiel ein Vergleich. Der Algorithmus könnte folgenden Kern enthalten:

```
if(binary_pred(*first, *second)) { // gilt Prädikat?
    do_something_with(*first, *second);
    // ...
}
```

In diesem Sinn könnten auch Funktionsobjekte als binäres Prädikat verwendet werden. Der zweite Parameter eines binären Prädikats braucht allerdings kein Iterator zu sein:

```
template <class InputIterator,
```

```

        class binaryPredicate,
        class T>
void another_algorithm(InputIterator first,
                      InputIterator last,
                      binaryPredicate bpred,
                      T aValue) {
    while (first != last) {
        if(bpred(*first, aValue)) {
            show_it(*first);
        }
        ++first;
    }
}

```



Die Anwendung binärer Prädikate sehen Sie u. a. auf den Seiten [660](#) und [693](#).



Übungen

30.1 Erweitern Sie die Lösung zu Aufgabe [28.4](#), indem Sie mit dem Algorithmus `count_if` ermitteln, wie viele Personen eines bestimmten Rangs vorhanden sind.

30.2 Erweitern Sie die Lösung zu Aufgabe [28.4](#), indem Sie mithilfe des Algorithmus `equal_range` alle Personen eines bestimmten Rangs anzeigen.

30.2 Übersicht

Der C++-Standard unterteilt die Algorithmen in die Bereiche

- Nicht-verändernde Algorithmen. Diese Algorithmen lassen die Elemente eines Containers unverändert (z.B. suchen).
- Verändernde Algorithmen. Diese Algorithmen ändern die Elemente eines Containers (z.B. Container mit Werten füllen).
- Sortieren und verwandte Algorithmen. Dazu gehören auch die binäre Suche, Algorithmen für Mengen, Heap-Algorithmen und andere.
- Algorithmen der C-Bibliothek. Damit sind die C-Algorithmen `qsort()` (Quicksort) und `bsearch()` (binäre Suche) gemeint.
- Verallgemeinerte numerische Algorithmen.

Dieses Buch führt die Algorithmen nicht entsprechend der Aufteilung im C++-Standard auf. Ein Algorithmus ist zur Lösung eines Problems gedacht, weswegen eine problemorientierte Darstellung bevorzugt wird. Das Kapitel [24](#) stellt Algorithmen für verschiedene Aufgaben vor.

Wenn Sie einen Algorithmus suchen, sehen Sie am besten im Inhaltsverzeichnis bei Kapitel 24 nach, ob die zu lösende Aufgabe dort aufgeführt ist. Wenn Sie jedoch gezielt nach einem bestimmten STL-Algorithmus suchen, helfen Ihnen das Register des Buchs und die nachfolgenden Übersichtstabellen.

Tabelle 30.1: Verändernde Algorithmen

Algorithmus	Header	siehe Abschnitt	Seite
copy	<code><algorithm></code>	24.12.5	717
copy_n	<code><algorithm></code>	24.12.5	719
copy_if	<code><algorithm></code>	24.12.5	718
copy_backward	<code><algorithm></code>	24.12.5	717
fill	<code><algorithm></code>	24.3.2	648
fill_n	<code><algorithm></code>	24.3.2	648
generate	<code><algorithm></code>	24.3.3	648
generate_n	<code><algorithm></code>	24.3.3	648
iter_swap	<code><algorithm></code>	24.12.6	719
is_partitioned	<code><algorithm></code>	siehe [ISOC++, 25.3.13]	
move	<code><algorithm></code>	27.2	749
move_backward	<code><algorithm></code>	27.2	749
partition	<code><algorithm></code>	24.4.1	666
partition_copy	<code><algorithm></code>	siehe [ISOC++, 25.3.13]	
partition_point	<code><algorithm></code>	siehe [ISOC++, 25.3.13]	
random_shuffle	<code><algorithm></code>	24.3.12	657
remove	<code><algorithm></code>	24.12.9	723
remove_if	<code><algorithm></code>	24.12.9	723
remove_copy	<code><algorithm></code>	24.12.9	723
remove_copy_if	<code><algorithm></code>	24.12.9	723
replace	<code><algorithm></code>	24.12.8	722
replace_if	<code><algorithm></code>	24.12.8	722
replace_copy	<code><algorithm></code>	24.12.8	722
replace_copy_if	<code><algorithm></code>	24.12.8	722
reverse	<code><algorithm></code>	24.3.14	660
reverse_copy	<code><algorithm></code>	24.3.14	660
rotate	<code><algorithm></code>	24.3.11	656
rotate_copy	<code><algorithm></code>	24.3.11	656
stable_partition	<code><algorithm></code>	24.4.1	666
swap	<code><algorithm></code>	24.12.6	719
swap_ranges	<code><algorithm></code>	24.12.6	719
transform	<code><algorithm></code>	24.12.7	720
unique	<code><algorithm></code>	24.3.13	658
unique_copy	<code><algorithm></code>	24.3.13	658

Tabelle 30.2: Sortieren und Verwandtes

Algorithmus	Header	siehe Abschnitt	Seite
binary_search	<algorithm>	24.5.6	681
equal_range	<algorithm>	24.5.6	682
includes	<algorithm>	24.6.1	684
inplace_merge	<algorithm>	24.4.6	673
is_heap	<algorithm>	24.7.5	692
is_heap_until	<algorithm>	24.7.5	692
is_sorted	<algorithm>	siehe [ISOC++, 25.4.1.5]	
is_sorted_until	<algorithm>	siehe [ISOC++, 25.4.1.5]	
lexicographical_compare	<algorithm>	24.3.18	665
lower_bound	<algorithm>	24.5.6	682
make_heap	<algorithm>	24.7.3	691
max	<algorithm>	24.12.11	726
max_element	<algorithm>	24.3.10	655
merge	<algorithm>	24.4.6	671
min	<algorithm>	24.12.11	726
min_element	<algorithm>	24.3.10	655
minmax	<algorithm>	24.12.11	726
minmax_element	<algorithm>	24.3.10	655
next_permutation	<algorithm>	24.3.17	663
nth_element	<algorithm>	24.4.5	669
partial_sort	<algorithm>	24.4.4	669
partial_sort_copy	<algorithm>	24.4.4	669
pop_heap	<algorithm>	24.7.1	689
prev_permutation	<algorithm>	24.3.17	663
push_heap	<algorithm>	24.7.2	690
set_difference	<algorithm>	24.6.4	686
set_intersection	<algorithm>	24.6.3	686
set_symmetric_difference	<algorithm>	24.6.5	687
set_union	<algorithm>	24.6.2	685
sort	<algorithm>	24.4.2	667
sort_heap	<algorithm>	24.7.4	691
stable_sort	<algorithm>	24.4.3	667
upper_bound	<algorithm>	24.5.6	682

Tabelle 30.3: Nicht-verändernde Algorithmen

Algorithmus	Header	siehe Abschnitt	Seite
adjacent_find	<code><algorithm></code>	24.5.4	679
all_of	<code><algorithm></code>	24.3.16	662
any_of	<code><algorithm></code>	24.3.16	662
count	<code><algorithm></code>	24.3.15	661
count_if	<code><algorithm></code>	24.3.15	661
equal	<code><algorithm></code>	24.8.2	694
find	<code><algorithm></code>	24.5.1	674
find_end	<code><algorithm></code>	24.5.3	678
find_first_of	<code><algorithm></code>	24.5.2	675
for_each	<code><algorithm></code>	24.12.3	716
mismatch	<code><algorithm></code>	24.8.1	692
none_of	<code><algorithm></code>	24.3.16	662
search	<code><algorithm></code>	24.5.3	677
search_n	<code><algorithm></code>	24.5.5	680

Tabelle 30.4: Verallgemeinerte numerische Algorithmen

Algorithmus	Header	siehe Abschnitt	Seite
accumulate	<code><numeric></code>	24.3.5	650
adjacent_difference	<code><numeric></code>	24.3.9	653
inner_product	<code><numeric></code>	24.3.7	651
iota	<code><numeric></code>	24.3.4	649
partial_sum	<code><numeric></code>	24.3.8	653

Tabelle 30.5: Algorithmen der C-Library

Algorithmus	Header	siehe Abschnitt	Seite
bsearch	<code><stdlib.h></code>	35.8	878
qsort	<code><stdlib.h></code>	5.9	224

31

Nationale Besonderheiten

Dieses Kapitel behandelt die folgenden Themen:

- Länderspezifische Zeichenformate einstellen
- Zeichensätze und -codierung
- Geld-, Datums- und Zahlenformate
- Konstruktion eines eigenen Formats

Die Klasse `locale` (Header `<locale>`) bestimmt die nationalen Besonderheiten von Zeichensätzen. Dazu gehören Sonderzeichen wie die deutschen Umlaute oder Zeichen mit Akzent wie in den Worten *señor* und *garçon*. Die Ordnung zum Vergleich von Zeichenketten wird dadurch definiert, das heißt zum Beispiel, ob *ä* unter *a* oder unter *ae* einsortiert wird, und das Erscheinungsbild für die Ein- und Ausgabe numerischer Größen (Dezimalpunkt oder -komma?) und Datumsangaben (31.1.2011 oder 1/31/2011). Die verschiedenen Kategorien werden in Facetten (englisch *facets*) unterteilt. Dieses Kapitel konzentriert sich auf die häufigsten Anwendungen. Es ist möglich, eigene Facetten zu schreiben, die von den vorhandenen abgeleitet sind. Weitere Details sind [\[ISOC++\]](#) und [\[KL\]](#) zu entnehmen.

31.1 Sprachumgebungen festlegen und ändern

Ein `locale`-Objekt wird von der `Iostream`-Bibliothek benutzt, damit die üblichen nationalen Gepflogenheiten bei der Ein- und Ausgabe eingehalten werden. Wenn von der deutschen Schreibweise von Zahlen auf die angloamerikanische umgeschaltet werden soll, wird das `locale`-Objekt des Ein- oder Ausgabestroms entsprechend ausgewechselt. Das geschieht mit der Funktion `imbue()`, der als Parameter ein `locale`-Objekt übergeben wird. Das englische Wort *imbue* bedeutet etwa »erfüllen (mit)« oder »inspirieren (mit)«.

```
using namespace std; // auch locale ist in std
// Beispiele zur Einstellung der Sprachumgebung
locale eineSprachumgebung("POSIX");
cin.imbue(eineSprachumgebung);

// global Deutsch als Sprachumgebung setzen, dabei vorherige
// globale Sprachumgebung merken:
locale deutsch("de_DE");
locale vorherigeSprachumgebung = locale::global(deutsch);
cout.imbue(locale("de_DE")); // Ausgabe deutsch formatieren
// ...
locale::global(vorherigeSprachumgebung); // Sprachumgebung zurücksetzen
```

Das Setzen der globalen Sprachumgebung, die normalerweise durch Abfrage der Betriebssystemumgebungsvariablen `LANG` voreingestellt wird, wirkt sich nicht auf existierende Streams wie `cin` oder `cout` aus, nur auf neu erzeugte – deswegen muss ggf. `imbue()` angewendet werden. Falls `LANG` nicht definiert ist, wird automatisch die C-Sprachumgebung gesetzt, auch `classic` genannt (siehe Beispiel unten). `POSIX` (= Portable Operating System Interface for uniX) ist eine Familie von Standards für Betriebssystemschnittstellen. Anstatt `POSIX` kann eine von vielen anderen Umgebungen gewählt werden, von denen einige hier aufgelistet sind:

<code>de_DE</code>	= Deutsch für Deutschland (ISO 8859-1)
<code>de_DE@euro</code>	= Deutsch für Deutschland mit Euro-Zeichen (ISO 8859-15)
<code>de_DE.utf8</code>	= Deutsch für Deutschland (Unicode UTF-8)
<code>de_CH</code>	= Deutsch für die Schweiz
<code>en_GB</code>	= Englisch für Großbritannien
<code>en_CA</code>	= Englisch für Kanada
<code>en_US</code>	= amerikanisches Englisch
<code>es_SV</code>	= Spanisch für El Salvador
...	

Auf Linux-Systemen kann das eingestellte Locale mit `locale` angezeigt werden. `locale -a` listet alle verfügbaren Locales auf.

```
cout.imbue(locale::classic());
```

setzt die Standardausgabe auf die C-Sprachumgebung zurück.

31.1.1 Die locale-Funktionen

- `locale()`
Konstruktor, der eine Kopie des aktuellen globalen `locale`-Objekts (gegebenenfalls mit `global()` eingestellt, siehe oben) erzeugt.
- `explicit locale(const char* name)`
`explicit locale(const string& name)`
Konstruktor. Die übergebene Zeichenkette bzw. der `String` `name` ist zum Beispiel `"de_DE"`.
- `locale(const locale& other, const char* name, category cat)`
`locale(const locale& other, string name, category cat)`
Der Konstruktor kopiert `other` mit Ausnahme der Facetten, die in `cat` definiert sind. `cat` kann zum Beispiel (`monetary | numeric`) sein. Sie werden entsprechend `name` gewählt. Zum Typ `category` siehe Abschnitt 31.4.
- `locale(const locale& loc1, const locale& loc2, category cats)`
Der Konstruktor kopiert `loc1` mit Ausnahme der Facetten, die in `loc2` definiert sind. Diese werden entsprechend `cats` gewählt.
- `template<class Facet> locale(const locale& other, Facet *f)`
Der Konstruktor kopiert `other`. Falls `f` ungleich 0 ist, wird aber die Facette `Facet` durch `*f` definiert.
- `const locale& operator=(const locale& rechts)`
Zuweisungsoperator; gibt `*this` zurück
- `bool operator==(const locale& other) const`
`bool operator!=(const locale& other) const`
Vergleichsoperatoren
- `template<class Facet> locale combine(const locale& other)`
gibt eine Kopie von `*this` zurück, wobei aber die Facette `Facet` durch die entsprechende von `other` ersetzt wird.
- `string name() const`
gibt den Namen des `locale`-Objekts zurück, falls definiert. Andernfalls wird `»*«` zurückgegeben.
- `bool operator()(const string& s1, const string& s2) const`
gibt `s1 < s2` zurück. Damit kann leicht zum Beispiel ein Vektor `v` entsprechend den nationalen Zeichenvergleichsregeln, die in einem `locale`-Objekt `loc` festgelegt sind, sortiert werden (Beispiel siehe Seite 630).
- `static locale global(const locale& loc)`
setzt das globale `locale`-Objekt. Der vorherige Wert wird zurückgegeben.
- `static const locale& classic()`
gibt ein `locale`-Objekt für die C-Sprachumgebung zurück (entspricht `locale("C")`).
- `template<class Facet> const Facet& use_facet(const locale& loc)`
gibt die Referenz der Facette des Typs `Facet` des `locale`-Objekts `loc` zurück. Falls eine Facette dieses Typs in `loc` nicht existiert, wird eine `bad_cast`-Exception ausgeworfen. Ein Beispiel für die Anwendung von `use_facet` finden Sie auf den Seiten 632 und 633.
- `template<class Facet> bool has_facet(const locale& loc)`
gibt zurück, ob eine Facette des Typs `Facet` in `loc` existiert.



Hinweis

Im Folgenden werden Objekte des Typs `locale` benutzt, um zum Beispiel eine deutsche Schreibweise für ein Datum einzustellen. Leider sind die Konventionen für die Namensgebung systemabhängig. Um festzustellen, welche Locales (wie `de_DE` und `en_US`) unterstützt werden, können Sie das folgende Programm *checklocale.cpp* abwandeln und nutzen.

Listing 31.1: Test von locales

```
// cppbuch/k31/checklocale.cpp
#include<locale>
#include<iostream>
using namespace std;

int main() {
    const char* loc = setlocale(LC_ALL, 0);
    cout << "aktuell eingestellte Locale: " << loc << endl;

    const char* locales[] = {
        // Unix
        "de_DE",
        "en_US",

        // Alias-Namen für viele Systeme (nicht genormt), siehe
        // /usr/share/locale/locale.alias (Linux)
        // C:/MinGW/share/locale/locale.alias (MinGW/Windows)
        "german",
        "deutsch",
        "french",
        "polish",

        // Windows
        "German_Germany.1252",
        "English_United States.1252"
    };

    for(size_t i = 0; i < sizeof locales / sizeof locales[0]; ++i) {
        cout << locales[i] << " wird von diesem System ";
        loc = setlocale(LC_ALL, locales[i]);
        if(!loc) {
            cout << "NICHT unterstuetzt." << endl;
        }
        else {
            cout << "unterstuetzt. loc = " << loc << endl;
        }
    }
}
```

31.2 Zeichensätze und -codierung

Ein Zeichensatz ist eine Abbildung von Bitmustern auf Zeichen. Damit Sender und Empfänger sich beim elektronischen Datenaustausch verstehen, ist eine Konvention über die Bedeutung der Bitmuster unerlässlich. Die bekannteste ist ASCII (American Standard Code for Information Interchange). Die ASCII-Tabelle definiert die ersten 7 Bits eines Bytes, also 128 Zeichen (siehe Anhang A.3). Umlaute und andere europäische Sonderzeichen sind nicht enthalten, sodass nach und nach viele weitere Zeichensätze hinzukamen. Sehr bekannt ist ISO-8859-1, auch »Latin 1« genannt. Dieser Zeichensatz stimmt in den ersten 128 Bytes mit ASCII überein und definiert in den anderen 128 Bytes Umlaute wie ä, ö, ü und andere Sonderzeichen, sodass ISO-8859-1 mehr als 20 Sprachen abdeckt. ISO-8859-15, auch »Latin 9« genannt, ist eine Modifikation, die auch das Euro-Zeichen € enthält.

ISO-8859-1 hat den großen Vorteil, dass für die Darstellung eines Zeichens ein Byte ausreichend ist. Wenn allerdings noch arabische, chinesische, japanische und koreanische Zeichen gefragt sind, reicht ein Byte nicht. Aus diesem Grund wurde Unicode [Unic] entwickelt. Unicode hat den Anspruch, jedem Zeichen eine Codierung zuzuordnen. Wegen der Fülle der möglichen Zeichen müssen viele Zeichen als Multi-Byte-Sequenzen abgebildet werden. Es gibt mehrere Unicode-Schemata; das am weitesten verbreitete ist UTF-8 (8-Bit Unicode Transformation Format). Jedes UTF-8-Byte hat die Eigenschaften:

- Falls das höchste Bit eines Bytes 0 ist, handelt es sich bei dem Byte um ein ASCII-Zeichen. Die anderen 7 Bits definieren den ASCII-Wert. Die ersten 128 UTF-8-Zeichen stimmen daher mit dem ASCII überein.
- Falls das höchste Bit 1 ist, ist das Byte Teil einer Multi-Byte-Sequenz.



Merke:

Eine Zeichenkette kann nur in Kenntnis der verwendeten Codierung sinnvoll bearbeitet und interpretiert werden.

Das bedeutet auch, dass Tastatur, Editor und Anzeige in der Codierung übereinstimmen müssen. Sie haben sicher schon den Effekt bemerkt, dass Umlaute, die ein Programm auf der Konsole ausgibt, nicht korrekt dargestellt werden. Die Ursache ist die fehlende Übereinstimmung der Codierung. In Linux wird die Codierung des Betriebssystems durch die Variable `LANG` (für language) eingestellt. Oft ist `de_DE.UTF-8` üblich. Der `de_DE`-Anteil sorgt dabei für die hier übliche Dezimalpunkt- und Datumsdarstellung usw. Diese Aspekte werden Facetten genannt; Einzelheiten folgen.



Tipp: Einstellung der Konsole auf UTF-8

Windows: Durch Anklicken des Fenstersymbols ganz oben links im Eingabeaufforderungsfenster geht man über »Eigenschaften« zu den Schriftarten. Dort Lucida Console wählen. Dann `chcp 65001` eintippen.

Linux (Suse 11.2): Normalerweise ist die Konsole auf UTF-8 eingestellt. Falls nicht: im

Konsolenfenster oben anklicken: Einstellungen → aktuelles Profil verwalten → Erweitert. Unten rechts können die Schriftarten gewählt werden.

C++-Zeichenliterale

»Normale« Zeichenliterale werden durch Hochkommata gekennzeichnet, zum Beispiel 'z'. Anstelle des Zeichens z können alle anderen Zeichen des Basiszeichensatzes, der normalerweise in etwa ASCII entspricht, treten, außer dem Hochkomma selbst wegen der Begrenzerfunktion, dem Backslash und dem Zeilenendezeichen. Mit dem Backslash werden Sonderzeichen eingeleitet, wie die Tabelle 1.6 auf Seite 52 zeigt. Es gibt aber weitere Möglichkeiten, die durch ein Präfix markiert werden.

- u'z' : Ein vorangestelltes u besagt, dass das Literal den Typ `char16_t` hat. Es muss mit 16 Bits darstellbar sein, und sein Wert wird durch ISO 10646 definiert. ISO 10646 ist eine Norm für den *Universal Character Set* (UCS), der die Unterformate UCS-2 und UCS-4 hat, entsprechend einer Codierung in 2 bzw. 4 Bytes. Das Standardisierungsgremium für ISO 10646 arbeitet mit dem Unicode-Gremium zusammen, um die verschiedenen Definitionen nicht auseinanderlaufen zu lassen. So entspricht UCS-4 UTF-32.
- U'z' : Ein vorangestelltes U ist dementsprechend ein `char32_t`-Zeichen.
- L'z' : Ein großes L steht für ein `wchar_t`-Zeichen. Das »w« steht für »wide«. Dieser Typ ist für Zeichensätze gedacht, die nicht mit einem Byte pro Zeichen auskommen. Die 1-Byte-Zeichen heißen im Gegensatz dazu »narrow characters«.

C++-Stringliterale

Den C++-Zeichenliteralen stehen entsprechende Stringliterale gegenüber. Die wichtigsten Typen sind:

```
"ASCII- oder ISO 8859-1-String", d.h. ein Byte pro Zeichen
L"String mit wchar_t-Zeichen"
u8"UTF-8 codierter String"
```

u8 und weitere Möglichkeiten sind neu dazugekommen, werden von heutigen Compilern aber noch nicht unterstützt. Wenn das Programm im ISO-8859-1-Format abgespeichert wird, werden im folgenden Beispiel die den Umlauten entsprechenden Zahlenwerte ausgegeben:

Listing 31.2: ISO-8859-1 Codierung

```
// cppbuch/k31/narrow.cpp (ISO-8859-1-codiert abgespeichert)
#include<algorithm>
#include<iostream>
#include<string>
using namespace std;

int main() {
    locale loc;
    cout << "loc.name()= " << loc.name() << endl;
    locale dt("de_DE");
```

```

cout << "dt.name()= " << dt.name() << endl;
locale vorher = locale::global(dt); // dt für alles setzen
string s("ÄÖÜäöüß");
cout.imbue(dt);
for(size_t i = 0; i < s.length(); ++i) {
    cout << "Zeichen " << i << ": " << s[i] << " "
        << ((int)s[i] +256) // Korrektur für signed char
        << endl;
}
for(size_t i = 0; i < s.length(); ++i) {
    s[i] = toupper(s[i], dt); // siehe Text
}
cout << "\n Nach toupper: " << s << endl;
}

```

Die Konsole muss natürlich auch auf ISO-8859-1 eingestellt sein. Die Anweisung `s[i] = toupper(s[i], dt);` hätte auch durch `s[i] = toupper(s[i]);`, also Aufruf der entsprechenden C-Funktion ersetzt werden können, weil die `locale`-Einstellung vorher auf `de_DE` gesetzt worden ist. Die aktuelle Einstellung wird von `toupper(char)` berücksichtigt. Die ISO-8859-1-codierte Zeile

```
string s("ÄÖÜäöüß");
```

könnte durch

```
string s("\xC4\xD6\xDC\xE4\xF6\xFC\xDF");
```

ersetzt werden. Das wäre zwar von jedem Editor lesbar, egal mit welcher Codierungseinstellung. Aber portabel wäre auch das nicht, denn UTF-8-codiert sehen die Umlaute so aus:

```
string s("\xC3\x84\xC3\x96\xC3\x9C\xC3\xA4\xC3\xB6\xC3\xBC\xC3\x9F");
```



Merke:

Ein Programm, das Zeichenketten mit nicht-ASCII-Zeichen enthält, ist *nicht portabel*.

Unter *portabel* wird hier verstanden, dass der Quellcode auf ein beliebiges anderes System übertragen und dort übersetzt werden kann, und dass das Programm unabhängig von der `locale`-Einstellung und der Umgebung dasselbe Ergebnis liefert. Im Folgenden sehen Sie ein Beispiel, das UTF-8-codiert ist und einen `wchar_t`-String benutzt.

Listing 31.3: Wide-String

```

1 // cppbuch/k31/wide.cpp (UTF-8 codiert abgespeichert)
2 #include<iostream>
3 #include<fstream>
4 #include<string>
5 #include<locale>
6 using namespace std;
7
8 int main() {

```



```

9   locale dt("de_DE.utf-8");
10  locale::global(dt); // dt fuer alles setzen
11  wstring ws(L"ÄÖÜäöüß");
12  wcout.imbue(dt);
13  wcout << "Ausgabe mit wcout: " << ws << endl;
14  wcout << L"Länge=" << ws.length() << endl;
15  wcout << "sizeof(wchar_t): " << sizeof(wchar_t) << endl;
16  wofstream wdatei("ausgabe.txt");
17  wdatei << "Ausgabe in wofstream: " << ws << endl;
18  for(size_t i = 0; i < ws.length(); ++i) {
19      wdatei << "WZeichen " << i << ": "
20          << ws[i] << endl;
21  }
22  for(size_t i = 0; i < ws.length(); ++i) {
23      ws[i] = toupper(ws[i], dt);
24  }
25  wcout << "nach toupper(): " << ws << endl;
26  }

```

Die Konsole muss natürlich auch auf UTF-8 eingestellt sein, um eine lesbare Anzeige zu bekommen. Bemerkungen zu diesem Beispiel:

- Bei einem auf UTF-8 eingestellten Editor wird der Quellcode in UTF-8 gespeichert. Nicht-ASCII-Zeichen werden also Multibyte-Sequenzen (Zeile 11).
- Mit `wcout`, der `cout`-Entsprechung für `wchar_t`-Zeichen, wird der `wstring` `ws` auf der Konsole ausgegeben (Zeile 13).
- Die interne Darstellung eines `wstrings` im ausführbaren Programm ist ein Array mit n `wchar_t`-Zeichen, wobei n die Anzahl der Zeichen (nicht der Bytes!) ist, wie sie in Zeile 14 angezeigt wird.
- `wchar_t` ist ein `int`-Typ. Seine Größe wird in Zeile 15 ausgegeben (auf meinem System 4 Byte).
- Die nachfolgende Ausgabe wird in eine Datei des Typs `wofstream` umgeleitet (Zeilen 16 bis 21).
- Die Funktion `toupper()` (Zeile 23) funktioniert zeichenweise unter Berücksichtigung der übergebenen `locale`-Einstellung.

Wie man sieht, ist die Arbeit mit Wide-Strings nicht so komfortabel wie mit »normalen« Strings. Auch wird die Umwandlung von Strings verschiedener Codierungen, wie es das Programm *iconv* (siehe Tipp) leistet, noch kaum unterstützt.



Tipp

Der Befehl `iconv` unter Linux (Windows: MinGW-Version von `iconv`) wandelt Dateiformate um. Aufruf zum Beispiel: `iconv -f ISO-8859-1 -t UTF-8 -o utf8.txt deDE.txt`

Dabei bedeuten `-f` (from) das Quellformat, `-t` (to) das Zielformat. Nach `-o` folgt der Name der Ausgabedatei; zuletzt der Name der zu konvertierenden Datei. `iconv -l` zeigt alle dem Programm bekannten Zeichensatzcodierungen an. Dabei sind einige unter mehreren Namen aufgeführt. Zum Beispiel sind die ISO-10464 UCS-Codes dasselbe wie UTF. Das

Wort »bekannt« bedeutet in diesem Zusammenhang nicht, dass von jeder Codierung zu jeder anderen konvertiert werden kann.

31.3 Zeichenklassifizierung und -umwandlung

Die Definition, ob zum Beispiel ein spezielles Zeichen ein Buchstabe oder etwas anderes ist, hängt von der Sprache ab. Aus diesem Grund gibt es die Funktionen der Tabellen 35.2 und 35.1 (Seite 875 f.) in einer sprachumgebungsabhängigen Variante für verschiedene Zeichentypen `charT`:

```
// Zeichenklassifizierung
template<typename charT> bool isspace (charT c, const locale& loc);
template<typename charT> bool isprint (charT c, const locale& loc);
template<typename charT> bool iscntrl (charT c, const locale& loc);
template<typename charT> bool isupper (charT c, const locale& loc);
template<typename charT> bool islower (charT c, const locale& loc);
template<typename charT> bool isalpha (charT c, const locale& loc);
template<typename charT> bool isdigit (charT c, const locale& loc);
template<typename charT> bool ispunct (charT c, const locale& loc);
template<typename charT> bool isxdigit(charT c, const locale& loc);
template<typename charT> bool isalnum (charT c, const locale& loc);
template<typename charT> bool isgraph (charT c, const locale& loc);

// Zeichenumwandlung
template<typename charT> charT toupper(charT c, const locale& loc);
template<typename charT> charT tolower(charT c, const locale& loc);
```

31.4 Kategorien

Locale-Sprachumgebungen enthalten verschiedene Kategorien, die in Facetten unterteilt sind. Das abfragbare Datum `locale::category` ist eine `int`-Bitmaske, die die Oder-Verknüpfung aller oder eines Teils der folgenden Konstanten ist: `none`, `ctype`, `monetary`, `numeric`, `time` und `messages`. Jede dieser Kategorien definiert eine Menge lokaler Facetten, wie die Tabelle 31.1 zeigt. Die Facetten sind Template-Klassen, die als Argument den Typ `char` oder `wchar_t` für wide characters haben können.

Tabelle 31.1: Kategorien und Facetten (charT-Template-Typ)

Kategorie	Facetten	Zweck
collate	collate<charT>	Zeichenvergleich
ctype	ctype<charT> codecvt<char, char, mbstate_t>	Zeichenklassifizierung Zeichenkonvertierung
numeric	numpunct<charT> num_get<charT> num_put<charT>	Zahlenformatierung Eingaben Ausgaben
monetary	moneypunct<char, Intl = false> money_get<charT> money_put<charT>	Währungsformatierung (Intl = true für internationale Festlegungen) Eingaben Ausgaben
time	time_get<charT> time_put<charT>	Zeiteingaben Zeitausgaben
messages	messages<charT>	Strings aus Message-Katalogen holen

31.4.1 collate

Die Klasse `template<typename charT> class collate` ist eine Facette, die die Funktionen für Vergleiche von Zeichenketten kapselt. Sie besitzt die folgenden öffentlichen Elementfunktionen:

■ `int compare(const charT* low1, const charT* high1,
 const charT* low2, const charT* high2) const`

Diese Funktion vergleicht zwei Zeichenketten, die durch die Intervalle `[low1, high1)` und `[low2, high2)` definiert werden (zur Definition von Intervallen siehe Seite 767). Es wird 1 zurückgegeben, falls die erste Zeichenkette größer als die zweite ist, -1 im umgekehrten Fall und 0 bei Gleichheit. Der Operator `locale::operator()()` von Seite 823 ruft diese Funktion auf. Das Beispiel auf Seite 630 zeigt, wie ein `locale`-Objekt zur sprachlich korrekten Sortierung eingesetzt wird.



Mehr zur sprachlich richtigen Sortierung lesen Sie auf Seite 629.

■ `basic_string<charT> transform(const charT* low, const charT* high) const`
gibt das String-Äquivalent des Bereichs zurück, wobei die Ordnungsrelationen erhalten bleiben, d.h. ein Vergleich zweier erzeugter Strings mit dem Algorithmus `lexical_compare` (siehe Seite 665) muss zum selben Ergebnis wie `compare()` führen.

■ `long hash(const charT* low, const charT* high) const`
gibt einen Hash-Wert für den übergebenen Bereich zurück. Dabei ist gewährleistet, dass der Hash-Wert auch bei unterschiedlichen Werten im Bereich stets derselbe ist, wenn nur `compare()` die Bereiche als gleich ansieht, d.h. 0 zurückgibt. Ein Beispiel dafür könnte sein, dass `ä` und `ae` bei einer Sortierung gleich behandelt werden sollen.

Ein Beispiel für die Benutzung von `collate` finden Sie in der Klasse `Stringvergleich` auf Seite 633.

31.4.2 ctype

Die Klasse `template<typename charT> class ctype` kapselt die Funktionen für Zeichenklassifizierung und -umwandlung. So gibt zum Beispiel der Aufruf der Funktion `toupper(c, loc)` von Seite 829 für Zeichen des Typs `char` nichts anderes als `use_facet<ctype<char>>(loc).toupper(c)` zurück. `ctype` erbt von der Basisklasse `ctype_base`, die eine Maske `mask` für Klassifizierungszwecke etwa wie folgt definiert:

```
enum mask {
    space = 1<<0, print = 1<<1, cntrl = 1<<2,
    upper = 1<<3, lower = 1<<4, alpha = 1<<5,
    digit = 1<<6, punct = 1<<7, xdigit = 1<<8,
    alnum = alpha|digit, graph = alnum|punct
};
```

Die öffentliche Schnittstelle enthält die folgenden Methoden:

- `bool is(mask m, charT c) const`
gibt zurück, ob `c` zur Klassifizierung `m` passt.
- `const charT* is(const charT* low, const charT* high, mask* vec) const`
Diese Funktion berechnet einen Wert `M` vom Typ `ctype_base::mask` für jedes der Zeichen im Intervall `[low, high)` und legt das Ergebnis im Array `vec` beginnend an der Stelle `vec[0]` ab. `high` wird zurückgegeben.
- `const charT* scan_is(mask m, const charT* low, const charT* high) const`
gibt einen Zeiger auf das erste Zeichen im Intervall `[low, high)` zurück, das der Klassifizierung `m` genügt. Existiert kein solches Zeichen, wird `high` zurückgegeben.
- `const charT* scan_not(mask m, const charT* low, const charT* high) const`
gibt einen Zeiger auf das erste Zeichen im Intervall `[low, high)` zurück, das *nicht* der Klassifizierung `m` genügt. Existiert kein solches Zeichen, wird `high` zurückgegeben.
- `charT toupper(charT c) const`
gibt den entsprechenden Großbuchstaben zurück, sofern ein solcher existiert. Andernfalls wird das Argument zurückgegeben.
- `const charT* toupper(charT* low, const charT* high) const`
verwandelt alle Zeichen im Bereich `[low, high)` in Großbuchstaben, sofern solche existieren. Es wird `high` zurückgegeben. Vergleichbares Beispiel siehe bei `tolower()` unten.
- `charT tolower(charT c) const`
gibt den entsprechenden Kleinbuchstaben zurück, sofern ein solcher existiert. Andernfalls wird das Argument zurückgegeben.
- `const charT* tolower(charT* low, const charT* high) const`
verwandelt alle Zeichen im Bereich `[low, high)` in Kleinbuchstaben, sofern solche existieren. Es wird `high` zurückgegeben. Im Beispiel von Seite 826 könnte der String `s` unter Verwendung des `locale`-Objekts `dt` wie folgt in Kleinbuchstaben umgewandelt werden:

```
for(size_t i = 0; i < s.length(); ++i) {
    s[i] = std::use_facet<std::ctype<charT>>(dt).tolower(s[i]);
}
```

Der Vorspann `use_facet<ctype<char>>(dt)` gibt die Facette `ctype` des `locale`-Objekts `dt` zurück, deren Funktion `tolower()` dann aufgerufen wird.

- `charT widen(char c) const`
wandelt `c` in eine entsprechende Repräsentation des Typs `charT` um (z.B. wide character `wchar_t`).
- `const char* widen(const char* low, const char* high, charT* to) const`
wandelt jedes Zeichen im Intervall `[low, high)` in eine entsprechende Repräsentation des Typs `charT` um und legt das Ergebnis in `to` ab. Der Rückgabewert ist `high`.
- `char narrow(charT c, char dfault) const`
wandelt `c` in eine entsprechende Repräsentation des Typs `char` um, falls eine solche existiert. Andernfalls wird `dfault` zurückgegeben.
- `const charT* narrow(const charT* low, const charT*, char vorgabe, char* to) const`
wandelt jedes Zeichen im Intervall `[low, high)` in eine entsprechende Repräsentation des Typs `char` um, falls eine solche existiert. Andernfalls wird `vorgabe` genommen. Das Ergebnis wird in `to` abgelegt. Der Rückgabewert ist `high`.

Es existiert eine Spezialisierung `ctype<char>`.

codecvt-Zeichensatzkonvertierung

Die Klasse `template<class internT, class externT, class stateT> class codecvt` dient zur Konvertierung von Zeichensätzen, zum Beispiel von Multibyte-Zeichen nach Unicode. Standardmäßig ist die Implementierung `codecvt<wchar_t, char, mbstate_t>` zur Konvertierung zwischen dem `char`-Zeichensatz und dem Zeichensatz für Wide Characters vorgesehen. Die Interna der Template-Klasse `stateT` bzw. `mbstate_t` sind dem jeweiligen Hersteller der C++-Standardbibliothek vorbehalten. Einzelheiten siehe [ISOC++].

31.4.3 numeric

Die Template-Klassen `num_get` und `num_put` wickeln das formatierte Einlesen bzw. die formatierte Ausgabe ab. Sie werden intern von den Standard-Iostreams benutzt, um Zahlen mit national bedingten Dezimal- und Tausendermarkierungen richtig zu bearbeiten, und sind für normale Benutzer/innen wohl kaum von Bedeutung, da sie versteckt innerhalb des `<<-` bzw. `>>-`Operators Anwendung finden, wie das folgende Beispiel zeigt:

```
// cppbuch/k31/inout.cpp
#include <iostream>
#include <locale>
using namespace std;

int main() {
    cin.imbue(locale("de_DE"));
    cout.imbue(locale("en_US"));
    double f;
    while (cin >> f)    // implizite Nutzung von num_get
        cout << f << endl; // implizite Nutzung von num_put
}
```

Mit den gegebenen `locale`-Objekten würde die Eingabe `3.456,78` die Ausgabe `3,456.78` bewirken. Die Abfrage der Markierungen und anderer Dinge mit der Klasse `num_punct` folgt:

numpunct

Die Facette `template<class charT> class numpunct` hat die folgende öffentliche Schnittstelle:

- `charT decimal_point() const`
gibt den verwendeten Dezimalpunkt zurück (z. B. einen Punkt für `en_US` oder ein Komma für `de_DE`).
- `charT thousands_sep() const`
gibt das Trennzeichen zwischen Tausender-Gruppen zurück.
- `string grouping() const`
Die Zeichen des zurückgegebenen Strings, im Folgenden `s` genannt, sind als *ganzzahlige* Zahlen zu interpretieren, die die Anzahl der Ziffern in der Gruppe angeben, beginnend mit Position 0 als am weitesten rechts stehende Gruppe. Wenn `s.size() <= i` für eine Position `i` gilt, ist die Zahl dieselbe wie die für Position `i-1`. Zum Beispiel wird die Anzahl der Ziffern einer Tausendergruppe gleich 3 sein, d.h. `s == "\003"`. Negative Zahlen charakterisieren unbegrenzte Gruppen wie etwa Zahlen ganz ohne Markierung der Tausender.
- `basic_string<charT> truenamename() const`
`basic_string<charT> falsename() const`
geben den verwendeten Namen (`true` bzw. `false`) für die Ausgabe zurück, sofern `boolalpha == true` ist (vgl. Abschnitt 10.1.1, Seite 378).

Die Klasse kann für ein bestimmtes `locale`-Objekt wie folgt benutzt werden:

```
locale loc; // Kopie des aktuellen globalen locale-Objekts
char dezPunkt = use_facet<numpunct<char>>(loc).decimal_point();
// oder
string wahr = use_facet<numpunct<char>>(loc).truenamename();
cout << wahr; // Ausgabe: true
```

31.4.4 monetary

Diese Kategorie enthält alles, was für die formatierte Ein- und Ausgabe von Geldbeträgen einschließlich der Währungsangaben gebraucht wird. In den folgenden Beispielen wird von einer einfachen Klasse `Geld` ausgegangen.

Listing 31.4: Klasse `Geld`

```
// cppbuch/k31/geld/Geld.h
#include<iostream>

class Geld {
public:
    Geld(long int b = 0L);
    long int getBetrag() const;
private:
    long int betrag;
};

std::istream& operator>>(std::istream& is, Geld& G);
std::ostream& operator<<(std::ostream& os, const Geld& G);
```

money_punct

Die Facette `template<class charT> class money_punct` definiert Währungssymbole und die Formatierung. Sie erbt von der Klasse `money_base`, die die öffentlichen Elemente

```
enum part { none, space, symbol, sign, value};
struct pattern { char field[4]};
```

bereitstellt. Ein monetäres Format wird durch eine Folge von vier Komponenten spezifiziert, die in einem `pattern`-Objekt `p` zusammengefasst werden. Das Element `static_cast<part>(p.field[i])` bestimmt die *i*-te Komponente des Formats. Aus Effizienzgründen ist `field` vom Typ `char` anstatt vom Typ `part`. Im Feld eines `pattern`-Objekts kann eines der Elemente von `part` genau einmal vorkommen. Die Klasse `money_punct` hat die folgende öffentliche Schnittstelle:

- `charT decimal_point() const`
gibt den verwendeten Dezimalpunkt zurück.
- `charT thousands_sep() const`
gibt das Trennzeichen zwischen Tausender-Gruppen zurück.
- `string grouping() const`
Die Funktion hat dieselbe Bedeutung wie die gleichnamige Funktion der Klasse `num_punct` (Seite 833).
- `basic_string<charT> curr_symbol() const`
gibt das Währungssymbol zurück, z.B. \$. Für internationale Instanziierungen (vgl. Tabelle 31.1, Seite 830) werden im Allgemeinen drei Buchstaben und ein Leerzeichen zurückgegeben, z.B. »USD «.
- `basic_string<charT> positive_sign() const` und
`basic_string<charT> negative_sign() const`
geben das Zeichen für einen positiven Wert ('+' oder Leerzeichen) bzw. einen negativen Wert (meistens '-') zurück.
- `int frac_digits() const`
gibt die Ziffern nach dem Dezimalpunkt an, im Allgemeinen zwei.
- `pattern pos_format() const` und
`pattern pos_format() const`
geben das benutzte Formatierungsmuster zurück. Das Standardmuster ist `{symbol, sign, none, value}`.

Die Klasse kann für ein bestimmtes `locale`-Objekt wie folgt benutzt werden:

```
locale loc; // Kopie des aktuellen globalen locale-Objekts
char dezPunkt = use_facet<money_punct<char>>(loc).decimal_point();
```

money_get

Die Template-Klasse `template<class charT> class money_get` wickelt das formatierte Einlesen von Geldbeträgen, ggf. mit Währungsangaben, ab. Sie hat zwei öffentliche Methoden

- `iter_type get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err, long double& units) const`

```

■ iter_type get(iter_type s, iter_type end, bool intl,
               ios_base& f, ios_base::iostate& err,
               string_type& units) const

```

`iter_type` ist eine öffentliche, in der Klasse definierte Typbezeichnung für einen Input-Iterator, dessen Typ mit `istreambuf_iterator<charT>` vorgegeben ist. Dieser Typ wird nicht weiter beschrieben, weil erstens der Typ über den Namen `money_get::iter_type` benutzbar ist und er zweitens im Allgemeinen nicht alleinstehend benötigt wird, wie das Beispiel unten zeigt. `string_type` ist der in der Klasse definierte Name für den Typ `basic_string<charT>`. Diese Methoden lesen einen Geldbetrag als `double`-Zahl bzw. einen String ein, wobei der Dezimalpunkt eliminiert wird. Sie können in einer benutzerdefinierten Klasse zur Implementierung des Eingabeoperators (`>>`) verwendet werden. Zurückgegeben wird ein Iterator, der auf das unmittelbar nach dem letzten gültigen Zeichen eines Geldbetrags folgende Zeichen verweist. Im folgenden Beispiel wird Bezug auf die oben erwähnte Klasse `Geld` genommen.

Listing 31.5: Locale-abhängiger Eingabeoperator der Klasse `Geld`

```

// Auszug aus der Datei cppbuch/k31/geld/Geld.cpp
#include "Geld.h"
#include <locale>
Geld::Geld(long int b)
    :betrag(b) {
}
long int Geld::getBetrag() const {
    return betrag;
}

std::istream& operator>>(std::istream& is, Geld& geld) {
    std::istream::sentry s(is); // sentry siehe Seite 396
    if(s) {
        std::ios_base::iostate fehler = is.rdstate();
        is.setf(std::ios::showbase); // damit die Währung ausgewertet wird
        long double wieviel = 0;
        std::use_facet<std::money_get<char> >(is.getloc())
            .get(is, 0, false, is, fehler, wieviel);
        is.setstate(fehler);
        if(!fehler) {
            geld = Geld(static_cast<long int>(wieviel));
        }
        else {
            std::cerr << "fehlerhafte Eingabe!\n";
        }
    }
    return is;
}

```

Zwar ist die Basis der Klasse `Geld` ein ganzzahliger Cent-Betrag, die obige `put()`-Funktion verlangt jedoch `long double`. Aus diesem Grund wird die Typumwandlung `static_cast` eingesetzt. Das Beispiel zeigt, dass der `Istream` `is` an die Stelle des verlangten Input-Iterators treten kann. Der Grund liegt darin, dass die Klasse `istreambuf_iterator<charT>` einen Konstruktor hat, der ein `istream`-Objekt als Parameter nimmt. Der vierte Parameter

von `get()` nutzt aus, dass die Klasse `istream` von der Klasse `ios_base` erbt. Er dient dazu, intern über `getloc()` auf die Facette `money_punct` zuzugreifen. Das folgende Programmfragment zeigt eine Anwendung:

```
Geld dollars;
cin.imbue(locale("en_US"));
cin >> dollars;
```

Eine Zeichenfolge "1056.23" im Eingabestrom führt zu dem Ergebnis

```
dollars.betrag == 105623.
```

money_put

Die Template-Klasse `template<class charT> class money_put` wickelt die formatierte Ausgabe von Geldbeträgen, gegebenenfalls mit Währungsangaben, ab. Sie hat zwei öffentliche Methoden:

- `iter_type put(iter_type s, bool intl, ios_base& f, charT fill, long double& units) const`
- `iter_type put(iter_type s, bool intl, ios_base& f, charT fill, string_type& digits) const`

Die Typbezeichnungen entsprechen denen der Klasse `money_get`, wobei der Typ `iter_type` natürlich ein Output-Iterator ist. Anwendungsmöglichkeiten ergeben sich in Analogie zur Klasse `money_get`, zum Beispiel der Ausgabeoperator für die obige Klasse `Geld`:

Listing 31.6: Locale-abhängiger Ausgabeoperator der Klasse `Geld`

```
// Auszug aus der Datei cppbuch/k31/geld/Geld.cpp
// .. Fortsetzung von oben

std::ostream& operator<<(std::ostream& os, const Geld& geld) {
    std::ostream::sentry s(os);
    os.setf(std::ios::showbase); // damit die Währung angezeigt wird
    if(s) {
        std::use_facet<std::money_put<char> >(os.getloc())
            .put(os, true, os, ' ', static_cast<double>(geld.getBetrag()));
    }
    return os;
}
```

31.4.5 time

Diese Kategorie enthält zwei Klassen, die für die formatierte Ein- und Ausgabe von Zeiten und Datumsangaben gebraucht werden können.

time_get

Die Template-Klasse `template<class charT> class time_get` wickelt das formatierte Einlesen von Datumsangaben und Zeiten ab. Die Klasse erbt von der Klasse `time_base`, die den öffentlichen Typ `dateorder` zur Verfügung stellt:

```
enum dateorder { no_order, dmy, mdy, ymd, ydm}
```

Dieser Typ spezifiziert die möglichen Ordnungen (dmy = day month year usw.). Die Klasse `time_get` deklariert den Typ `iter_type` für einen Input-Iterator, dessen Typ mit `istreambuf_iterator<charT>` vorgegeben ist. Wie bei der Klasse `money_get` ist auch hier die genaue Typkenntnis nicht notwendig (siehe Beispiel unten). Die öffentlichen Methoden der Klasse `time_get` sind:

- `dateorder date_order() const`
liefert die verwendete Reihenfolge von Tag, Monat und Jahr.
- `iter_type get_time(iter_type s, iter_type end, ios_base& f,
ios_base::iostate& err, tm* t) const`
- `iter_type get_date(iter_type s, iter_type end, ios_base& f,
ios_base::iostate& err, tm* t) const`
- `iter_type get_weekday(iter_type s, iter_type end, ios_base& f,
ios_base::iostate& err, tm* t) const`
- `iter_type get_monthname(iter_type s, iter_type end,
ios_base& f, ios_base::iostate& err,
tm* t) const`
- `iter_type get_year(iter_type s, iter_type end, ios_base& f,
ios_base::iostate& err, tm* t) const`

Alle `get`-Methoden ermitteln über den Parameter `f` die verwendete Sprachumgebung und das Format. Der Typ `tm` ist auf Seite 881 beschrieben. Die Methoden lesen ab Position `s` alle Zeichen, die notwendig sind, die Struktur `*t` bezüglich der gewünschten Information (Zeit, Datum, Wochentag, Monatsname, Jahr) zu füllen bzw. bis ein Fehler auftritt. Zurückgegeben wird ein Iterator auf die Position direkt nach dem letzten Zeichen, das noch zu der gelesenen Information gehört. Das Beispiel zeigt, wie der Eingabeoperator für die Klasse `Datum` aus Abschnitt 9.3, Seite 334, realisiert werden kann.

Listing 31.7: Locale-abhängiger Eingabe-Operator

```
// Auszug aus der Datei cppbuch/k31/datum/datum.cpp
#include "datum.h" // Deklarationen und #include<iostream> dort nachtragen

std::istream& operator>>(std::istream& is, Datum& d) {
    std::istream::sentry s(is);
    if(s) {
        std::ios_base::iostate fehler = std::ios_base::goodbit;
        struct std::tm t;
        std::use_facet<std::time_get<char>>(is.getloc())
            .get_date(is, 0, is, fehler, &t);
        if (!fehler) {
            d = Datum(t.tm_mday, t.tm_mon + 1, t.tm_year + 1900);
        }
        is.setstate(fehler);
    }
    return is;
}
```

time_put

Die Template-Klasse `template<class charT> class time_put` wickelt die formatierte Ausgabe von Datumsangaben und Zeiten ab. Die Klasse hat den öffentlichen Typ `iter_type` für einen Output-Iterator und die folgenden öffentlichen Methoden:

- `iter_type put(iter_type s, ios_base& f, charT fill, const tm* tmb, const charT* pat, const charT* pat_end) const`
Diese Methode gibt die in der Struktur `tmb` liegende Zeit entsprechend einem Muster aus, das im Formatstring von `pat` bis `pat_end` vorliegt. Das Muster entspricht den für `strftime()` (Seite 882) üblichen Konventionen. `fill` ist ein Füllzeichen, zum Beispiel das Leerzeichen.
- `iter_type put(iter_type s, ios_base& f, charT fill, const tm* tmb, char format, char modifier = 0) const;`
Diese Methode gibt die in der Struktur `tmb` liegende Zeit entsprechend einem Muster aus, das im Zeichen `format` definiert ist. Dieses Zeichen ist eins der möglichen, die nach dem `'%'`-Zeichen im `strftime()`-Format vorkommen können. Der Parameter `modifier` ist implementationsabhängig.

Das folgende Beispiel zeigt, wie der Ausgabeoperator (vgl. Aufgabe auf Seite 337) und die Methode `toString()` (vgl. Aufgabe auf Seite 339) realisiert werden können.

Listing 31.8: Locale-abhängiger Ausgabe-Operator und `toString()`

```
// Auszug aus der Datei cppbuch/k31/datum/datum.cpp
std::ostream& operator<<(std::ostream& os, const Datum& d) {
    std::ostream::sentry s(os);
    if(s) {
        struct std::tm t;
        t.tm_mday = d.tag();
        t.tm_mon = d.monat()-1;
        t.tm_year = d.jahr()-1900;
        std::use_facet<std::time_put<char>>(os.getloc())
            .put(os, os, ' ', &t, 'x'); // x: siehe strftime
    }
    return os;
}

std::string Datum::toString(const std::locale& loc) const {
    std::ostringstream oss; // siehe Seite 393
    oss.imbue(loc);
    oss << *this; // Benutzung des obigen operator<<()
    return oss.str();
}
```

Eine kleines Beispiel demonstriert die Anwendung von `time_get` und `time_put` mit den neu definierten Ein- und Ausgabeoperatoren:

Listing 31.9: Datumsformate

```
// cppbuch/k31/datum/main.cpp
#include "datum.h"
using namespace std;
```

```
int main() {
    Datum einDatum;
    locale deDE("de_DE");
    cout << " bitte Datum im Format tt.mm.yyyy eingeben:" << endl;
    cin.imbue(deDE);
    cin >> einDatum;
    cout.imbue(deDE);
    cout << "deutsches Format : " << einDatum << endl;
    locale enUS("en_US");
    cout.imbue(enUS);
    cout << "US Format      : " << einDatum << endl;
    cout << "toString() mit Standard-Locale deDE:"
        << einDatum.toString() << endl;
    cout << "toString() mit Locale enUS      : "
        << einDatum.toString(enUS) << endl;
}
```

31.4.6 messages

Die Klasse `template<class charT> class messages` implementiert das Holen von Meldungen aus Katalogen. Wie ein Katalog realisiert ist, ob zum Beispiel als Datei oder Teil einer Datenbank, ist implementationsabhängig. Der Typ `catalog`, ein `int`-Typ, steht für eine Katalognummer. Es gibt die folgenden Elementfunktionen:

- `catalog open(const string& fn, const locale&) const`
eröffnet den Katalog, der durch den String `fn` identifiziert wird, und gibt eine Identifizierungszahl zurück, die bis zum folgenden `close()` zu dem Katalog gehört. Falls diese Zahl negativ ist, kann der Katalog nicht geöffnet werden.
- `void close(catalog c)`
schließt den Katalog `c`.
- `basic_string<charT> get(catalog c, int set, int msgid, const basic_string<charT>& vorgabe) const`
Es wird die durch die Argumente `set`, `msgid` und `vorgabe` identifizierte Meldung zurückgegeben. Falls keine Meldung gefunden wird, ist `vorgabe` das Ergebnis.

31.5 Konstruktion eigener Facetten

Man kann vorhandene Facetten durch eigene ersetzen. Dazu muss man wissen, dass zu allen Methoden der oben beschriebenen Facetten zusätzliche virtuelle Methoden mit exakt denselben Schnittstellen existieren, die ein vorangestelltes `do_` im Namen haben und `protected` sind. Diese Methoden werden von den oben beschriebenen aufgerufen. Ferner gibt es Klassen, die von den beschriebenen Facetten nur die `protected`-Schnittstelle erben und im Namen ein nachgestelltes `_byname` tragen, um auszudrücken, dass Namen für die Facetten vergeben werden können. Von diesen Klassen können eigene Klassen abgeleitet und die Methoden überschrieben werden. Das folgende Beispiel zeigt, wie das

vorgegebene Standardsymbol für Euro, nämlich EUR, mithilfe einer eigenen Facette für Währungssymbole durch das Symbol € ersetzt wird.

```
// cppbuch/k31/euro.cpp
#include <iostream>
#include <locale>
#include <string>
#include "Geld.h"
typedef std::moneypunct_byname<char, true> MeinMoneypunct;

class MeinWaehrungsformat : public MeinMoneypunct {
protected:
    // Redefinieren der virtuellen Funktion do_curr_symbol(), die von der
    // public-Funktion curr_symbol() der Basisklasse moneypunct gerufen wird:
    std::string do_curr_symbol() const {
        return wsymbol;
    }
public:
    MeinWaehrungsformat(const std::locale& loc, const char* ws)
        : MeinMoneypunct(loc.name().c_str()), wsymbol(ws) {}
private:
    const char* wsymbol;
};

using namespace std;

int main() {
    locale locUS("en_US");
    Geld derBetrag;
    cout << "Eingabe in Cent(!), z.B. 123456:?">
    cin >> derBetrag;
    cout << "direkte Abfrage mit voreingestellter locale ("
        << locale().name() << ") :>                               // locale 'C'
        << derBetrag.getBetrag() << endl;                             // 123456
    cout.imbue(locUS); // cout auf enUS umschalten
    cout << "Es wurde " << derBetrag
        << " eingegeben (US-Format).\n";                               // USD 1,234.56
    locale deDEeuro("de_DE@euro");
    cout << "Ausgabe Standard-Währungssymbol EUR und Dezimalkomma "
        << "statt -punkt :> << endl;

    // Achtung: KEINE Währungsumrechnung!
    cout.imbue(deDEeuro); // cout auf deDE@euro umschalten
    cout << derBetrag << endl; // 1.234,56 EUR
    MeinWaehrungsformat* mwptr = new MeinWaehrungsformat(deDEeuro,
        "\u20ac"); // Euro-Symbol in UTF-8
        // \xA4 in ISO-8859-15
    cout << "Ausgabe eigenes Währungssymbol und Dezimalkomma "
        << " statt Dezimalpunkt :> << endl;
    cout.imbue(locale(deDEeuro, mwptr));
    cout << derBetrag << endl; // 1.234,56 €
}
```

32

String

Dieses Kapitel behandelt die folgenden Themen:

- Konstruktor und Basismethoden
- Anhängen, Einfügen, Ersetzen
- Suchen und Finden
- Numerische Umwandlungen

Die C++-Stringklasse (Header `<string>`) kann in den meisten Fällen die C-Strings ersetzen. Sie ist nicht nur verständlicher in der Anwendung für Neulinge, sondern auch erheblich komfortabler. Der in der Standardbibliothek definierte Typ `basic_string` ist ein Template und ermöglicht das Arbeiten mit verschiedenen Arten von Zeichen, also auch mit »wide characters« (Typ `wchar_t`). `wstring` ist die Spezialisierung für `wchar_t`. Der Typ `string` ist die Spezialisierung von `basic_string` für den Datentyp `char`:

```
typedef basic_string<char> string;
```

Die im Folgenden beschriebene Schnittstelle der Klasse `basic_string` bezieht sich der Kürze wegen nur auf die Spezialisierung `string`. Die von `string` bereitgestellten öffentlichen Datentypen korrespondieren mit denen der Tabelle [28.2](#) auf Seite [765](#). Der Typ `string::size_type` wird im Folgenden kurz `size_type` genannt. Zunächst folgen die Konstruktoren sowie diejenigen Methoden, die in ähnlicher Form auch in anderen Containern der C++-Standardbibliothek auftreten.

- `string()` ist der Standardkonstruktor. Er erzeugt einen leeren String.
- `string(const string& s, size_type pos = 0, size_type n = string::npos)`
Der Kopierkonstruktor erzeugt einen String, wobei `s` ab Position `pos` bis zum Ende kopiert wird. Dabei gilt die Einschränkung, dass maximal `n` Zeichen kopiert werden. `string::npos` ist eine `-1` konvertiert in den unsigned-Typ `size_type`, also die größtmögliche unsigned-Zahl.
- `string(const char* s, size_type n)`
Bei der Erzeugung des Strings werden `n` Zeichen aus dem bei `s` beginnenden Array kopiert.
- `string(const char* s)` erzeugt String aus dem C-String `s`.
- `string(size_type n, char c)` erzeugt String mit `n` Kopien von `c`.
- `template<InputIterator> string(InputIterator a, InputIterator b)`
Falls `InputIterator` ein integraler Typ ist, entspricht dieser Konstruktor dem vorhergehenden (`string(size_type n, char c)`), wobei `a` und `b` in die entsprechenden Typen `size_type` und `char` umgewandelt werden. Andernfalls wird der String aus den Zeichen im Intervall `[a, b)` gebildet.
- `~string()` Destruktor
- `const_iterator begin() const` und `iterator begin()`
geben den Anfang des Strings zurück.
- `const_iterator end() const` und `iterator end()`
geben die Position *nach* dem letzten Zeichen zurück.
- `const_iterator rbegin() const` und `iterator rbegin()`
geben einen Iterator zurück, der auf das letzte Zeichen zeigt.
- `const_iterator rend() const` und `iterator rend()`
geben einen Iterator zurück, der auf die Position vor dem Anfang zeigt.
- `size_type size() const`
gibt die aktuelle Größe des Strings zurück (Anzahl der Zeichen).
- `size_type length() const` ist dasselbe wie `size()`.
- `resize(size_type n, char c = '\0')`
Der String wird durch eine auf `n` Zeichen verkürzte Kopie ersetzt, falls `n ≤ size()` ist. Andernfalls wird der String durch eine auf `n` Zeichen vergrößerte Kopie ersetzt, wobei die restlichen Elemente mit `c` initialisiert werden.
- `void reserve(size_type n = 0)`
Speicherplatz reservieren, sodass der verfügbare Platz (Kapazität) größer als der aktuell benötigte ist. Zweck: Vermeiden von Speicherbeschaffungsoperationen während der Benutzung des Strings.
- `void shrink_to_fit()`
Wenn keine weiteren Operationen mit dem String mehr zu erwarten sind, reduziert der Aufruf dieser Funktion den Speicherplatz auf das Notwendige.

- `size_type capacity() const`
gibt die Größe des dem String zugewiesenen Speichers zurück. Der Wert ist größer oder gleich dem Argument von `reserve()`, falls `reserve()` vorher aufgerufen wurde.
- `void clear()` löscht den Inhalt des Strings; entspricht `erase(begin(), end())`.
- `bool empty() const` gibt `size() == 0` bzw. `begin() == end()` zurück.
- `const_reference operator[](size_type n) const` und
`reference operator[](size_type n)`
geben eine Referenz auf das n-te Zeichen zurück.
- `const_reference at(size_type n) const` und
`reference at(size_type n)`
geben eine Referenz auf das n-te Zeichen zurück, wobei die Gültigkeit des Arguments geprüft wird. Es wird eine `out_of_range`-Exception geworfen, falls $n \geq \text{size}()$ ist.
- `size_type copy(char* z, size_type n, size_type pos = 0) const`
überschreibt das Array `z` ab `pos` mit den Zeichen des Strings, aber maximal `n`. Das Stringendezeichen wird nicht kopiert. Es wird vorausgesetzt, das `z` auf einen Bereich mit genug Platz verweist. Die Methode gibt die Anzahl der kopierten Zeichen zurück.
- `void swap(const string& s)` vertauscht den Inhalt der beiden Strings.
- `const char* c_str()` und
`const char* data()`
geben einen Zeiger `z` auf das erste der intern gespeicherten Zeichen zurück. Für alle weiteren Positionen `i` im Bereich `[0, size())` gilt entsprechend `p + i = &operator[](i)`.
- `const char& front() const` und
`char& front()`
geben eine Referenz auf das erste Zeichen zurück, d.h. `operator[](0)`.
- `const char& back() const` und
`char& back()`
geben eine Referenz auf das letzte Zeichen zurück, d.h. `operator[](size()-1)`.

Interessanter sind die Methoden, die in anderen Containern nicht vertreten und speziell zur Bearbeitung von Zeichenketten geeignet sind, zum Beispiel Finden eines Substrings. Die wichtigsten sind im Folgenden aufgeführt.

Zuweisen und Anhängen

- `string& append(const string& s)`
`string& append(const char* s)`
`string& operator+=(const string& s)`
`string& operator+=(const char* s)`
`string& operator+=(char c)`
verlängern den String um den C-String oder String `s` bzw. das Zeichen `c`.
- `string& append(const string& s, size_type pos, size_type n)`
Von der Position `pos` des Strings `s` bis zum Ende wird alles an den String angehängt, aber nicht mehr als `n` Zeichen.
- `string& append(const char* s, size_type n)` verlängert den String um `string(s,n)`.
- `string& append(size_type n, char c)` verlängert den String um `string(n,c)`.

- `void push_back(char c)` bewirkt dasselbe wie `append(1, c)`.
- `string& assign(const string& s)`
`string& assign(const char* s)`
`string& operator=(const string& s)`
`string& operator=(const char* s)`
`string& operator=(char c)`
 weisen dem String den C-String oder String `s` bzw. das Zeichen `c` zu.
- `string& assign(const string& s, size_type pos, size_type n)`
 Dem String wird der String `s` von der Position `pos` des Strings `s` an bis zum Ende zugewiesen, aber nicht mehr als `n` Zeichen. Die vorher beschriebene Funktion `assign(s)` entspricht `assign(s, 0, string::npos)`.

Einfügen

- `string& insert(size_type pos, const char* s)`
`string& insert(size_type pos, const string& s)`
 C-String bzw. String `s` vor der Stelle `pos` einfügen (das heißt am Anfang, falls `pos` gleich 0).
- `string& insert(size_type pos1, const string& s, size_type pos2, size_type n)`
 Vor die Stelle `pos1` wird der String `s` eingefügt, wobei an der Position `pos2` von `s` begonnen wird und insgesamt nicht mehr als `n` Zeichen kopiert werden.
- `string& insert(size_type pos, const char* s, size_type n)`
 wirkt wie `insert(pos, string(s,n))`.
- `string& insert(size_type pos, size_type n, char c)`
 wirkt wie `insert(pos, string(n,c))`.
- `template<InputIterator>`
`void insert(iterator p, InputIterator first, InputIterator last)`
`p` ist ein Iterator des Strings selbst, `first` und `last` sind Iteratoren eines anderen Strings oder Containers. Die Funktion bewirkt das Einfügen der Zeichen im Bereich `[first, last)` vor der Stelle `p`.
- `iterator insert(iterator p, char c)`
`p` ist ein Iterator des Strings selbst. `c` wird vor der Stelle `p` eingefügt.
- `iterator insert(iterator p, size_type n, char c)`
`p` ist ein Iterator des Strings selbst. `n` Kopien von `c` werden vor der Stelle `p` eingefügt.

Löschen und Ersetzen

- `iterator erase(iterator p)`
 Zeichen an der Stelle `p` löschen. Zurückgegeben wird die Position direkt vorher, sofern sie existiert (andernfalls `end()`).
- `iterator erase(iterator p, iterator q)`
 Zeichen im Bereich `p` bis ausschließlich `q` löschen.
- `string& erase(size_type pos = 0, size_type n = string::npos)`
 löscht alle Zeichen ab der Stelle `pos`, aber nicht mehr als `n` Zeichen.

- `string& replace(size_type pos1, size_type n1, const string& s, size_type pos2, size_type n2)`
Alle Zeichen des Strings ab der Stelle `pos1`, aber maximal `n1` Zeichen, werden entfernt. An dieser Stelle werden alle Zeichen des Strings `s` ab der Stelle `pos2`, aber maximal `n2` Zeichen, eingefügt.
- `string& replace(size_type pos, size_type n, const string& s)`
`string& replace(size_type pos, size_type n, const char* s)`
Alle Zeichen des Strings ab der Stelle `pos`, aber maximal `n` Zeichen, werden entfernt. An dieser Stelle werden alle Zeichen des Strings bzw. C-Strings `s` eingefügt.
- `string& replace(size_type pos, size_type n1, const char* s, size_type n2)`
wirkt wie `replace(pos, n1, string(s, n2))`.
- `string& replace(size_type pos, size_type n1, size_type n2, char c)`
wirkt wie `replace(pos, n1, string(n2, c))`.
- `string& replace(iterator p, iterator q, const string& s)`
`string& replace(iterator p, iterator q, const char* s)`
Bereich zwischen `p` und ausschließlich `q` durch `s` ersetzen.
- `string& replace(iterator p, iterator q, const char* s, size_type n)`
wirkt wie `replace(p, q, string(s, n))`.
- `string& replace(iterator p, iterator q, size_type n, char c)`
wirkt wie `replace(p, q, string(n, c))`.
- `template<class InputIterator>`
`string& replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2)`
wirkt wie `replace(i1, i2, string(j1, j2))`.

Suchen und Finden

- `size_type find(const string& s, size_type pos = 0) const`
gibt die Position zurück, an der der Substring `s` gefunden wird, andernfalls wird `string::npos` zurückgegeben. Gesucht wird ab `pos`. `find()` findet die erste Position bei mehrfachem Vorkommen von `s`.
- `size_type find(const char* s, size_type pos, size_type n) const`
gibt `find(string(s, n), pos)` zurück.
- `size_type find(const char* s, size_type pos = 0) const`
gibt `find(string(s), pos)` zurück.
- `size_type find(char c, size_type pos = 0) const`
gibt `find(string(1,c), pos)` zurück.
- `size_type rfind(const string& s, size_type pos = string::npos) const`
Gibt die Position zurück, an der der Substring `s` gefunden wird, andernfalls wird `string::npos` zurückgegeben. Gesucht wird ab `pos`. `rfind()` findet die letzte Position bei mehrfachem Vorkommen von `s`.
- `size_type rfind(const char* s, size_type pos, size_type n) const`
gibt `rfind(string(s, n), pos)` zurück.
- `size_type rfind(const char* s, size_type pos = string::npos) const`
gibt `rfind(string(s), pos)` zurück.

- `size_type rfind(char c, size_type pos = string::npos) const`
gibt `rfind(string(1,c), pos)` zurück.
- `size_type find_first_of(const string& s, size_type pos=0) const` und
`size_type find_first_of(const char* s, size_type pos = 0) const`
geben die erste Position zurück, an der ein Zeichen gefunden wird, das auch im String bzw. C-String `s` vorhanden ist, andernfalls wird `string::npos` zurückgegeben. Gesucht wird ab `pos`.
- `size_type find_first_of(const char* s, size_type pos, size_type n) const`
gibt `find_first_of(string(s,n), pos)` zurück.
- `size_type find_first_of(char c, size_type pos = 0) const`
gibt `find_first_of(string(1,c), pos)` zurück.
- `size_type find_last_of(const string& s, size_type pos = string::npos) const`
`size_type find_last_of(const char* s, size_type pos = string::npos) const`
`size_type find_last_of(const char* s, size_type pos, size_type n) const`
`size_type find_last_of(char c, size_type pos = string::npos) const`
Diese Funktionen entsprechen `find_first_of()` mit dem Unterschied, dass jeweils die letzte gefundene Position zurückgegeben wird.
- `size_type find_first_not_of(const string& s, size_type pos = 0) const`
`size_type find_first_not_of(const char* s, size_type pos = 0) const`
`size_type find_first_not_of(const char* s, size_type pos, size_type n) const`
`size_type find_first_not_of(char c, size_type pos = 0) const`
Diese Funktionen entsprechen `find_first_of()` mit dem Unterschied, dass jeweils die erste Position zurückgegeben wird, an der ein Zeichen steht, das *nicht* in `s` vorkommt bzw. das nicht `c` entspricht. Wenn so eine Position nicht gefunden wird, wird `string::npos` zurückgegeben.
- `size_type find_last_not_of(const string& s, size_type pos = string::npos) const`
`size_type find_last_not_of(const char* s, size_type pos = string::npos) const`
`size_type find_last_not_of(const char* s, size_type pos, size_type n) const`
`size_type find_last_not_of(char c, size_type pos=string::npos) const`
Diese Funktionen entsprechen `find_first_not_of()` mit dem Unterschied, dass jeweils die letzte gefundene Position zurückgegeben wird.

Substrings und Vergleiche

- `string substr(size_type pos = 0, size_type n = string::npos) const`
Gibt den Substring zurück, der ab `pos` beginnt. Die Anzahl der Zeichen im Substring wird durch das Ende des Strings bestimmt, kann aber nicht größer als `n` werden.
- `int compare(const string& s) const`
vergleicht zeichenweise die Strings `*this` und `s`. Es wird 0 zurückgegeben, wenn keinerlei Unterschied festgestellt wird. Falls das erste unterschiedliche Zeichen von `*this` kleiner als das entsprechende in `s` ist, wird eine negative Zahl zurückgegeben, andernfalls eine positive. Falls bei unterschiedlicher Länge bis zum Ende eines der Strings keine verschiedenen Zeichen gefunden werden, wird eine negative Zahl zurückgegeben, falls `size() < s.size()` ist, andernfalls eine positive Zahl.

- `int compare(size_type pos, size_type n, const string& s) const`
gibt `string(*this, pos, n).compare(s)` zurück. Das heißt: Es werden nur die Zeichen ab Position `pos` in `*this` berücksichtigt, aber maximal `n`.
- `int compare(size_type pos1, size_type n1, const string& s, size_type pos2, size_type n2) const`
gibt `string(*this, pos1, n1).compare(string(s, pos2, n2))` zurück, Das heißt, es werden nur die Zeichen ab Position `pos1` in `*this` berücksichtigt, aber maximal `n1`. In `s` werden nur die Zeichen ab Position `pos2` berücksichtigt, aber maximal `n2`.

Numerische Umwandlungen

Die Funktionen zur Umwandlung eines Strings in eine Zahl bzw. umgekehrt sind keine Methoden der Klasse `string`, sondern im Namespace `std` [ISOC++].

```
int stoi(const string& str, size_t *idx = 0, int base = 10)
long stol(const string& str, size_t *idx = 0, int base = 10)
unsigned long stoul(const string& str, size_t *idx = 0, int base = 10)
long long stoll(const string& str, size_t *idx = 0, int base = 10)
unsigned long long stoull(const string& str, size_t *idx = 0, int base = 10)
float stof(const string& str, size_t *idx = 0)
double stod(const string& str, size_t *idx = 0)
long double stold(const string& str, size_t *idx = 0)
```

Ein Anwendungsbeispiel finden Sie auf Seite 625. Die Funktionen benutzen intern die C-Funktionen `strtol()`, `strtod()` usw., deren Wirkungsweise auf Seite 625 beschrieben wird. `base` ist die gewünschte Zahlenbasis. Falls der Zeiger `idx` ungleich 0 ist, wird der Index des ersten nicht umgewandelten Zeichens in `*idx` abgelegt. Eine Alternative zu diesen Funktionen ist `boost::lexical_cast<T>(arg)` (Seite 627). Zur Umwandlung von Zahlen in einen String gibt es die Funktionen

```
string to_string(X)
```

wobei `X` für einen der Zahl-Typen `int`, `float`, `double` usw. steht, einschließlich der `unsigned` und `long`-Varianten.

Binäre Operatoren

Darüber hinaus gibt es einige Funktionen, die mit Strings arbeiten, aber *keine* Element-funktionen sind:

- `string operator+(const string&, const string&)`
`string operator+(const string&, const char*)`
`string operator+(const char*, const string&)`
Diese Operatoren verketten zwei Strings (bzw. einen String und einen C-String oder umgekehrt) und geben das Ergebnis zurück.
- `string operator+(const string&, char)`
`string operator+(char, const string&)`
Diese Operatoren verketten einen String mit einem Zeichen und geben das Ergebnis zurück.
- `bool operator==(X, Y)`
`bool operator!=(X, Y)`

```
bool operator<=(X, Y)
bool operator>=(X, Y)
bool operator<(X, Y)
bool operator>(X, Y)
```

sind die relationalen Operatoren zum Vergleichen von Strings. X und Y stehen hier für jeweils einen der Typen `const string&` oder `const char*`. Es sind drei Kombinationen für X und Y möglich:

```
const string&, const string&
const char*, const string&
const string&, const char*
```

- `istream& operator>>(istream&, string&)`

Dieser Operator erlaubt das Einlesen von Strings auf bequeme Weise. Die üblichen Eigenschaften des `>>`-Operators werden beibehalten (vgl. Seite [94](#)).

- `ostream& operator<<(ostream&, string&)` Ausgabeoperator für Strings.

- `istream& getline(istream& is, string& s, char ende = '\\n')`

Liest Zeichen für Zeichen aus der Eingabe `is` in den String `s`, bis das Zeichen `ende` gelesen wird. `ende` wird zwar gelesen, aber *nicht* an den String angehängt (vergleiche `getline()` auf Seite [382](#)).

33

Speichermanagement

Dieses Kapitel behandelt die folgenden Themen:

- Smart Pointer der Standardbibliothek
- `new` mit vorgegebenem Speicherort
- Hilfsfunktionen

33.1 Smart Pointer `unique_ptr`, `shared_ptr`, `weak_ptr`

Die Wirkungsweise eines Smart Pointers wird in Abschnitt [9.5](#) beschrieben. Hier geht es um die verschiedenen Realisierungen der C++-Standardbibliothek.

`unique_ptr`

Die Klasse `unique_ptr` verhält sich wie die Klasse `SmartPointer` des Abschnitts [9.5](#), hat aber zusätzliche Funktionen. So kann zum Beispiel ein Objekt übergeben werden, das die Zerstörung anstelle des normalen Destruktors übernimmt. Da der Letztere im Allgemeinen genügt, wird hier auf eine Darstellung verzichtet. Ein einfaches Beispiel:

Listing 33.1: unique_ptr

```
// cppbuch/k33/uniqueptr/main.cpp
#include<iostream>
#include<memory>
using namespace std;

class Ressource {
public:
    Ressource(int i)
        : id(i){
        cout << "Konstruktor Ressource()" << endl;
    }
    void hi() const {
        cout << "hier ist Ressource::hi (), Id=" << id << endl;
    }
    ~Ressource() {
        cout << "Ressource::~Destruktor, Id=" << id << endl;
    }
private:
    int id;
};

int main() {
    cout << "Zeiger auf dynamisches Objekt:" << endl;
    unique_ptr<Ressource> p1(new Ressource(1));
    cout << "Operator -> ";
    p1->hi();
    cout << "Operator * ";
    (*p1).hi();
    // Null-Zeiger
    unique_ptr<Ressource> nullp((Ressource*)0);
    // nullp->hi(); // Speicherzugriffsfehler!
}
```

Ausgewählte Methoden:

- `unique_ptr()` erzeugt ein Objekt, das nichts enthält, gleichbedeutend mit der Anweisung `unique_ptr<Ressource> nullp((Ressource*)0);` im obigen Programm.
- `operator->()` gibt den Zeiger auf das enthaltene Objekt zurück.
- `operator*()` gibt eine Referenz auf das enthaltene Objekt zurück, d.h. `*operator->()`.
- `get() const` gibt genau wie `operator->()` den Zeiger auf das enthaltene Objekt zurück.
- `operator bool() const` gibt `get() != NULL` zurück.
- `reset(ptr)` setzt den internen Zeiger auf `ptr`. Der Destruktor für das möglicherweise vorher enthaltene Objekt wird aufgerufen.
- `release()` setzt den internen Zeiger auf `NULL`. Achtung: Der Destruktor für das möglicherweise vorher enthaltene Objekt wird *nicht* aufgerufen! Daher wird es im Allgemeinen besser sein, `reset(NULL)` zu nehmen..



Zur Verwendung von `unique_ptr` für Arrays siehe Abschnitt 20.3.3, Seite 568.

shared_ptr

Die Klasse `shared_ptr` implementiert eine Benutzungszählung. Damit können mehrere Objekte dieser Klasse auf ein Objekt (im Folgenden zur Unterscheidung Ressource genannt) verweisen. Ein `shared_ptr`-Objekt speichert die Adresse einer mit `new` erzeugten Ressource:

```
class X {};
shared_ptr<X> p1(new X);
```

Wenn ein weiteres `shared_ptr`-Objekt mit derselben Adresse initialisiert wird, erhöht sich der interne Benutzungszähler:

```
shared_ptr<X> p2(p1);
cout << p1.use_count() << endl; // 2
```

Der Destruktor der Klasse `shared_ptr` zählt den Benutzungszähler um eins herunter. Der Destruktor des letzten auf die Ressource verweisenden `shared_ptr`-Objekts ist für ihre Zerstörung verantwortlich. Man sagt auch, dass das `shared_ptr`-Objekt die Ressource *besitzt*. Das folgende Beispiel zeigt diese und weitere Eigenschaften von `shared_ptr`:

Listing 33.2: Beispiel mit SmartPointer-Objekten

```
// cppbuch/k33/sharedptr/main.cpp
#include<memory>
#include<iostream>
using namespace std;

class Ressource {
public:
    Ressource(int i)
        : id(i){
    }
    void hi() const {
        cout << "hier ist Ressource::hi (), Id=" << id << endl;
    }

    ~Ressource() {
        cout << "Ressource::Destruktor, Id=" << id << endl;
    }
private:
    int id;
};

int main() {
    cout << "Konstruktoraufruf" << endl;
    shared_ptr<Ressource> p1(new Ressource(1));

    cout << "Operator -> "; p1->hi();
    cout << "Operator * "; (*p1).hi();
    cout << "Benutzungszähler: " << p1.use_count() << endl; // 1

    { // Blockanfang
        // zweiter shared_ptr für dasselbe Objekt
```



```

    shared_ptr<Ressource> p2(p1);
    cout << "Benutzungszähler p1: " << p1.use_count() << endl; // 2
    cout << "Benutzungszähler p2: " << p2.use_count() << endl; // 2
    p2->hi();
} // p2 wird zerstört
cout << "Benutzungszähler p1: " << p1.use_count() << endl; // 1
cout << "Objekt existiert noch: ";
p1->hi();
// Zuweisung
shared_ptr<Ressource> p3(new Ressource(3));
p3 = p1; // Ressource 3 wird freigegeben (delete), danach
        // verweisen beide auf das Objekt *p1
p1->hi();
p3->hi();
// Null-Zeiger
shared_ptr<Ressource> nullp((Ressource*)0);
// nullp->hi(); // Speicherzugriffsfehler!
} // p3 und p1 werden zerstört

```

Die fehlerhafte Nutzung mit einem Null-Zeiger, wie am Ende gezeigt, führt zum Programmabbruch. Bei richtiger Handhabung entsprechend dem unten beschriebenen Tipp tritt dieser Fall nicht auf. An der abschließenden `}`-Klammer werden die Destruktoren der noch verbliebenen Objekte `p3` und `p1` ausgeführt. Nur der zuletzt ausgeführte Destruktor löscht das referenzierte `Ressource`-Objekt. Container der Standardbibliothek (siehe Kapitel 28) können statt Zeigern `shared_ptr`-Objekte enthalten:

```

// STL-Container mit shared_ptr
vector<shared_ptr<Ressource> > vec(10);
vec.push_back(p3); // p3 von oben
vec[1] = shared_ptr<Ressource>(new Ressource(4));
vec[1]->hi();

```



Tipp 1

Wenn Sie dynamische Objekte erzeugen, verwenden Sie `shared_ptr`. Über die Zerstörung mit `delete` an einer geeigneten Stelle müssen Sie sich keine Gedanken mehr machen. Die Erzeugung des Zeigers mit `new` muss innerhalb der Parameterliste geschehen (Begründung siehe Abschnitt 20.3.1 auf Seite 567).



Tipp 2

Wenn Sie `shared_ptr` für Arrays einsetzen, müssen Sie eine Hilfsklasse zur Vermeidung von Memory-Leaks schreiben. Die Einzelheiten finden Sie in Abschnitt 20.3.2, Seite 567.

weak_ptr

weak_ptr-Objekte sind für Objekte gedacht, die bereits von shared_ptr-Objekten verwaltet werden. Der Konstruktor:

```
template<class T>
weak_ptr(const shared_ptr<T>& ptr);
```

Der Unterschied zu shared_ptr ist, dass ein weak_ptr kopiert und zugewiesen werden kann. Der weak_ptr-Destruktor hat keine Wirkung auf das enthaltene Objekt. Im Gegensatz zu einem shared_ptr besitzt ein weak_ptr keine Ressource, er verweist nur auf sie. weak_ptr ist für Container geeignet. Der Sinn von weak_ptr-Objekten ist es, zyklische Datenstrukturen unterbrechen zu können. Wenn in einer zyklischen Datenstruktur ein Knoten auf den nächsten mit einem shared_ptr verweist, kann der Benutzungszähler bei keinem Knoten null werden, d.h. der Destruktor wird nicht aufgerufen. Ein einfaches Beispiel:

```
// cppbuch/k33/weakptr/main.cpp
#include <iostream>
#include <memory>
using namespace std;

struct ZyklStruktur { // Konstruktor für das Beispiel nicht erforderlich
    ~ZyklStruktur() {
        cout << "Destruktor ~ZyklStruktur() aufgerufen" << endl;
    }
    weak_ptr<ZyklStruktur> nachbar; // *** siehe Text
};

void f() {
    ZyklStruktur* a1 = new ZyklStruktur;
    ZyklStruktur* a2 = new ZyklStruktur;
    // zyklische Struktur (gegenseitige Verweise) herstellen:
    a1->nachbar = shared_ptr<ZyklStruktur>(a2);
    a2->nachbar = shared_ptr<ZyklStruktur>(a1);
}

int main() {
    f();
}
```

Der Destruktor wird für die Objekte *a1 und *a2 aufgerufen – es gibt kein Problem. Wenn aber die ***-markierte Zeile durch shared_ptr<ZyklStruktur> nachbar; ersetzt würde, blieben sie, wenn main() nicht beendet würde, unerreichbar auf dem Heap! weak_ptr besitzt die folgenden Methoden:

- long use_count() gibt sptr.use_count() zurück, wobei sptr das bei der Konstruktion verwendete shared_ptr-Objekt ist. Falls sptr == NULL ist, wird 0 zurückgegeben.
- bool expired() gibt use_count() == 0 zurück.
- shared_ptr<T> lock() gibt das zugeordnete shared_ptr-Objekt zurück, falls vorhanden, andernfalls shared_ptr().

33.2 new mit vorgegebenem Speicherort

Der Header `<new>` enthält die Operatoren `new`, `new[]`, `delete` und `delete[]`, die in Abschnitt 5.4 ab Seite 200 beschrieben werden. Die Darstellung des Funktionszeigers `new_handler` und der zugehörigen Funktion `set_new_handler` zur Fehlerbehandlung sowie die Klasse `bad_alloc` finden sich in Abschnitt 8.2 ab Seite 312. Hier wird deshalb nur eine im Header `<new>` vorhandene weitere Form des `new`-Operators beschrieben, die »Placement-Form«.

```
// Placement-Form für new
void* operator new (std::size_t size, void *ptr);
void* operator new[](std::size_t size, void *ptr);
```

Zurückgegeben wird `ptr`. Die Placement-Operatoren dürfen nicht durch eigene mit derselben Signatur ersetzt werden. Die Placement-Form ist nützlich, wenn die Adresse, an der ein Objekt abgelegt werden soll, schon vorher bekannt ist. Bei vielen Objekten kann dies durchaus Zeit sparen, weil das normale `new` erst das Betriebssystem um Speicher ersucht.

Die entsprechenden Placement-`delete`-Operatoren gibt es nur der Form halber. Sie bewirken nichts, weil durch ein Placement-`new` kein neuer Speicher zugewiesen wird, sondern nur Objekte in einem vorhandenen Speicher angelegt werden. Es muss also kein Speicher freigegeben werden. Wenn Sie allerdings das Placement-`new` für eine Klasse überladen, sollten Sie auch das zugehörige Placement-`delete` schreiben. Zur Begründung siehe [ScM]. Ein Beispiel für das Placement-`new`:

```
// cppbuch/k33/placement/placement.cpp
#include<iostream>
#include<new>
using namespace std;

class Irgendwas {
public:
    Irgendwas() : id(++maxid) { }
    void machwas() const {
        cout << "Id = " << id << endl;
    }
private:
    int id;
    static int maxid;
};

int Irgendwas::maxid = 0;

int main() {
    char vielPlatz[1000*sizeof(Irgendwas)] = {0}; // mit 0 initialisieren

    // Ein Objekt in vielPlatz anlegen:
    Irgendwas* p = new (vielPlatz) Irgendwas; // Objekt 1
    p->machwas();
```

```
// Weitere 10 Objekte mit Array-Operator anlegen:
char* naechsteAdresse = (char*)p + sizeof(Irgendwas);
new (naechsteAdresse) Irgendwas[10]; // Objekte 2 bis 11

// Alle 11 Objekte abfragen (1-11)
for(int i = 0; i < 11; ++i) {
    cout << i << ": ";
    p++->machwas();
}

p->machwas(); // Ausgabe 0!
}
```

Die nächste Position ist nicht belegt, das heißt, der letzte Aufruf `p->machwas()` gibt 0 aus, weil das Feld `vielPlatz` mit 0 initialisiert wurde. `p` zeigt auf einen Bereich, in dem gar kein Objekt des Typs `Irgendwas` angelegt wurde. Nichtsdestoweniger wird der Zeiger `p` so interpretiert, als zeige er auf ein Objekt. Die Daten sind aber alle 0.

33.3 Hilfsfunktionen

Im Header `<memory>` sind unter anderem die folgenden Klassen und Funktionen vertreten:

- `template<typename T> class allocator`
Die `allocator`-Klasse stellt die Dienstleistungen bereit, die zur Beschaffung von Speicherplatz notwendig sind. In Abhängigkeit vom Memory-Modell wird ein passender Allokator vom System bereitgestellt, sodass sich ein Anwender nicht darum kümmern muss, es sei denn, er möchte selbst spezielle Memory-Funktionen realisieren, zum Beispiel eine Speicherverwaltung mit garbage collection. Solche Aufgabenstellungen sind recht speziell, sodass auf eine Beschreibung hier verzichtet wird. Wer einen Allokator selbst schreiben möchte, sei auf [\[Alex\]](#) verwiesen.
- `template<class OutputIterator, class T>`
`class raw_storage_iterator`
Die Klasse ermöglicht es, Daten in nicht-initialisierten Speicher zu schreiben. Es wird auf die Beschreibung in [\[ISOC++\]](#) verwiesen.
- `template<typename T>`
`pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n)`
Diese Funktion beschafft temporären Speicher. Zurückgegeben wird ein Paar, das die Speicheradresse und den verfügbaren Platz in Einheiten von `sizeof(T)` enthält, das heißt, `n` bei Erfolg und 0, falls die Speicherbeschaffung nicht gelingt.
- `template<typename T> void return_temporary_buffer(T* p)`
Diese Funktion gibt einen mit `get_temporary_buffer()` an der Stelle `p` beschafften Speicher wieder frei.

```

■ template<class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(InputIterator first,
InputIterator last, ForwardIterator result)

```

Diese Funktion kopiert alle Werte des Bereichs `[first, last)` nach `result`. Beispiel:

```

// cppbuch/k33/placement/uninitcopy.cpp
#include<algorithm>
#include<iostream>
#include<new>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
    vector<int> v1(10), v2(10);
    fill(v1.begin(), v1.end(), 999); // v1 mit Daten füllen
    // Jetzt v1 nach v2 kopieren:
    uninitialized_copy(v1.begin(), v1.end(), v2.begin());
    showSequence(v2);
}

```

Voraussetzung ist, dass der `operator*()` eines Iterators ein Objekt zurückgibt, dessen Adressoperator definiert ist und der einen Zeiger auf das Objekt zurückgibt. Das ist bei einem `vector<T>` der Fall, sofern `T` nicht `bool` ist. An diese Stelle wird der Speicherinhalt mit dem von oben bekannten Placement-new kopiert. Die Wirkung ist

```

for (; first != last; ++result, ++first) {
    new (static_cast<void*>(&*result))           // Ziel-Speicherplatz
        // abzulegendes Objekt:
        typename iterator_traits<ForwardIterator>::value_type(*first);
}

```

```

■ template<class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x)

```

Diese Funktion füllt alle Positionen im Bereich `[first, last)` mit Kopien von `x`. Beispiel mit dem Vektor `v1` von oben:

```

int Wert = 17;
uninitialized_fill(v1.begin(), v1.end(), Wert);

```

```

■ template<class ForwardIterator, class Size, class T>
void uninitialized_fill_n(ForwardIterator first, Size n,
const T& x)

```

Diese Funktion füllt `n` Positionen ab Position `first` mit Kopien von `x`. Im Beispiel mit dem Vektor `v1` und dem Wert von oben werden 20 Werte eingetragen:

```

uninitialized_fill_n(v1.begin(), 20, Wert);

```

34

Optimierte numerische Arrays (valarray)

Dieses Kapitel behandelt die folgenden Themen:

- `valarray` – Eine Klasse für optimierte numerische Arrays
- Beispiele für die verschiedenen Anwendungsmöglichkeiten

Der Header `<valarray>` schließt die Template-Klasse `valarray` ein, die für mathematische Vektor-Operationen gedacht ist. Ein Objekt der Klasse `vector` ist ein Container zur bequemen und flexiblen Verwaltung von Objekten, die ganz verschiedenen Typen angehören können. Ein `valarray`-Objekt hingegen ist ein für numerische Berechnungen optimierter Vektor – mit der Auswirkung, dass er für den Benutzer etwas umständlicher zu benutzen ist, um den Compiler-Herstellern möglichst viele Freiheiten bei der Optimierung zu ermöglichen. Im Header `<valarray>` sind auch die Klassen `slice` und `gslice` definiert, mit denen ein Ausschnitt aus einem `Valarray` bzw. die Struktur einer Matrix abgebildet werden kann, sowie weitere Template-Hilfsklassen (siehe Seite [864](#) ff.). Anwendungsbeispiele zur Klasse `valarray` finden sich in den Beispielen im Verzeichnis *cppbuch/k34*.

34.1 Konstruktoren

Die Konstruktoren ähneln teilweise denen der `vector`-Klasse.

- `valarray()`
konstruiert ein `Valarray` der Größe 0.
- `explicit valarray(size_t n)`
erzeugt ein `Valarray` mit `n` Elementen, die alle zu 0 initialisiert werden.
- `valarray(const T& val, size_t n)`
erzeugt ein `Valarray` mit `n` Elementen, die alle zu `val` initialisiert werden. *Bei diesem Konstruktor ist die Reihenfolge der Argumente im Vergleich zum Vektor vertauscht.* Es kommt erst der Wert und dann die Anzahl.
- `valarray(const T* ptr, size_t n)`
erzeugt ein `Valarray` mit `n` Elementen. Die Elemente werden mit den ersten `n` Werten initialisiert, auf die `ptr` zeigt. Dieser Konstruktor eignet sich daher gut zur Umwandlung eines C-Arrays in ein `Valarray`. Beispiel:

```
double a[] = {2.2, 3.3, 4.4, 9.9};
valarray<double> w(a, 4);
```

- `valarray(const valarray<T>&)` Kopierkonstruktor
- `valarray(const slice_array<T>&)`
`valarray(const gslice_array<T>&)`
`valarray(const mask_array<T>&)`
`valarray(const indirect_array<T>&)`
Die Argumente dieser Konstruktoren werden in den Abschnitten 34.5 bis 34.8 beschrieben (Seiten 866 – 870).

34.2 Elementfunktionen

- `valarray<T>& operator=(const valarray<T>& rhs)`
Voraussetzung für die Zuweisung ist, dass die Größe des Arguments `rhs` mit der Größe des aufrufenden `Valarrays` übereinstimmt, andernfalls ist das Verhalten undefiniert.
- `valarray<T>& operator=(const T& t)`
Jedem Element des `Valarrays` wird `t` zugewiesen. Beispiel:

```
valarray<double> vd(4);
vd = 100.123;
```

- `valarray<T>& operator=(const slice_array<T>&)`
`valarray<T>& operator=(const gslice_array<T>&)`
`valarray<T>& operator=(const mask_array<T>&)`
`valarray<T>& operator=(const indirect_array<T>&)`

Die Wirkungsweise dieser Zuweisungsoperatoren wird in den Abschnitten 34.5 bis 34.8 beschrieben (Seiten 866 – 870).

- `T operator[](size_t n) const`
`T& operator[](size_t n)`
 Der Indexoperator verhält sich wie üblich: Eine Überprüfung auf Einhaltung des Bereichs findet nicht statt.
- Mit den folgenden Indexoperatoren können definierte Untermengen aus einem Valarray erzeugt werden:

```
valarray<T> operator[](slice) const
slice_array<T> operator[](slice)
valarray<T> operator[](const gslice&) const
gslice_array<T> operator[](const gslice&)
valarray<T> operator[](const valarray<bool>&) const
mask_array<T> operator[](const valarray<bool>&)
valarray<T> operator[](const valarray<size_t>&) const
indirect_array<T> operator[](const valarray<size_t>&)
```

Die Wirkungsweise dieser Indexoperatoren wird in den Abschnitten 34.5 bis 34.8 beschrieben (Seiten 864 – 870).

- `valarray<T> operator+()` const
`valarray<T> operator-()` const
`valarray<T> operator~()` const
`valarray<bool> operator!()` const
 Diese unären Operatoren existieren nur, sofern sie für den Typ der Elemente sinnvoll anwendbar sind. Anwendungsbeispiel:

```
v = -u; // v und u sind Valarrays.
```

- `valarray<T>& operator*=(const valarray<T>&)`
`valarray<T>& operator/=(const valarray<T>&)`
`valarray<T>& operator%=(const valarray<T>&)`
`valarray<T>& operator+=(const valarray<T>&)`
`valarray<T>& operator-=(const valarray<T>&)`
`valarray<T>& operator^=(const valarray<T>&)`
`valarray<T>& operator&=(const valarray<T>&)`
`valarray<T>& operator|=(const valarray<T>&)`
`valarray<T>& operator<<=(const valarray<T>&)`
`valarray<T>& operator>>=(const valarray<T>&)`

Diese Kurzformoperatoren bewirken, dass die jeweilige Operation mit den entsprechenden Elementen ausgeführt wird, d.h. `u += v;` ist dasselbe wie `u[i] += v[i];` für alle `i`.

- `valarray<T>& operator*=(const T&)`
`valarray<T>& operator/=(const T&)`
`valarray<T>& operator%=(const T&)`
`valarray<T>& operator+=(const T&)`
`valarray<T>& operator-=(const T&)`
`valarray<T>& operator^=(const T&)`
`valarray<T>& operator&=(const T&)`


```
valarray<T>& operator|=(const T&)
valarray<T>& operator<<=(const T&)
valarray<T>& operator>>=(const T&)
```

Diese Kurzformoperatoren bewirken, dass die jeweilige Operation mit allen Elementen ausgeführt wird, d.h. $u += t$, ist dasselbe wie $u[i] += t$, für alle i .

- `size_t size() const`
gibt die Anzahl der Elemente zurück.
- `T sum() const`
gibt die Summe aller Elemente zurück. Das Valarray muss mindestens ein Element besitzen.
- `T min() const`
gibt das kleinste Element zurück.
- `T max() const`
gibt das größte Element zurück.
- `valarray<T> shift(int n) const`
gibt ein Valarray der Länge `size()` zurück, dessen Elemente um n Positionen verschoben sind. Anwendungsbeispiele:

```
u = v.shift(2);
// Ergebnis: u[i] = v[i+2] für  $0 \leq i < \text{size}() - 2$ , sowie
// u[size()-2] = u[size()-1] = T(), d.h. 0.
```

```
u = v.shift(-2);
// Ergebnis: u[i] = v[i-2] für  $1 < i < \text{size}()$ , sowie
// u[0] = u[1] = T(), d.h. 0.
```

- `valarray<T> cshift(int n) const`
gibt ein Valarray der Länge `size()` zurück, dessen Elemente um n Positionen verschoben sind. Im Unterschied zu `shift()` werden die Elemente rotiert, d.h. durch das Verschieben herausfallende Elemente werden am anderen Ende eingefügt. Beispiel:

```
int n = ... // irgendeine Zahl
u = v.cshift(n); // Ergebnis: u[i] = v[(i+n)%size()]
```

- `valarray<T> apply(T func(T)) const` und
`valarray<T> apply(T func(const T&)) const`
geben ein Valarray der Länge `size()` zurück, wobei die Elemente aus den Ergebnissen der Funktion `func()` angewendet auf die entsprechenden Elemente des aufrufenden Valarrays bestehen:

```
u = v.apply(f); // Ergebnis: u[i] = f(v[i]) für alle i
```

Leider lassen sich nur (Zeiger auf) Funktionen übergeben, keine Funktionsobjekte!

- `void resize(size_t sz, T c = T())`
Die Funktion ändert die Größe eines Valarrays und initialisiert alle Elemente. Nach dem Aufruf gilt `size() == sz` und `operator[] (i) == c` für alle i .

34.3 Binäre Valarray-Operatoren

Zunächst werden alle binären Operatoren und Funktionen beschrieben, ehe auf Elementfunktionen und die Hilfsklassen eingegangen wird.

Binäre arithmetische Valarray-Operatoren

Für `valarray`-Objekte sind die folgenden arithmetischen Operationen definiert, sofern sie mit dem Typ der Elemente verträglich sind (zum Beispiel existiert der Operator `'%'` nicht für `float`-Zahlen): `*`, `/`, `%`, `+` und `-`. Alle Operatoren treten in drei überladenen Varianten auf, hier gezeigt am Beispiel des Multiplikationsoperators:

```
template<typename T>
valarray<T> operator*(const valarray<T>& v, const valarray<T>& w);
// Anwendung zum Beispiel:
valarray<double> u = v*w; // Ergebnis: u[i] = v[i]*w[i] für alle i

template<typename T>
valarray<T> operator*(const valarray<T>& v, const T& t);
// Anwendung zum Beispiel:
valarray<double> u = v*t; // Ergebnis: u[i] = v[i]*t für alle i

template<typename T>
valarray<T> operator*(const T& t, const valarray<T>& v);
// Anwendung zum Beispiel:
valarray<double> u = t*v; // Ergebnis: u[i] = t*v[i] für alle i
```

Binäre bitweise Valarray-Operatoren

Für `valarray`-Objekte sind die folgenden binären Bit-Operationen definiert, sofern sie mit dem Typ der Elemente verträglich sind: `^`, `&`, `|`, `<<` und `>>`. Auch hier treten die Operatoren in drei überladenen Varianten auf, gezeigt am Beispiel des Und-Operators:

```
template<typename T>
valarray<T> operator&(const valarray<T>& v, const valarray<T>& w);
// Anwendung zum Beispiel:
valarray<int> u = v&w; // Ergebnis: u[i] = v[i] & w[i] für alle i

template<typename T>
valarray<T> operator&(const valarray<T>& v, const T& t);
// Anwendung zum Beispiel:
valarray<int> u = v&t; // Ergebnis: u[i] = v[i] & t für alle i

template<typename T>
valarray<T> operator&(const T& t, const valarray<T>& v);
// Anwendung zum Beispiel:
valarray<int> u = t&v; // Ergebnis: u[i] = t & v[i] für alle i
```

Binäre logische Valarray-Operatoren

Für `valarray`-Objekte sind die folgenden binären logischen Operationen definiert: `&&` und `||`. Auch hier treten die Operatoren in drei überladenen Varianten auf, gezeigt am Beispiel des logischen Und-Operators:

```
template<typename T>
valarray<bool> operator&&(const valarray<T>& v,
                        const valarray<T>& w);

// Anwendung zum Beispiel:
valarray<bool> u = v&&w; // Ergebnis: u[i] = v[i] && w[i] für alle i

template<typename T>
valarray<bool> operator&&(const valarray<T>& v, const T& t);

// Anwendung zum Beispiel:
valarray<bool> u = v&&t; // Ergebnis: u[i] = v[i] && t für alle i

template<typename T>
valarray<bool> operator&&(const T& t, const valarray<T>& v);

// Anwendung zum Beispiel:
valarray<bool> u = t&&v; // Ergebnis: u[i] = t && v[i] für alle i
```

Relationale Valarray-Operatoren

Für `valarray`-Objekte sind die folgenden binären Operationen definiert, sofern sie mit dem Typ der Elemente verträglich sind: `==`, `!=`, `<`, `>`, `<=` und `>=`. Auch hier treten die Operatoren in drei überladenen Varianten auf, gezeigt am Beispiel des Gleichheitsoperators:

```
template<typename T>
valarray<bool> operator==(const valarray<T>& v,
                        const valarray<T>& w);

// Anwendung zum Beispiel:
valarray<bool> u = v==w; // Ergebnis: u[i] = (v[i] == w[i]) für alle i

template<typename T>
valarray<bool> operator==(const valarray<T>& v, const T& t);

// Anwendung zum Beispiel:
valarray<bool> u = v==t; // Ergebnis: u[i] = (v[i] == t) für alle i

template<typename T>
valarray<bool> operator==(const T& t, const valarray<T>& v);

// Anwendung zum Beispiel:
valarray<bool> u = t==v; // Ergebnis: u[i] = (t == v[i]) für alle i
```

34.4 Mathematische Funktionen

Es gibt die folgenden mathematischen Funktionen, die ein Valarray als Argument nehmen und ein Valarray mit dem Ergebnis zurückgeben:

```
template<typename T> valarray<T> abs (const valarray<T>&)
template<typename T> valarray<T> acos (const valarray<T>&)
template<typename T> valarray<T> asin (const valarray<T>&)
template<typename T> valarray<T> atan (const valarray<T>&)
template<typename T> valarray<T> cos (const valarray<T>&)
template<typename T> valarray<T> cosh (const valarray<T>&)
template<typename T> valarray<T> exp (const valarray<T>&)
template<typename T> valarray<T> log (const valarray<T>&)
template<typename T> valarray<T> log10 (const valarray<T>&)
template<typename T> valarray<T> sin (const valarray<T>&)
template<typename T> valarray<T> sinh (const valarray<T>&)
template<typename T> valarray<T> sqrt (const valarray<T>&)
template<typename T> valarray<T> tan (const valarray<T>&)
template<typename T> valarray<T> tanh (const valarray<T>&)
```

Als Anwendung sei hier die Sinusfunktion gezeigt:

```
valarray<double> dieWinkel(100); // 100 Werte
// ... Berechnung der Winkel fehlt hier
// Berechnung der Sinuswerte:
valarray<double> dieSinusWerte = sin(dieWinkel);
// Ergebnis: dieSinusWerte[i] = sin(dieWinkel[i]) für alle i
```

Die Funktionen `atan2()` und `pow()` treten in überladenen Variationen auf:

```
template<typename T>
valarray<T> atan2(const valarray<T>& v, const valarray<T>& w);
// Beispielanwendung mit dem Ergebnis  $u[i] = \text{atan}(v[i]/w[i])$  für alle  $i$ :
valarray<double> u = atan2(v, w);
```

```
template<typename T>
valarray<T> atan2(const valarray<T>& v, const T& t);
// Beispielanwendung mit dem Ergebnis  $u[i] = \text{atan}(v[i]/t)$  für alle  $i$ :
valarray<double> u = atan2(v, t);
```

```
template<typename T>
valarray<T> atan2(const T& t, const valarray<T>& v);
// Beispielanwendung mit dem Ergebnis  $u[i] = \text{atan}(t/v[i])$  für alle  $i$ :
valarray<double> u = atan2(t, v);
```

```
template<typename T>
valarray<T> pow(const valarray<T>& v, const valarray<T>& w);
// Beispielanwendung mit dem Ergebnis  $u[i] = v[i]^{w[i]}$  für alle  $i$ :
valarray<double> u = pow(v, w);
```

```
template<typename T>
valarray<T> pow(const valarray<T>& v, const T& t);
// Beispielanwendung mit dem Ergebnis  $u[i] = v[i]^t$  für alle  $i$ :
valarray<double> u = pow(v, t);
```

```
template<typename T>
valarray<T> pow(const T& t, const valarray<T>& v);
// Beispielanwendung mit dem Ergebnis  $u[i] = t^{v[i]}$  für alle  $i$ :
valarray<double> u = pow(t, v);
```

34.5 slice und slice_array

Die Klasse `slice` (dt. Scheibe, Querschnitt, Teil) ist eine Abstraktion, die es erlaubt, jede Zeile oder Spalte eines Valarrays, das als Matrix interpretiert wird, zu beschreiben oder andere Teilmengen eines Valarrays anzusprechen. Die Klasse ist wie folgt definiert:

```
class slice {
public:
    slice();
    slice(size_t start, size_t size, size_t stride);
    size_t start() const; // Index des ersten Elements
    size_t size() const; // Anzahl der Elemente
    size_t stride() const; // Schrittweite N
private:
    size_t start_, size_, stride_;
};
```

Ein `slice`-Objekt definiert eine Folge von Zahlen, die die Position eines jeden N -ten Elements eines Valarrays angeben, indem die Zahlen *Index des ersten Elements* und *Schrittweite* auf die Position im Valarray abgebildet werden. Zum Beispiel definiert `slice(3, 8, 2)` die Folge 3, 5, 7, 9, 11, 13, 15, 17. Damit lassen sich Untermengen einer Matrix definieren, zum Beispiel Zeilen, Spalten oder Untermatrizen. Im folgenden Beispiel werden die Indizes aller Zeilen und aller Spalten einer Matrix mit 3 Zeilen und 4 Spalten mithilfe von `slice`-Objekten ausgegeben.

Listing 34.1: `slice`

```
// Auszug aus cppbuch/k34/slices.cpp
#include <iostream>
#include "valarray"
using namespace std;

void printIndices(slice s) {
    for(size_t i = 0; i < s.size(); ++i)
        cout << s.start() + i * s.stride() << ' ';
    cout << endl;
}
```

```
int main() {
    const size_t ZEILEN = 3;
    const size_t SPALTEN = 4;
    cout << "Zeige Indizes von Zeilen und Spalten:\n";
    for(size_t zeile = 0; zeile < ZEILEN; ++zeile) {
        cout << "Zeile " << zeile << ": ";
        size_t start_index = zeile * SPALTEN;
        printIndices(slice(start_index, SPALTEN, 1));
    }
    cout << endl;
    for(size_t spalte = 0; spalte < SPALTEN; ++spalte) {
        cout << "Spalte " << spalte << ": ";
        printIndices(slice(spalte, ZEILEN, SPALTEN));
    }
}
```

Das Ergebnis dieses Programms ist:

```
Zeile 0:  0  1  2  3
Zeile 1:  4  5  6  7
Zeile 2:  8  9 10 11

Spalte 0: 0  4  8
Spalte 1: 1  5  9
Spalte 2: 2  6 10
Spalte 3: 3  7 11
```

Der Indexoperator `valarray<T>::operator[](slice)` tritt in zwei Varianten auf:

```
valarray<T> valarray<T>::operator[](slice) const
slice_array<T> valarray<T>::operator[](slice)
```

Er erlaubt es, eine mit einem `slice`-Objekt definierte Untermenge zu extrahieren und zu verändern:

```
valarray<double> v(ZEILEN * SPALTEN); // Platz für 3x4-Matrix
                                   // Konstanten wie oben
// ... (hier Werte zuweisen)
// Zeile 1 (Zählung ab 0) als Valarray extrahieren:
slice z1(4, 4, 1); // Zeile 1 einer 3x4 Matrix definieren
valarray<double> zeile1 = v[z1];
```

Das Valarray `zeile1` besteht aus 4 Elementen, die Kopien der Elemente `v[4]` bis `v[7]` sind. Veränderungen einer Valarray-Untermenge sind auf folgende Art möglich:

```
v[z1] = 0.0; // Wirkung: v[4] = v[5] = v[6] = v[7] = 0.0
// auf Zeile 2 die oben erzeugte zeile1 addieren, d.h
// v[8] += zeile1[0] usw. bis v[11] += zeile1[3]:
v[slice(2*4, 4, 1)] += zeile1;
```

Der auf der linken Seite eines binären Operators stehende Indexoperator liefert ein `slice_array` zurück, für das der binäre Operator aufgerufen wird. Die möglichen Operatoren sind im folgenden Abschnitt 34.5 aufgelistet. Die Klasse `slice_array` tritt im obigen Beispiel nicht explizit auf. Man muss sie nicht selbst kennen, sondern nur ihre Operatoren, die verändernd auf Untermengen von Valarrays wirken.

slice_array

Ein `slice_array` ist eine Hilfsklasse, die eine Referenz auf ein `Valarray` und ein `slice`-Objekt zur Definition einer Untermenge für dieses `Valarray` enthält. Die Klasse beschreibt also nur eine Untermenge, und sie ermöglicht für diese Untermenge die in der Klassendefinition beschriebenen Operationen:

```
template<typename T>
class slice_array {
public:
    typedef T value_type;
    void operator= (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<= (const valarray<T>&) const;
    void operator>>= (const valarray<T>&) const;
    void operator=(const T&);
private:
    // ...
};
```

34.6 gslice und gslice_array

Ein `slice`-Objekt kann eine Zeile oder Spalte einer Matrix beschreiben. Manchmal wird aber eine Untermatrix benötigt, im allgemeinsten Fall eine m -dimensionale Untermatrix einer n -dimensionalen Matrix ($m \leq n$). Dazu dienen `gslice`-Objekte. Die Klasse `gslice` (Abk. für *generalized slice*) enthält alle dafür notwendigen Informationen, nämlich die `slice`-Werte für Anzahl und Schrittweite, gespeichert in `Valarrays`, und den gemeinsamen Startpunkt:

```
class gslice {
public:
    gslice();
    gslice(size_t start,
           valarray<size_t>& lengths, valarray<size_t>& strides);
    size_t start() const;
    valarray<size_t> size() const;
    valarray<size_t> stride() const;
private:
    size_t start_;
    valarray<size_t> sizes_, strides_;
};
```

Die Wirkung soll an einer 4x5-Matrix gezeigt werden, zu der die Indizes von Submatrizen ausgegeben werden. Dazu werden zunächst zwei Hilfsfunktionen definiert, die in `main()` benutzt werden:

```
// mit einem gsllice definierte Indizes ausgeben
void printIndices(const gsllice& gs) {
    for(size_t r = 0; r < gs.size()[0]; ++r) { // Zeilen
        for(size_t c = 0; c < gs.size()[1]; ++c) // Spalten
            cout << '\t'
                << gs.start()
                    + r * gs.stride()[0]
                    + c * gs.stride()[1];
        cout << endl;
    }
}

// gsllice einer zweidimensionalen Untermatrix aus einer durch gs
// definierten zweidimensionalen Matrix zurückgeben
gsllice submatrix_gslice(const gsllice& gs, // 2-D Matrix
                        size_t position, // Startposition
                        size_t rows,    // Zeilen
                        size_t cols) { // Spalten
    size_t sz[] = {rows, cols};
    size_t str[] = {gs.size()[1], 1};
    valarray<size_t> sizes(sz, 2);
    valarray<size_t> strides(str, 2);
    return gsllice(position, sizes, strides);
}
```

```
// Auszug aus cppbuch/k34/gsllices.cpp
int main() {
    const size_t ZEILEN = 4, SPALTEN = 5; // 4x5 Matrix
    valarray<size_t> sizes(2);           // 2 Dimensionen
    valarray<size_t> strides(2);
    sizes[0] = ZEILEN;
    sizes[1] = SPALTEN;
    strides[0] = SPALTEN;
    strides[1] = 1;
    // gsllice in der Ecke oben links konstruieren
    gsllice g(0, sizes, strides);
    cout << endl << "Indizes der 4x5 Matrix:\n";
    printIndices(g);
}
```

Ergebnis:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Die Indizes einer 3x4 Untermatrix, die an Position 1 beginnt, werden mit

```
printIndices(submatrix_gslice(g, 1, 3, 4));
```

ausgegeben und führen zu dem Ergebnis

1	2	3	4
6	7	8	9
11	12	13	14

Ähnlich wie bei der `slice`-Klasse lassen sich Untermengen eines als Matrix interpretierten Valarrays verändern und extrahieren. Der Indexoperator `valarray<T>::operator[](const gsl::slice&)` dient zum Ermitteln der Untermenge und tritt in zwei Varianten auf:

```
valarray<T> valarray<T>::operator[](const gsl::slice&) const
gsl::slice_array<T> valarray<T>::operator[](const gsl::slice&)
```

Die folgende Fortsetzung des `main()`-Programms zeigt die mögliche Anwendung:

```
// 4x5 Matrix erzeugen und initialisieren
valarray<double> v(ZEILEN*SPALTEN);
for(size_t i = 0; i < v.size(); ++i)
    v[i] = i;           // Wert = Index

// gsl::slice für 2x3 Untermatrix an Position 6 erzeugen
gsl::slice gs(submatrix_gslice(g, 6, 2, 3));

// Kopie der Untermatrix erzeugen
valarray<double> sub = v[gs];
```

Das Valarray `sub` enthält die folgenden Elemente:

6	7	8
11	12	13

Mit verschiedenen Operatoren kann eine Untermatrix innerhalb der Matrix verändert werden. Die möglichen Operatoren sind im folgenden Abschnitt 34.6 aufgelistet. Hier wird das Nullsetzen der durch `gs` definierten Untermatrix gezeigt:

```
// Untermatrix = 0 setzen
v[gs] = 0.0;
```

Nach dieser Anweisung enthält Matrix `v` die folgenden Werte:

0	1	2	3	4
5	0	0	0	9
10	0	0	0	14
15	16	17	18	19

Die mathematischen Operationen setzen ein Valarray als Argument voraus, zum Beispiel

```
v1[gs] *= v2; // v1 und v2 sind Valarrays
```

Die Wirkung dieser Anweisung besteht darin, dass diejenigen Elemente von `v1`, die durch das `gsl::slice`-Objekt `gs` indiziert werden, mit den entsprechenden Elementen von `v2` multipliziert werden. Bei der oben gegebenen Definition von `gs` ist die Wirkung

```
// Wirkung der Anweisung v1[gs] *= v2;
v1[6] *= v2[0];
v1[7] *= v2[1];
v1[8] *= v2[2];
v1[11] *= v2[3];
```

```
v1[12] *= v2[4];
v1[13] *= v2[5];
```

gslice_array

Ein `gslice_array` ist eine Hilfsklasse, die eine Referenz auf ein `Valarray` und ein `gslice`-Objekt zur Definition einer Untermenge für dieses `Valarray` enthält. Die Klasse `gslice_array` ermöglicht für diese Untermenge dieselben Operationen wie die Klasse `slice_array`, also = für ein `const T&`-Argument und `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `|=`, `<<=`, `>>=`, = für `Valarray`-Argumente.

34.7 mask_array

Ein `mask_array` spielt eine ähnliche Rolle für Operationen auf Untermengen von `Valarrays` wie ein `slice_array`. Der Unterschied besteht in der Auswahl der Untermenge, die hier durch ein `Valarray` mit `bool`-Elementen geschieht. Die Untermenge besteht aus denjenigen Elementen eines `Valarrays`, für die das entsprechende Element des `bool`-Arrays den Wert `true` hat. Der Indexoperator `valarray<T>::operator[](const valarray<bool>&)` dient zum Ermitteln der Untermenge und tritt in zwei Varianten auf:

```
valarray<T> valarray<T>::operator[](const valarray<bool>&) const
mask_array<T> valarray<T>::operator[](const valarray<bool>&)
```

Die zweite Variante wird implizit benutzt, wenn die durch das `bool`-Array ausgewählte Untermenge geändert werden soll. Wie die Klassen `slice_array` und `gslice_array` tritt die Klasse `mask_array` nicht direkt in Erscheinung, man muss nur die möglichen Operationen kennen. Beispiele:

```
// Valarray anlegen und initialisieren
valarray<double> v(12);
for(size_t i = 0; i < v.size(); ++i)
    v[i] = i;

// bool-Array mit einer Größe ≤ v.size() anlegen.
valarray<bool> maske(10);

// Jedes 3. Element = true setzen
for(size_t i = 0; i < maske.size(); i+= 3)
    maske[i] = true;

// Untermenge auswählen
const valarray<double> u(v[maske]);
// Ergebnis: u = 0 3 6 9

// Elementen der Untermenge einen Wert zuweisen
v[maske] = 20;
// Ergebnis: v = 20 1 2 20 4 5 20 7 8 20 10 11
```

```
// Zweites Valarray vmult erzeugen
valarray<double> vmult(8); //vmult.size() ≥ Anzahl der trues in maske
for(size_t i = 0; i < vmult.size(); ++i)
    vmult[i] = 0.1*i; // 0.0 0.1 0.2 ... 0.7

// Untermenge mit allen Elementen von vmult multiplizieren
v[maske] *= vmult;
// Ergebnis: v = 0 1 2 2 4 5 4 7 8 6 10 11,
// d.h. 20*0.0, 1, 2, 20*0.1, 4, 5, 20*0.2, 7, 8, 20*0.3, 10, 11
```

Die Klasse `mask_array` ermöglicht für die durch das boolesche Array ausgewählte Untermenge dieselben Operationen wie die Klasse `slice_array`, also = für ein `const T&`-Argument und `*, /, %, +, -, =, &, |=, <<=, >>=, =` für Valarray-Argumente.

34.8 indirect_array

Ein `indirect_array` spielt eine ähnliche Rolle für Operationen auf Untermengen von Valarrays wie ein `slice_array`. Der Unterschied besteht in der Auswahl der Untermenge, die hier durch ein Index-Array geschieht (indirekte Adressierung). Ein Index-Array ist ein Valarray mit Elementen vom Typ `size_t`, die Indizes, also Adressen von Array-Elementen, darstellen. Die Untermenge besteht aus denjenigen Elementen eines Valarrays, auf die das entsprechende Element des Index-Arrays verweist. Der Indexoperator `valarray<T>::operator[](const valarray<size_t>&)` dient zum Ermitteln der Untermenge und tritt in zwei Varianten auf:

```
valarray<T> valarray<T>::operator[](const valarray<size_t>&) const
indirect_array<T> valarray<T>::operator[](const valarray<size_t>&)
```

Die zweite Variante wird implizit benutzt, wenn die durch das Index-Array ausgewählte Untermenge geändert werden soll. Wie die Klassen `slice_array` und `gslice_array` tritt die Klasse `indirect_array` nicht direkt in Erscheinung, man muss nur die möglichen Operationen kennen. Beispiele:

```
// Valarray anlegen und initialisieren
valarray<double> v(10);
for(size_t i = 0; i < v.size(); ++i) {
    v[i] = 0.1 * i; // 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
}

// indirektes Valarray (Index-Array) anlegen und initialisieren
valarray<size_t> i_arr(v.size()-4);
for(size_t i = 0; i < i_arr.size(); ++i) {
    i_arr[i] = i_arr.size() - i + 2; // 8 7 6 5 4 3
}
```

Für ein Index-Array müssen zwei Bedingungen gelten, hier mit den beiden Valarrays `v` und `i_arr` formuliert:

```

    i_arr.size() ≤ v.size()
und
    i_arr[i] < v.size(), 0 ≤ i < i_arr.size()

```

Andernfalls ist das Verhalten des Programms undefiniert. Eine Untermenge kann wie folgt erzeugt werden:

```
const valarray<double> u(v[i_arr]); // 0.8 0.7 0.6 0.5 0.4 0.3
```

Es gilt $v[i\_arr[i]] == u[i]$ für alle i im Bereich $0 \leq i < i\_arr.size()$. Die Klasse `indirect_array` ermöglicht für die durch das Index-Array ausgewählte Untermenge dieselben Operationen wie die Klasse `slice_array`, also = für ein `const T&`-Argument und `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `|=`, `<<=`, `>>=`, = für `Valarray`-Argumente. Beispiele:

```

v[i1_arr] = 20;
// Ergebnis: v = 0.0 0.1 0.2 20 20 20 20 20 20 0.9

valarray<double> vb(10);
for(size_t i = 0; i < vb.size(); ++i)
    vb[i] = 0.01*i; // .0 .01 .02 .03 .04 .05 .06 .07 .08 .09

v[i1_arr] = vb;
// Ergebnis: v = 0.0 0.1 0.2 0.05 0.04 0.03 0.02 0.01 0.0 0.9

v[i1_arr] *= vb;
// Ergebnis: v = 0 0.1 0.2 0.0025 0.0016 0.0009 0.0004 0.0001 0 0.9

```

Für solche Operationen gilt $v[i\_arr[i]] *= vb[i]$, hier also

```

v[8] *= vb[0]; v[7] *= vb[1]; v[6] *= vb[2];
v[5] *= vb[3]; v[4] *= vb[4]; v[3] *= vb[5];

```

Außer den gezeigten Beispielen zur Klasse `valarray` finden sich weitere in den Beispielen, siehe Verzeichnis *cppbuch/k34*. Diese Beispiele demonstrieren auf einfache Art die beschriebenen Konzepte. Als Ergänzung gibt es ein Beispiel, das nicht so einfach und daher eher für Mathematiker interessant ist: Ein Programm zur rekursiven Matrixmultiplikation (Unterverzeichnis *cppbuch/k34/rekursiveMatrixMult*).

35

C-Header

Dieses Kapitel behandelt die folgenden Themen:

- Auswahl der nach C++ übernommenen C-Header
- Unterschiede und Alternativen

Dieser Abschnitt zeigt die von der Programmiersprache C übernommenen Header, soweit sie nicht obsolet sind. Der Inhalt ist derselbe wie in der C-Standard Library [ISOC]. Die Dateinamen ergeben sich aus den Header-Namen nach den üblichen Konventionen. Die Datei zum Header `<cmath>` heißt dementsprechend *math.h*. Viele der C-Header sind gut und mit Beispielen in [Her] beschrieben. In diesem Abschnitt *nicht* ausführlich behandelt werden:

- `<cmath>`: Wegen des Headers `<limits>` (siehe Seite 725) ist die Verwendung von `<cmath>` nicht mehr notwendig.
- `<iso646>`: Der Header `<iso646>` existiert nur aus Gründen der Kompatibilität zu C. *iso646.h* definiert Makros für nationale Tastaturen, die bestimmte Sonderzeichen nicht zur Verfügung haben. Zum Beispiel ist `and` ein Makro für `&&`. Die C-Makros sind Schlüsselwörter in C++ (siehe Tabelle auf Seite 887), weswegen `<iso646>` im Allgemeinen nicht mehr gebraucht wird – und die Makros auch nicht mehr enthält.

- `<climits>`: Wegen des Headers `<limits>` (siehe Seite 725) ist die Verwendung von `<climits>`, der Grenzwerte für ganzzahlige Datentypen festlegt, im Allgemeinen nicht mehr notwendig.
- `<locale>`: Wegen des Headers `<locale>` (siehe Seite 821) ist die Verwendung von `<locale>` im Allgemeinen nicht mehr notwendig.
- `<csetjmp>`: Mit den Funktionen `setjmp()` und `longjmp()` des Headers `<csetjmp>` lassen sich Sprünge über Funktionsgrenzen realisieren, mit denen Fehlersituationen in den Funktionen abgefangen werden. Die Funktionen sind dank des in Kapitel 8 beschriebenen Exception-Handlings von C++ im Allgemeinen nicht mehr notwendig.
- `<csignal>`: Im Header `<csignal>` sind Signale und die zugehörigen Funktionen zur Behandlung deklariert. Signale sind Unterbrechungen (englisch *interrupts*), die durch Soft- oder Hardware erzeugt werden, wenn besondere Ausnahmesituationen auftreten, wie zum Beispiel Division durch Null. Signale werden zum Großteil wegen des in Kapitel 8 beschriebenen Exception-Handlings von C++ im Allgemeinen nicht mehr benötigt. Eine ausführliche Beschreibung ist in [Her] zu finden.



35.1 <cassert>

Zusicherungen (englisch *assertions*) werden mit dem Header `<cassert>` eingebunden. Einzelheiten siehe Abschnitt 3.3.5 auf Seite 133.



35.2 <cctype>

Der Header `<cctype>` enthält die in den Tabellen 35.1 und 35.2 aufgeführten C-Funktionen zum Klassifizieren und Umwandeln von Zeichen. Maßgeblich ist die aktuell gültige `locale`-Einstellung, siehe dazu Kapitel 31.

Tabelle 35.1: Umwandlungsfunktionen aus `<cctype>`

Schnittstelle	Bedeutung
<code>tolower(z)</code>	gibt z als Kleinbuchstaben zurück
<code>toupper(z)</code>	gibt z als Großbuchstaben zurück

Weil in C sowohl Zeichen als auch Wahrheitswerte vom Datentyp `int` sind, sind die Parameter und Rückgabewerte der Funktionen in Tabelle 35.1 vom Typ `int`. Der C++-Compiler nimmt die Umwandlung nach `bool` bzw. von oder nach `char` automatisch vor. Nicht darstellbare Zeichen der Tabelle 35.2 sind in Hexadezimalnotation geschrieben (vergleiche mit der ASCII-Tabelle auf Seite 887 f.). Die beiden Funktionen der Tabelle 35.2 geben ihr Argument unverändert zurück, wenn sich eine Umwandlung erübrigt. Unter dem Hea-

der <cwctype> finden sich die entsprechenden Funktionen für »wide character«-Zeichen. `isalnum()` heißt dort `iswalnum()`, und Entsprechendes gilt für die anderen Funktionen.

Tabelle 35.2: Klassifizierungsfunktionen aus <cctype>

Schnittstelle	wahr, wenn z ==	Bereich
<code>isalnum(z)</code>	Buchstabe oder Ziffer	A..Z, a..z, 0..9
<code>isalpha(z)</code>	Buchstabe	A..Z, a..z
<code>isblank(z)</code>	Leerzeichen (C locale)	
<code>iscntrl(z)</code>	Steuerzeichen	0x00..0x1f, 0x7f
<code>isdigit(z)</code>	Ziffer	0..9
<code>isgraph(z)</code>	druckbares Zeichen (ohne ' ')	0x21..0x7e
<code>islower(z)</code>	Kleinbuchstabe	a..z
<code>isprint(z)</code>	druckbares Zeichen (mit ' ')	0x20..0x7e
<code>ispunct(z)</code>	druckbar, aber weder ' ' noch alphanumerisch	0x21..0x2f, 0x3a..0x40 und 0x5b..0x7e
<code>isspace(z)</code>	Zwischenraumzeichen	0x09..0x0d
<code>isupper(z)</code>	Großbuchstabe	A..Z
<code>isxdigit(z)</code>	hexadezimale Ziffer	0..9, A..F, a..f

35.3 <cerrno>

Im Header <cerrno> wird eine globale Variable `errno` deklariert, deren Wert von vielen Systemfunktionen im Fehlerfall gesetzt wird. Der zugehörige Fehlertext wird als `char*` von der Funktion `strerror(int)` (Header <cstring>) zurückgegeben. Beispiel: `cout << strerror(errno);`. Der Verwendung von `errno` ist die in Kapitel 8 beschriebene Ausnahmebehandlung vorzuziehen.

35.4 <cmath>

Die mathematischen Funktionen der folgenden Tabelle 35.3 sind im Header <cmath> für Grunddatentypen zu finden. Es gibt einige Ausnahmen: Die Funktionen `abs()` für Ganzzahlen, `div()`, `rand()` und `srand()` sind aus historischen Gründen unter dem Header <cstdlib> (siehe unten) abgelegt. Hinweis: Manche Funktionen, die laut Text eine Ganzzahl zurückliefern sollen, liefern den Wert dieser Zahl als *Gleitkommazahl* zurück (Beispiel: `ceil()`).

Tabelle 35.3: Mathematische Funktionen

Schnittstelle	Mathematische Entsprechung
F abs(F x)	$ x $
F acos(F x)	$\arccos x$
F asin(F x)	$\arcsin x$
F atan(F x)	$\arctan x$
F atan2(F x, F y)	$\arctan (x/y)$
F ceil(F x)	kleinste Ganzzahl größer oder gleich x
F cos(F x)	$\cos x$
F cosh(F x)	$\cosh x$
F exp(F x)	e^x
F fabs(F x)	$ x $
F floor(F x)	größte Ganzzahl kleiner oder gleich x
F fmod(F x, F y)	Rest der Division x/y
F frexp(F x, int* pn)	zerlegt eine Zahl x in die Mantisse m und den Exponent n . pn ist ein Zeiger auf den Exponenten n , d.h. $n = *pn$. Es gilt $0.5 \leq m < 1$ und $x = m \cdot 2^n$.
F ldexp(F x, int n)	$x \cdot 2^n$
F log(F x)	$\ln x$
F log10(F x)	$\log_{10} x$
F modf(F x, F* i)	zerlegt x in einen ganzzahligen Anteil und den Rest. Dabei ist $*i$ der ganzzahlige Anteil von x . Der restliche Bruchteil wird zurückgegeben.
F pow(F x, F y)	x^y
F pow(F x, int y)	x^y
F sin(F x)	$\sin x$
F sinh(F x)	$\sinh x$
F sqrt(F x)	$\sqrt{x}$
F tan(F x)	$\tan x$
F tanh(F x)	$\tanh x$

Abkürzung: F = einer der Typen float, double oder long double

35.5 <stdarg.h>

Funktionen mit Argumentlisten variabler Länge enthalten eine Ellipse (drei Punkte als Auslassungszeichen) in der Parameterliste, etwa (int, ...). Die Funktionen benötigen die Datentypen und Makros des Headers `<stdarg.h>`. Ein Beispiel für eine Ellipse in der Parameterliste ist die C-Funktion `printf()`, ein anderes ist auf Seite 306 zu finden. Von Funktionen dieser Art wird grundsätzlich abgeraten, weil die Typprüfung der Aufrufparameter durch den Compiler außer Kraft gesetzt ist.

35.6 <cstdlib>

Der Header <cstdlib> enthält Standarddefinitionen des jeweiligen Systems (Tabelle 35.4).

Tabelle 35.4: Standarddefinitionen aus <cstdlib>

Name	Bedeutung
size_t	vorzeichenloser Ganzzahltyp für das Ergebnis von sizeof
ptrdiff_t	ganzahliger Typ mit Vorzeichen zur Subtraktion von Zeigern
wchar_t	Typ für die auf Seite 51 erwähnten »wide characters«
offsetof	Abstand eines Strukturelements vom Strukturanfang in Bytes
NULL	Null-Zeiger (dasselbe wie 0 oder 0L in C++)

35.7 <cstdio>

Die im Header <cstdio> abgelegten Ein- und Ausgabefunktionen sind wegen der iostream-Bibliothek im Allgemeinen nicht mehr notwendig. Deswegen werden sie hier nicht aufgeführt. Es gibt aber Ausnahmen, weil C++ keine Bibliothek für die Dateiverwaltung (löschen, umbenennen, Verzeichnis lesen und anlegen usw.) hat. Diese Ausnahmen werden jedoch von der Boost-Library abgedeckt. Abschnitt 25.1 gibt einen Überblick über die wichtigsten Funktionen, teilweise auf cstdio basierend, teilweise auf Boost.

35.8 <stdlib>

Zunächst seien in Tabelle 35.5 die in Tabelle 35.3 fehlenden mathematischen Funktionen aufgeführt, die aus historischen Gründen zum Header <stdlib> gehören. div_t ist eine vordefinierte Struktur, die das Divisionsergebnis und den Rest enthält. Die long-Variante dieser Struktur ist ldiv_t.

```
struct div_t { int quot; int rem; }; // Quotient, Rest (remainder)
```

Die wichtigsten anderen Funktionen zeigt Tabelle 35.6. Die Speicherverwaltungsfunktionen sind weggelassen worden, weil sie wegen new und delete nicht mehr notwendig sind.

Tabelle 35.5: Mathematische Funktionen aus <stdlib>

Schnittstelle	Mathematische Entsprechung
int abs(int x)	$ x $
long abs(long x)	$ x $
long labs(long x)	$ x $
div_t div(int z, int n)	Struktur div_t (siehe Text)
ldiv_t div(long z, long n)	Struktur ldiv_t (siehe Text)
ldiv_t ldiv(long z, long n)	Struktur ldiv_t (siehe Text)
int rand()	Pseudozufallszahl zwischen 0 und RAND_MAX. RAND_MAX ist die größtmögliche Pseudozufallszahl.
void srand(unsigned seed)	initialisiert den Zufallszahlengenerator

Tabelle 35.6: Ausgewählte Funktionen aus <stdlib>

Schnittstelle	Bedeutung
void abort() void atexit(void (*f)())	Programmabbruch trägt die Funktion f in eine Liste von Funktionen ein, die vor dem normalen Programmende aufgerufen werden.
void exit(int status)	normales Programmende. Der Status wird an das Betriebssystem gemeldet.
int system(const char* B)	Befehl B an den Kommandointerpreter des Betriebssystems geben
char* getenv(const char* E)	Wert der Environmentvariablen E
int atoi(const char* s)	interpretiert den C-String s als int-Zahl
long int atol(const char*s)	dito als long-Zahl
double atof(const char* s)	dito als double-Zahl
long strtol(const char* s, char** rest, int basis)	interpretiert den C-String s als long-Zahl. basis gibt die Basis an. Ein ggf. nicht konvertierbarer Rest wird in rest abgelegt. Ausführliche Beschreibung und Anwendungsbeispiel siehe Seite 627.
unsigned long strtoul(const char* s, char** rest, int basis)	dito für unsigned long
double strtod(const char* s, char** rest)	dito für double
void* bsearch(const void* key, const void* base, size_t n, size_t size, int (*cmp)(const void*, const void*))	binäre Suche. key zeigt auf den Schlüssel, der im Feld base mit n Elementen gesucht wird. Die Größe der Feldelemente ist size, die Vergleichsfunktion ist cmp.
void qsort(void*, size_t, size_t, int (*)(const void*, const void*))	Quicksort (siehe Seite 224)

35.9 <cstring>

Obwohl die C++-String-Klasse existiert, sind die auf `char*` basierenden C-Strings unerlässlich, weswegen die wichtigsten Funktionen des Headers `<cstring>` hier beschrieben werden. Außer Funktionen für C-Strings sind Funktionen für Bytefelder enthalten, die nicht mit `'\0'` abgeschlossen sein müssen.

Abweichung vom C-Standard

In Abweichung vom C-Standard treten manche Prototypen doppelt auf, siehe zum Beispiel `strchr()`, aber auch viele andere. Der Grund liegt in der gewünschten besseren Typsicherheit. So möchte man evtl. mit `strchr()` die Adresse eines Zeichens in einer konstanten Zeichenkette ermitteln. Ein Rückgabotyp `char*` statt `const char*` würde die `const`-Eigenschaft beseitigen, und der Compiler würde sich beschweren. Andererseits möchte man evtl. mit `strchr()` die Adresse eines Zeichens in einer Zeichenkette ermitteln, die anschließend modifiziert werden soll. Dann wäre der Typ `const char*` ungeeignet. Das Vorhandensein beider Varianten gestattet es dem Compiler, die zum Typ des Parameters passende auszusuchen.

Funktionen für C-Strings (Auswahl)

- `char* strcat(char* s1, const char* s2)` kopiert die Zeichenkette `s2` an das Ende von `s1` und liefert `s1` zurück. Das Nullbyte am Ende von `s1` wird überschrieben. Die Zeichenketten dürfen sich nicht überlappen.
- `const char* strchr(const char* s, int c)` und
`char* strchr(char* s, int c)`
 geben die Adresse des Zeichens `c` zurück, falls es im Feld `s` einschließlich `'\0'` gefunden wurde, ansonsten ist das Ergebnis `NULL`.
- `int strcmp(const char* s1, const char* s2)`
 vergleicht die Speicherbereiche `s1` und `s2`. Es wird 0 zurückgegeben, wenn kein Unterschied festgestellt wird. Falls das erste unterschiedliche Zeichen von `s1` kleiner als das entsprechende in `s2` ist, wird eine negative Zahl zurückgegeben; falls es größer ist, eine positive Zahl.
- `char* strcpy(char* z, const char* q)`
 kopiert alle Bytes bis einschließlich des Nullbytes von der Quelle `q` zum Ziel `z`. Die Zieladresse wird zurückgegeben. Quell- und Zielbereich dürfen sich nicht überlappen.
- `int strcspn(const char* s1, const char* s2)`
 gibt die größte Länge einer Zeichenkette vom Anfang bis zu der Position in `s1` an, bis zu der kein in der Zeichenkette `s2` vorhandenes Zeichen gefunden wird.
- `char* strerror(int N)`
 liefert den Text der Systemfehlermeldung mit der Nummer `N` als C-String (vergleiche `errno` auf Seite 875).
- `size_t strlen(const char* s)`
 gibt die Anzahl der Zeichen (ohne Nullbyte) im C-String `s` zurück.

- `const char *strpbrk(const char* s1, const char* s2)` und
`char *strpbrk(        char* s1, const char* s2)`
 liefern die Position in `s1`, an der erstmalig ein Zeichen aus `s2` gefunden wird. Falls nichts gefunden wird, ist das Funktionsergebnis `NULL`.
- `const char* strchr(const char* s, int c)` und
`char* strchr(        char* s, int c)`
 liefern die Position in `s`, an der letztmalig das Zeichen `c` gefunden wird. Falls nichts gefunden wird, ist das Funktionsergebnis `NULL`.
- `const char *strstr(const char* s1, const char* s2)` und
`char *strstr(        char* s1, const char* s2)`
 liefern die Position in `s1`, an der erstmalig die Zeichenkette `s2` (ohne `'\0'`) gefunden wird. Falls nichts gefunden wird, ist das Funktionsergebnis `NULL`.
- `char* strtok(char* s, const char* trenn)`
 Die Zeichenkette `s` wird in sogenannte Tokens zerlegt, wobei die möglichen Trennzeichen, durch die die Tokens getrennt sind, durch `trenn` definiert sind. Die Funktion gibt in aufeinanderfolgenden Aufrufen (mit dem Argument `NULL` ab dem 2. Aufruf) jeweils die Adresse des nächsten Tokens an, sofern eins gefunden wird. Der C-String `s` wird dabei modifiziert, indem die Trennzeichen mit `'\0'` überschrieben werden. Ein Beispiel ist auf Seite 643 zu sehen.

Funktionen für C-Strings maximaler Länge

- `char* strncat(char* s1, const char* s2, size_t n)`
 kopiert die Zeichenkette `s2` an das Ende von `s1` und liefert `s1` zurück. Das Nullbyte am Ende von `s1` wird überschrieben. Es werden maximal `n` Zeichen geschrieben. Die Zeichenketten dürfen sich nicht überlappen.
- `int strncmp(const char* s1, const char* s2, size_t n)`
 vergleicht maximal `n` Zeichen der Speicherbereiche `s1` und `s2`. Es wird 0 zurückgegeben, wenn kein Unterschied festgestellt wird. Falls das erste unterschiedliche Zeichen von `s1` kleiner als das entsprechende in `s2` ist, wird eine negative Zahl zurückgegeben; falls es größer ist, eine positive Zahl.
- `char* strncpy(char* z, const char* q, size_t n)`
 kopiert die Bytes bis einschließlich des Nullbytes, aber maximal `n`, von der Quelle `q` zum Ziel `z`. Die Zieladresse wird zurückgegeben. Quell- und Zielbereich dürfen sich nicht überlappen.

Funktionen für Bytefelder

- `const void* memchr(const void* s, int c, size_t n)` und
`void* memchr(        void* s, int c, size_t n)`
 geben die Adresse des Bytes `c` zurück, falls es in den ersten `n` Positionen des Feldes `s` gefunden wurde, ansonsten ist das Ergebnis `NULL`.
- `int memcmp(const void* s1, const void* s2, size_t n)`
 vergleicht maximal `n` Positionen der Speicherbereiche `s1` und `s2`. Es wird 0 zurückgegeben, wenn kein Unterschied festgestellt wird. Falls das erste unterschiedliche Byte

von `s1` kleiner als das entsprechende in `s2` ist, wird eine negative Zahl zurückgegeben; falls es größer ist, eine positive Zahl.

- `void* memcpy(void* z, const void* q, size_t n)`
kopiert `n` Bytes von der Quelle `q` zum Ziel `z`. Die Zieladresse wird zurückgegeben. Quell- und Zielbereich dürfen sich nicht überlappen.
- `void* memmove(void* z, const void* q, size_t n)`
kopiert `n` Bytes von der Quelle `q` zum Ziel `z`. Die Zieladresse wird zurückgegeben. Quell- und Zielbereich dürfen sich überlappen.
- `void* memset(void* s, int c, size_t n)`
initialisiert die ersten `n` Positionen des Feldes `s` mit dem Byte `c`. Die Adresse `s` wird zurückgegeben.

35.10 <ctime>

Der Header <ctime> enthält verschiedene Funktionen zur Bearbeitung und Auswertung der Systemzeitinformation (Anwendungsbeispiel siehe Seite 335).

Datentypen

Neben den bekannten Typen `NULL` und `size_t` gibt es die Datentypen `clock_t` (für CPU-Zeiten), `time_t` (für Datums- und Zeitangaben) und `struct tm`, eine Struktur mit mindestens den Elementen der Tabelle 35.7:

Tabelle 35.7: Struktur `tm`

tm-Element	Bedeutung
<code>int tm_sec</code>	Sekunden 0 .. 59
<code>int tm_min</code>	Minuten 0 .. 59
<code>int tm_hour</code>	Stunden 0 .. 23
<code>int tm_mday</code>	Monatstag 1 .. 31
<code>int tm_mon</code>	Monat 0 .. 11
<code>int tm_year</code>	Jahr seit 1900
<code>int tm_wday</code>	Wochentag seit Sonntag 0 .. 6
<code>int tm_yday</code>	Tag seit 1. Januar 0 .. 365
<code>int tm_isdst</code>	<i>is daylight saving time</i> , Werte: Sommerzeit (> 0), Winterzeit (0), undefiniert (-1)

Funktionen

- `char* asctime(const tm*)`
`char* ctime(const time_t*)`
Die Funktionen wandeln die im `tm`-Format oder `time_t`-Format vorliegende Zeit in einen formatierten C-String um. Der C-String endet mit `\n\0`, beendet also die laufende Zeile.

- `clock_t clock()`
gibt die seit Programmstart verstrichene CPU-Zeit in »Ticks« zurück. Die Ticks können in Sekunden umgerechnet werden, wenn der von `clock()` zurückgegebene Wert durch die vordefinierte Konstante `CLOCKS_PER_SEC` dividiert wird.
- `double difftime(time_t t1, time_t t2)`
ermittelt die Differenz beider Zeiten in Sekunden.
- `size_t strftime(char* buf, size_t max, const char* format, const tm* z)`
Die Funktion wandelt die im `tm`-Format vorliegende Zeit in einen formatierten C-String um, wobei das Ergebnis im Puffer `buf` abgelegt wird und `format` einen C-String mit Formatvorgaben darstellt. Die Anzahl der in den Puffer geschriebenen Zeichen wird zurückgegeben, falls sie $< \text{max}$ ist, ansonsten ist das Ergebnis der Funktion 0. Die im Unixsystem möglichen Formate können in der Shell mit `man strftime` erfragt werden. Beispiele für übliche Formate: `"%j"` gibt den Tag des Jahres aus, `"%c"` Datum und Uhrzeit, `"%x"` nur das Datum. Die Formate können im Formatstring aneinandergehängt werden. Nicht als Format interpretierte Zeichen werden direkt übertragen.
- `tm* gmtime(const time_t* z)`
`tm* localtime(const time_t* z)`
Beide Funktionen wandeln die in `*z` vorliegende Zeit in die Struktur `tm` um. Dabei gibt `localtime()` die lokale Ortszeit unter Berücksichtigung von Sommer- und Winterzeit zurück, während `gmtime()` die UTC (Universal Time Coordinated, entspricht GMT (Greenwich Mean Time)) zurückgibt. Die UTC in einem Unixsystem basiert auf der Zahl der seit dem 1.1.1970 verstrichenen Sekunden.
- `time_t mktime(const tm*)`
wandelt eine Zeit im `tm`-Format in das `time_t`-Format um.
- `time_t time(time_t* z)`
gibt die momentane Kalenderzeit zurück bzw. -1 bei Fehler. Falls `z` $\neq$ NULL ist, wird der Rückgabewert an der Stelle `z` hinterlegt.

A

Anhang

Im Anhang finden Sie die folgenden Themen:

- Empfehlungen zur Programmierung
- C++-Schlüsselwörter
- Die ASCII-Tabelle
- Rangfolge der Operatoren
- Compilerbefehle
- Lösungen zu den Aufgaben
- Hinweise zur Installation der Software von der DVD

A.1 Programmierhinweise

Im Text sind Tips und Hinweise zur Programmierung vorhanden, von denen einige hier in zusammengefasster Darstellung erscheinen.

1. Programme werden für Menschen geschrieben!

Nur lesbare und verständliche Programme sind wartbar. Ein Programm wird nur einmal geschrieben, aber mehrfach gelesen. Schwer verständliche Programme bergen überdies die Gefahr einer erhöhten Fehlerwahrscheinlichkeit. Die Bedeutung von Kommentaren und der Strukturierung des Programmcodes sollte man nicht unterschätzen! Außerdem sollte es bei etwas größeren Programmen getrennt vom Code eine problembezogene (objektorientierte Analyse) und eine programmbezogene (objektorientierter Entwurf) Dokumentation geben.

Das Einhalten von Programmierrichtlinien unterstützt das Schreiben gut lesbarer Programme. Es gibt sehr einige dieser Richtlinien, die sich in großen Teilen ähneln. Deshalb sei hier nur auf die wohl am besten bekannten »Ellementel«-Regeln [HeNy], die JSF AV C++ Coding Standards [JSF] und auf den CERT C++ Secure Coding Standard [CERT] hingewiesen. Ein einfaches Beispiel für solche Regeln sind Vorschriften für die Schreibweise, etwa:

- Die Namen von eigenen Klassen sollen stets mit einem Großbuchstaben beginnen (im Gegensatz zu denen der C++-Standardbibliothek).
- Die Namen von Variablen und Funktionen beginnen mit einem Kleinbuchstaben.
- Konstantennamen sind vollständig groß zu schreiben, z.B. FAKTOR.
- Worttrennungen sind durch Wechsel in der Groß-/Kleinschreibung oder durch einen Unterstrich zu kennzeichnen, z.B. `anzahlDerObjekte` oder `anzahl_der_objekte`.
- Der Name einer Header-Datei soll dem Namen der Klasse entsprechen, die in dieser Datei deklariert wird.

Weitere Empfehlungen sind die nachfolgend aufgezählten Punkte, auch finden Sie in diesem Buch viele weitere Hinweise an den thematisch entsprechenden Stellen.

2. Trennung von Schnittstellen und Implementation

Die Trennung von Schnittstellen und Implementation ist ein wichtiges Mittel, um Software wartbar und wieder verwendbar zu gestalten. Üblich sind

- die Trennung von Funktionsprototyp und Funktionsdefinition sowie
- die Definition einer gemeinsamen Schnittstelle für Klassen mit Hilfe einer abstrakten Klasse.

3. Konstruktion von Schnittstellen

Empfehlungen zur Konstruktion von Schnittstellen sind in Abschnitt 20.1 auf Seite 557 ff. zusammengefasst. Falls nicht ausgeschlossen ist, dass von einer Klasse geerbt wird, müssen alle Methoden, die dabei überschrieben werden könnten, `virtual` sein. Siehe dazu auch Punkt 12 unten.

4. Datenkapselung

Der Zugriff auf die Daten von Objekten sollte restriktiv gehandhabt werden. Die Erleichterung des Zugriffs mit `friend` oder `public`-Datenbereichen muss begründet sein. Verzichten Sie nach Möglichkeit auf globale Daten und Funktionen.

5. One Definition Rule

Jede Variable, Funktion, Struktur, Konstante und so weiter in einem Programm hat *genau eine* Definition.

6. Zeiger in Klassen

Zeiger in einer Klasse, die auf dynamisch erzeugte Objekte verweisen, erfordern für die Klasse in der Regel je einen besonderen Kopierkonstruktor, Zuweisungsoperator und Destruktor.

7. Kopierkonstruktor, Zuweisungsoperator, Destruktor

Wenn einer der drei für eine Klasse `X` geschrieben werden muss, sind meistens auch die anderen beiden notwendig.

- Der Kopierkonstruktor soll das zu kopierende Objekt nicht verändern und es daher als konstante Referenz übergeben:

```
X::X(const X&);           // Kopierkonstruktor
```

- Der Zuweisungsoperator soll `*this` als Referenz (`X&`) zurückgeben, damit die Verkettung von Zuweisungen möglich ist.
- Der Zuweisungsoperator soll bei dynamischen, also mit `new` erzeugten Teilen des Objekts, die folgende Struktur aufweisen, wenn die linke und die rechte Seite der Zuweisung vom selben statischen Typ sind:

```
X& X::operator=(X obj) { // Übergabe per Wert! (vgl. Seite 329)
    swap(obj);           // wirft keine Exception
    return *this;
} // Aufruf des Destruktors von obj
```

Die Funktion `X::swap(X&)` muss natürlich existieren. Sie vertauscht die Attribute von `*this` mit denen von `obj`. Dabei kann vorteilhaft die Funktion `swap()` der Standardbibliothek eingesetzt werden, zum Beispiel

```
void X::swap(X& obj) {
    std::swap(attribut1, obj.attribut1);
    std::swap(attribut2, obj.attribut2);
    // usw.
}
```

Diese Form ist exception-sicher. Sie ermöglicht dem Compiler, bei einem temporären Argument auf der rechten Seite der Zuweisung, den Kopierkonstruktor zu umgehen.

- Ein nicht nach obigem Muster geschriebener Zuweisungsoperator kann eine Prüfung der Zuweisung des Objekts auf sich selbst enthalten, nicht unnötige oder gefährliche Anweisungen (zum Beispiel `delete`) auszuführen:

```
X& X::operator=(const X& obj) {
    if(this != &obj) {
        //... Anweisungen (Ausführung nur bei Nicht-Identität)
    }
    return *this;
}
```

In der Praxis wird das wohl kaum vorkommen. Das Weglassen dieser Prüfung ist unschädlich, wenn der Zuweisungsoperator exception-sicher ist.

8. Referenzen oder Zeiger?

Alles, was mit Referenzen getan werden kann, ist im Prinzip auch mit Zeigern möglich. In manchen Fällen sind Referenzen jedoch vorzuziehen.

Referenzen sind bei der Übergabe in und aus Funktionen sinnvoll, weil sie innerhalb der Funktion syntaktisch wie ein Objektname verwendet werden können. Der Compiler löst die Referenz auf, während beim Zeiger stets vom Programmierer dereferenziert werden muss. Eine Referenz bezieht sich immer auf ein existierendes Objekt, sie kann nie NULL sein.

9. Wann wird `delete [ ]` benötigt?

`delete [ ]` ist genau dann erforderlich, wenn das zu löschende Objekt mit `new [ ]` erzeugt wurde. Der Compiler weiß (leider) nicht, ob ein Objekt mit `new [ ]` erzeugt wurde, und prüft daher auch nicht, ob es mit `delete [ ]` freigegeben wird.

10. Speicherbeschaffung und -freigabe kapseln

Die Operatoren `new` und `delete` sind stets paarweise zu verwenden. Um Speicherfehler zu vermeiden, empfiehlt sich das »Verpacken« dieser Operationen in Konstruktor und Destruktor wie bei der Beispielklasse `MeinString` (Seite 233) oder bei der Verwendung der »Smart Pointer« von Seite 339. Ein weiterer Vorteil ist die korrekte Speicherfreigabe bei Exceptions (siehe Seite 567).

11. Wird ein virtueller Destruktor benötigt?

Das Vorhandensein virtueller Funktionen ist ein Indiz für die Notwendigkeit eines virtuellen Destruktors. Er wird genau dann benötigt, wenn `delete` auf einen Basis-klassenzeiger angewendet wird, der auf ein dynamisch erzeugtes Objekt einer abgeleiteten Klasse verweist. Virtuelle Destruktoren sollten immer dann verwendet werden, wenn von der betreffenden Klasse abgeleitet wird oder nicht auszuschließen ist, dass von ihr zukünftig durch Ableitung neue Klassen gebildet werden.

12. Nur virtuelle Funktionen überschreiben!

Nicht-virtuelle Funktionen einer Basisklasse sollten *nicht* in abgeleiteten Klassen überschrieben werden. Der Grund liegt darin, dass das Verhalten eines Programms sich nicht ändern sollte, wenn auf eine Methode über den Objektnamen oder über Basis-klassenzeiger bzw. -referenzen zugegriffen wird.

13. Initialisierung von Objekten

Objekte sollten aus Effizienzgründen über Initialisierungslisten anstatt mit Zuweisungen im Codeblock des Konstruktors initialisiert werden. Die Initialisierung von Objektkonstanten ist ohnehin nur über eine Liste möglich.

14. Konstanz von Objekten

Nutzen Sie die Prüfungsmöglichkeiten des Compilers! Alle Modifikationsversuche unveränderlicher Objekte werden schon vom Compiler zurückgewiesen, wenn sie als `const` deklariert sind.

Ein (konstantes oder veränderliches) Objekt einer Klasse `X`, das durch einen Funktionsaufruf *nicht* verändert werden soll, ist an eine Funktion per Wert (`int func(X Obj)`), per konstanter Referenz (`int func(const X& Obj)`) oder per Zeiger auf ein konstantes Objekt (`int func(const X* ZeigerAufObjekt)`) zu übergeben. Bei größeren Objekten empfiehlt sich eine der beiden letzten Möglichkeiten.

15. Makros

Verwenden Sie nur wirklich notwendige Makros. Meistens gibt es eine alternative Lösung in C++.

16. inline

Funktionen sollten nur dann `inline` deklariert werden, wenn sie sehr kurz sind und/oder die Laufzeit deutlich verbessert wird.

A.2 C++-Schlüsselwörter

Die Bezeichner in Tabelle A.1 sind reserviert für den Gebrauch als Schlüsselwort und sollen nicht anderweitig benutzt werden. In der Tabelle sind Symbole, die als Ersatz für bestimmte Zeichen gelten können, *nicht* enthalten (Beispiele: `and`, `or`, `not_eq`, ...);

Tabelle A.1: C++-Schlüsselwörter

<code>alignas</code>	<code>const_cast</code>	<code>extern</code>	<code>noexcept</code>	<code>static_assert</code>	<code>union</code>
<code>alignof</code>	<code>constexpr</code>	<code>false</code>	<code>nullptr</code>	<code>static_cast</code>	<code>unsigned</code>
<code>asm</code>	<code>continue</code>	<code>float</code>	<code>operator</code>	<code>struct</code>	<code>using</code>
<code>auto</code>	<code>decltype</code>	<code>for</code>	<code>private</code>	<code>switch</code>	<code>virtual</code>
<code>bool</code>	<code>default</code>	<code>friend</code>	<code>protected</code>	<code>template</code>	<code>void</code>
<code>break</code>	<code>delete</code>	<code>goto</code>	<code>public</code>	<code>this</code>	<code>volatile</code>
<code>case</code>	<code>do</code>	<code>if</code>	<code>register</code>	<code>thread_local</code>	<code>wchar_t</code>
<code>catch</code>	<code>double</code>	<code>inline</code>	<code>reinterpret_cast</code>	<code>throw</code>	<code>while</code>
<code>char</code>	<code>dynamic_cast</code>	<code>int</code>	<code>return</code>	<code>true</code>	
<code>char16_t</code>	<code>else</code>	<code>long</code>	<code>short</code>	<code>try</code>	
<code>char32_t</code>	<code>enum</code>	<code>mutable</code>	<code>signed</code>	<code>typedef</code>	
<code>class</code>	<code>explicit</code>	<code>namespace</code>	<code>sizeof</code>	<code>typeid</code>	
<code>const</code>	<code>export</code>	<code>new</code>	<code>static</code>	<code>typename</code>	

Darüber hinaus gibt es die reservierten Bezeichner `final` und `override`.

A.3 ASCII-Tabelle

ASCII ist die Abkürzung für *American Standard Code for Information Interchange*. Es gibt auch einen ISO-Code (ISO = *International Standards Organization*), der teilweise nationale Symbole erlaubt. ASCII ist jedoch weiter verbreitet. Er ist ein 7-Bit-Code und besteht aus 128 Zeichen, die in nichtdruckbare und druckbare Zeichen unterteilt werden. Die ersteren werden Steuerzeichen (englisch *control characters*) genannt. Die Zeichen sind in den folgenden Tabellen A.2 und A.3 dargestellt.

Der Piepton »bell« der ersten Tabelle könnte natürlich als `\x07` anstatt als `\a` geschrieben werden, dasselbe gilt entsprechend für `\0`, `\t`, `\v` und `\r`. Anstelle der Hex-Darstellung `\x..` ist auch die oktale Darstellung möglich (siehe Seite 44). Die Zeichen mit den Nummern 34, 39 und 92 haben eine besondere Bedeutung in C++, weswegen sie in einem Programm durch einen vorangestellten Backslash (`\`) gekennzeichnet werden müssen, wenn nur das Zeichen selbst gemeint ist.

Tabelle A.2: ASCII-Steuerzeichen

Nr.	hex	Abkürzung	Name	C++
0	0x00	NUL	null	\0
1	0x01	SOH	start of heading	\x01
2	0x02	STX	start of text	\x02
3	0x03	ETX	end of text	\x03
4	0x04	EOT	end of transmission	\x04
5	0x05	ENQ	enquiry	\x05
6	0x06	ACK	acknowledge	\x06
7	0x07	BEL	alert	\a
8	0x08	BS	backspace	\b
9	0x09	HT	horizontal tab	\t
10	0x0A	NL/LF	new-line	\n
11	0x0B	VT	vertical tab	\v
12	0x0C	FF	form feed	\f
13	0x0D	CR	carriage return	\r
14	0x0E	SO	shift out	\x0E
15	0x0F	SI	shift in	\x0F
16	0x10	DLE	data link escape	\x10
17	0x11	DC1	device control 1	\x11
18	0x12	DC2	device control 2	\x12
19	0x13	DC3	device control 3	\x13
20	0x14	DC4	device control 4	\x14
21	0x15	NAK	negative acknowledge	\x15
22	0x16	SYN	synchronous idle	\x16
23	0x17	ETB	end of transmission block	\x17
24	0x18	CAN	cancel	\x18
25	0x19	EM	end of medium	\x19
26	0x1A	SUB	substitute	\x1A
27	0x1B	ESC	escape	\x1B
28	0x1C	FS	file separator	\x1C
29	0x1D	GS	group separator	\x1D
30	0x1E	RS	record separator	\x1E
31	0x1F	US	unit separator	\x1F
127	0x7F	DEL	delete	\x7F

Tabelle A.3: Druckbare ASCII-Zeichen (Spalte Z. = Zeichen)

Nr.	hex	Z.	C++	Nr.	hex	Z.	C++	Nr.	hex	Z.	C++
32	0x20			64	0x40	@	@	96	0x60	`	`
33	0x21	!	!	65	0x41	A	A	97	0x61	a	a
34	0x22	"	\"	66	0x42	B	B	98	0x62	b	b
35	0x23	#	#	67	0x43	C	C	99	0x63	c	c
36	0x24	\$	\$	68	0x44	D	D	100	0x64	d	d
37	0x25	%	%	69	0x45	E	E	101	0x65	e	e
38	0x26	&	&	70	0x46	F	F	102	0x66	f	f
39	0x27	'	\'	71	0x47	G	G	103	0x67	g	g
40	0x28	(	(	72	0x48	H	H	104	0x68	h	h
41	0x29	)	)	73	0x49	I	I	105	0x69	i	i
42	0x2A	*	*	74	0x4A	J	J	106	0x6A	j	j
43	0x2B	+	+	75	0x4B	K	K	107	0x6B	k	k
44	0x2C	,	,	76	0x4C	L	L	108	0x6C	l	l
45	0x2D	-	-	77	0x4D	M	M	109	0x6D	m	m
46	0x2E	.	.	78	0x4E	N	N	110	0x6E	n	n
47	0x2F	/	/	79	0x4F	O	O	111	0x6F	o	o
48	0x30	0	0	80	0x50	P	P	112	0x70	p	p
49	0x31	1	1	81	0x51	Q	Q	113	0x71	q	q
50	0x32	2	2	82	0x52	R	R	114	0x72	r	r
51	0x33	3	3	83	0x53	S	S	115	0x73	s	s
52	0x34	4	4	84	0x54	T	T	116	0x74	t	t
53	0x35	5	5	85	0x55	U	U	117	0x75	u	u
54	0x36	6	6	86	0x56	V	V	118	0x76	v	v
55	0x37	7	7	87	0x57	W	W	119	0x77	w	w
56	0x38	8	8	88	0x58	X	X	120	0x78	x	x
57	0x39	9	9	89	0x59	Y	Y	121	0x79	y	y
58	0x3A	:	:	90	0x5A	Z	Z	122	0x7A	z	z
59	0x3B	;	;	91	0x5B	[	[	123	0x7B	{	{
60	0x3C	<	<	92	0x5C	\	\\	124	0x7C		
61	0x3D	=	=	93	0x5D	]	]	125	0x7D	}	}
62	0x3E	>	>	94	0x5E	^	^	126	0x7E	~	~
63	0x3F	?	?	95	0x5F	_	_				



A.4 Rangfolge der Operatoren

Die Rangfolge der Operatoren ist im C++-Standard [ISO C++] nicht direkt spezifiziert. Sie kann aber durch die Syntax der Programmiersprache abgeleitet werden, etwa wie es hier im Buch auf Seite 116 gemacht wird, wo durch die Syntax die Regel »Punktrechnung vor Strichrechnung« gewährleistet wird. In der Tabelle A.4 bedeuten kleine Zahlen große Prioritäten.

Tabelle A.4: Präzedenz von Operatoren

Rang	Operatoren
0	::
1	. -> [] f() (Funktionsaufruf) Typ() (Typumwandlung im funktionalen Stil) ++ -- (postfix) typeid() dynamic_cast<>() static_cast<>() reinterpret_cast<>() const_cast<>()
2	sizeof ++ -- (präfix) ~ ! + - (unär) & (Adressoperator) * (Dereferenzierung) new new[] delete delete[] (Typ) Ausdruck (C-Stil-Typumwandlung)
3	.* ->*
4	* / %
5	+ - (binär)
6	<< >>
7	< > <= >=
8	== !=
9	& (bitweises UND)
10	^ (bitweises exklusiv-ODER)
11	(bitweises ODER)
12	&& (logisches UND)
13	(logisches ODER)
14	? : (Bedingungsoperator)
15	alle Zuweisungsoperatoren =, +=, <=< usw.
16	throw
17	,

Auf gleicher Prioritätsstufe wird ein Ausdruck von links nach rechts abgearbeitet mit Ausnahme der Ränge 2, 14 und 15, die von rechts abgearbeitet werden. Wegen der leichten Konvertierbarkeit zwischen char, int und bool werden mögliche Fehler nicht durch den Compiler entdeckt. Beispiele für mögliche Missverständnisse (teilweise aus [vdL]):

Operatorenrangfolge: Mögliche Missverständnisse

Tabelle A.5: Operatorenrangfolge: Mögliche Missverständnisse

Ausdruck oder Anweisung	vermutlich erwartetes Ergebnis	tatsächliches Ergebnis
<code>i = 1, 2;</code>	<code>i</code> wird 2	<code>i</code> wird 1, die 2 wird verworfen
<code>a[2, 3];</code>	<code>a[2][3]</code>	<code>a[3]</code>
<code>if(a != 2)</code>	<code>if(a != 2)</code>	<code>if(a = (!2))</code> , d.h. <code>if(false)</code>
<code>x = msb<<4 + lsb</code>	<code>x = (msb<<4) + lsb</code>	<code>x = msb << (4+lsb)</code>
<code>c = getchar() != EOF</code>	<code>(c = getchar()) != EOF</code>	<code>c = (getchar() != EOF)</code>
<code>val&mask != 0</code>	<code>(val&mask) != 0</code>	<code>val & (mask != 0)</code>
<code>a < b < c</code>	<code>a < b && b < c</code>	<code>(a < b) < c</code> (Vergleich bool mit int!)
<code>cout << a<<2</code>	<code>cout << (a<<2)</code>	<code>(cout << a) << 2</code>
<code>int *fp()</code>	<code>int (*fp)()</code> Deklaration eines Funktionszeigers	Deklaration einer Funktion, die einen Zeiger auf <code>int</code> zurückgibt

A.5 Compilerbefehle

Hier finden Sie die wichtigsten Befehle für den GNU C++-Compiler, die auch von einigen anderen Compilern verstanden werden. In der Windows-Welt ist die Endung `.exe` für ausführbare Dateien vorgesehen, in der Unix-Welt ist der Name frei wählbar, zum Beispiel könnte die Datei einfach `summe` heißen. Wenn der Name nicht vordefiniert wird, heißt die ausführbare Datei `a.out`.

```
g++ --help           die wichtigsten Optionen anzeigen
g++ --version        Compiler-Version anzeigen
g++ -c summe.cpp     nur Compilieren (summe.o wird erzeugt)
g++ -o summe summe.o Linken
g++ summe.cpp        Compilieren und Linken
```

Mehrere Dateien:

```
g++ a1.cpp main.cpp  Compilieren und Linken
```

oder einzeln

```
g++ -c a1.cpp         Compilieren
g++ -c main.cpp       Compilieren
g++ a1.o main.o       Linken, Ergebnis a.out
g++ -o main.exe a1.o main.o Linken, Ergebnis main.exe
```

Überall kann die Option `-Wall` dazugenommen werden. `W` steht für »Warnung«, `all` für »alle«. Diese Option ist empfehlenswert, weil der Compiler nicht nur Fehler, sondern auch Warnungen ausgibt, die auf syntaktisch richtigen, aber vermutlich falschen Programmcode deuten.

Eigene *include*-Verzeichnisse werden mit der Option `-I` voreingestellt. Einzelheiten sind auf Seite 128 zu finden.

Eine für ein spezielles Gebiet vorübersetzte und gepackte Bibliothek (englisch *library*) hat in der Regel einen Namen, der mit *lib* anfängt und mit *.a* aufhört. So kann eine Bibliothek zur Komprimierung von Bilddaten *libjpeg.a* heißen. Solche Bibliotheken werden mit der Option `-l` eingebunden, wobei *lib* und *.a* weggelassen werden. Beispiel:

```
g++ -o main.e a1.o main.o -ljpeg
```

Die Option `-g` fügt dem Ergebnis Informationen für den Debugger `gdb` zu. Der Debugger ist ein mächtiges Werkzeug zum Aufspüren von Fehlern, wenn alles Nachdenken versagt hat. Weitere Informationen zum Compiler oder zum Debugger erhalten Sie auf Ihrem Unix-System durch Eingabe von `info g++` oder `info gdb` bzw. `man g++` und `man gdb`.

A.6 Lösungen zu den Übungsaufgaben

Das Verzeichnis *cppbuch* der Beispiele enthält nicht nur die Beispielprogramme, sondern auch die Lösungen, und zwar im Verzeichnis *cppbuch/loesungen*. Die Kapitelnummer bestimmt den Verzeichnisnamen, die laufende Nummer der Aufgabe den entsprechenden Dateinamen. So ist die Datei *4.cpp* im Unterverzeichnis *k1* (= Kapitel 1) die Lösung zu Aufgabe 1.4. Manchmal gehören zu einer Lösung mehrere Dateien. Diese befinden sich in einem entsprechend gekennzeichneten Unterverzeichnis. Zum Beispiel enthält ein Unterverzeichnis 7 die Dateien zur Lösung von Aufgabe 7.

Die Lösungen sind nur als Vorschlag aufzufassen. Oft gibt es mehrere Lösungen, auch wenn nur eine angegeben ist. Einige wenige Programme zu den Lösungen wurden aus Platzgründen nicht abgedruckt, sind aber im Verzeichnis *cppbuch/loesungen* enthalten.

Kapitel 1

- 1.1 $\log(x)$ ist für $x \leq 0$ nicht definiert, $\sqrt{x}$ ist für ein negatives x nicht definiert. Die möglichen Ausgaben *inf* bzw. *nan* stehen für »infinity« (unendlich) bzw. »not a number« (keine gültige Zahl).

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

int main() {
    unsigned int ui = 0;
    unsigned long int uli = 0;
    cout << "max. unsigned int   = "<< ~ui      << endl;
    cout << "max. unsigned long int= "<< ~uli    << endl;
    cout << "max. int             = "<< (~ui)>>1) << endl;
    cout << "max. long int          = "<< (~uli)>>1) << endl;
}
```

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

int main() {
    int anfang;
    int ende;
    cout << "Nur ganze Zahlen eingeben:" << endl
         << "Bereichsanfang:";
    cin >> anfang;
    cout << "Bereichsende:";
    cin >> ende;
    if(anfang > ende) {
        cout << "Der Bereichsanfang darf nicht nach dem Bereichsende"
              << " liegen!" << endl;
    }
    else {
        cout << "Zahl:";
        int zahl;
        cin >> zahl;
        if(zahl >= anfang && zahl <= ende) {
            cout << zahl << " liegt im Bereich " << anfang
                 << ".." << ende << endl;
        }
        else {
            cout << zahl << " liegt nicht im Bereich " << anfang
                 << ".." << ende << endl;
        }
    }
}
```

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

int main() {
    cout << "Maximum dreier Zahlen! Eingabe: ";
    int a, b, c;
    cin >> a >> b >> c;
    cout << "Maximum = ";
    if(a > b) {
        if(a > c) {
            cout << a;
        }
        else {
            cout << c;
        }
    }
    else {
        if(b > c) {
            cout << b;
        }
        else {
            cout << c;
        }
    }
}
```

```

    }
    cout << endl;
}

```

```

1.5 #include<iostream>
using namespace std;

int main() {
    cout << "Eingabe einer Zahl: ";
    int zahl = 0;
    cin >> zahl;
    int anzahlDerBytes = sizeof zahl;
    int anzahlDerBits = 8 * anzahlDerBytes;
    cout << "binär: ";
    for(int k = anzahlDerBits-1; k >= 0 ; --k) {
        if(zahl & (1 << k)) {
            cout << "1";
        }
        else {
            cout << "0";
        }
    }
    cout << endl;
}

```

Bemerkung: Die 1 in der `if(...)`-Bedingung ist vom Typ `int`. Sie muss durch mindestens so viele Bits wie `zahl` repräsentiert werden. Wenn `zahl` als `long` deklariert werden soll, ist daher `1L` zu schreiben.

- 1.6 a) Unendliche Schleife, falls der Startwert $i > 0$ ist, weil i nicht verändert wird.
 b) Unendliche Schleife, falls $i < 0$ oder i ungerade ist. Auch in allen anderen Fällen ist die Schleife nicht besonders sinnvoll, da das Ergebnis stets $i == 0$ ist.
 c) Die Schleife terminiert nur, falls zu Beginn $i < 0$ (bei beliebigem n) ist.
- 1.7 a) Es wird die Summe der Zahlen 2...101 gebildet. Abhilfe: Anweisungen im Block vertauschen.
 b) Die geschweiften Klammern fehlen. Das Ergebnis ist 101.
 c) Korrekte Lösung. Ohne Schleife geht es auch!
 d) `sum` wird innerhalb der Schleife stets auf 0 gesetzt.
 e) Durch die vorangestellte 0 ist 0100 eine *Oktalzahl* mit dem Dezimalwert 64 (siehe auch Seite 44).

```

1.8 #include<iostream>
using namespace std;

int main() {
    int n1, n2;
    bool ungültig;
    do {
        cout << "Natürliche Zahlen n1 und n2 eingeben (n1 <= n2):";
    }
    while (ungültig);
}

```

```

    cin >> n1 >> n2;
    unguelteig = n1 < 0 || n2 < 0 || n1 > n2;
    if(unguelteig) {
        cout << "Eingabefehler!" << endl;
    }
} while(unguelteig);

// Berechne die Summe beider Zahlen
int summe = 0;
cout << "a) Summe mit for-Schleife berechnet: ";
for(int i = n1; i <= n2; ++i) {
    summe += i;
}
cout << summe << endl;
cout << "b) Summe mit while-Schleife berechnet: ";
summe = 0;
int i = n1;
while(i <= n2) {
    summe += i++;
}
cout << summe << endl;
cout << "c) Summe mit do while-Schleife berechnet: ";
summe = 0;
i = n1;
do {
    summe += i++;
} while(i <= n2);
cout << summe << endl;

cout << "d) Summe ohne Schleife berechnet: ";
summe = n2*(n2+1)/2 - (n1-1)*n1/2 ;
cout << summe << endl;
return 0;
}

```

1.9 #include<iostream>
using namespace std;

```

int main() {
    char c;
    bool zuEnde = false;
    while(!zuEnde) {
        cout << "Wählen Sie: a, b, x = Ende : ";
        cin >> c;

        switch(c) {
            case 'a': cout << "Programm a\n"; break;
            case 'b': cout << "Programm b\n"; break;
            case 'x': zuEnde = true; break;
            default : cout << "Falsche Eingabe! "
                        "Bitte wiederholen!\n";
        }
    }
}

```

```
    cout << "\n Programmende\n";
}
```

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

int main() {
    string str = "17462309"; // aus Aufgabentext
    long int z = 0;
    for(unsigned int i = 0; i < str.size(); ++i) {
        z *= 10;
        z += (int)str.at(i) - (int)'0';
    }
    cout << "z = " << z;
    int quersumme = 0;
    while(z > 0) {
        quersumme += z % 10;
        z /= 10;
    }
    cout << " Quersumme = " << quersumme << endl;
    return 0;
}
```

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

int main() {
    cout << "Umwandlung einer natürlichen Dezimalzahl in "
         << "eine römische Zahl.\n Dezimalzahl eingeben:";
    int dezimalzahl;
    cin >> dezimalzahl;
    // Position 0123456
    const string ZEICHENVORRAT("IVXLCDM");
    int zehner = 1000, n = 6; // Start mit M=1000 (Pos. 6)
    string ergebnis;

    while (dezimalzahl != 0) { // Ziffern sukzessive abtrennen
        int ziffer = dezimalzahl / zehner;
        if ((ziffer > 3 && zehner == 1000) // Tausender
            || ziffer <= 3) { // oder 0,1,2,3
            for (int i=1; i<=ziffer; i++) {
                ergebnis += ZEICHENVORRAT.at(n);
            }
        }
        else if (ziffer <= 4) { // 4
            ergebnis += ZEICHENVORRAT.at(n);
            ergebnis += ZEICHENVORRAT.at(n+1);
        }
        else if (ziffer <= 8) { // 5,6,7,8
            ergebnis += ZEICHENVORRAT.at(n+1);
            for (int i=1; i<=ziffer-5; i++) {
                ergebnis += ZEICHENVORRAT.at(n);
            }
        }
        dezimalzahl %= zehner;
        zehner /= 10;
    }
    cout << ergebnis << endl;
    return 0;
}
```

```

    }
}
else {
    ergebnis += ZEICHENVORRAT.at(n); // 9
    ergebnis += ZEICHENVORRAT.at(n+2);
}
n -= 2;
dezimalzahl %= zehner;
zehner /= 10;
}
cout << "Ergebnis: " << ergebnis << endl;
}

```

1.12

```

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

int main() {
    const int MINIMUM = -99;
    const int MAXIMUM = 100;
    const int INTERVALLZAHL = 10;
    const int INTERVALLBREITE = (MAXIMUM - MINIMUM + 1)/INTERVALLZAHL;
    int eingabe;
    vector<int> intervale(INTERVALLZAHL);
    cout << "Bitte Zahlen im Bereich " << MINIMUM
         << " bis " << MAXIMUM << " eingeben:\n";
    cin >> eingabe;

    while(eingabe >= MINIMUM && eingabe <= MAXIMUM) {
        intervale[(eingabe-MINIMUM) /INTERVALLBREITE]++;
        cin >> eingabe;
    }
    for(int i = 0; i < INTERVALLZAHL; i++) {
        cout << "Intervall "
             << i*INTERVALLBREITE + MINIMUM << ".. "
             << (i+1)*INTERVALLBREITE + MINIMUM -1 << ": "
             << intervale[i] << endl;
    }
}

```

1.13

```

#include<iostream>
using namespace std;

int main() {
    cout << "Bitte eine Startzahl > 0 eingeben: ";
    long long zahl;
    cin >> zahl;
    int iterationen = 0;
    long long maxzahl=0;
    while(zahl > 1) {
        ++iterationen;
        if(zahl % 2 == 0) {           // Zahl ist gerade

```

```

        zahl /= 2;
    }
    else {
        zahl = 3 * zahl + 1;
    }
    cout << zahl << endl;
    if(maxzahl < zahl) {
        maxzahl = zahl;
        cout << " neues Maximum. Weiter mit ENTER" << endl;
        string dummy;
        getline(cin, dummy); // weiter mit Tastendruck
    }
}
cout << iterationen << " Iterationen. Maximale Zahl ="
    << maxzahl << endl;
}

```

```

1.14 #include<iostream>
#include<string>
using namespace std;

struct Person {           // Person-Typ anlegen
    string nachname;
    string vorname;
    int alter;
};

int main() {
    Person diePerson;      // Person-Objekt anlegen
    cout << "Nachnamen eingeben: ";
    cin >> diePerson.nachname;
    cout << "Vornamen eingeben: ";
    cin >> diePerson.vorname;
    cout << "Alter eingeben: ";
    cin >> diePerson.alter;
    cout << "Die Person hat folgende Daten:" << endl;
    cout << "Nachname : " << diePerson.nachname << endl;
    cout << "Vorname  : " << diePerson.vorname << endl;
    cout << "Alter    : " << diePerson.alter << endl;
}

```



Kapitel 2

- 2.1 Aus Platzgründen wird die Lösung nicht abgedruckt. Sie liegt im Verzeichnis *cpp-buch/loesungen/k2* der Beispiele vor.
- 2.2 Die Lösung ist in der Lösung zu Aufgabe 2.3 enthalten.

```

2.3 #include <iostream>
#include <cstdlib> // für exit()
#include<string>
#include <fstream>

```

```
using namespace std;

int main() {
    ifstream quelle;
    cout << "Dateiname :";
    string Quelldateiname;
    cin >> Quelldateiname;

    quelle.open(Quelldateiname.c_str(), ios::binary|ios::in);
    if (!quelle) { // muss existieren
        cerr << Quelldateiname
              << " kann nicht geöffnet werden!\n";
        exit(-1);
    }

    unsigned char c;      // unsigned! (Bereich 0..255 statt -128..127)
    unsigned int count = 0, low, hi;
    // char ist notwendig, weil get(unsigned char) nicht implementiert ist (GNU C++).
    char cc;
    const int ZEILENLAENGE = 16;
    string buchstaben;
    string hexcodes;
    while (quelle.get(cc)) {
        c = cc;
        low = int(c) & 15;
        hi = int(c) >> 4;
        // Umsetzung der Werte 0...15 auf ASCII-Zeichen (vgl. Tabelle Seite 887)
        if (low < 10) {
            low += 48;      // '0'...'9'
        }
        else {
            low += 55;      // 'A'...'F'
        }
        if (hi < 10) {
            hi += 48;
        }
        else {
            hi += 55;
        }
        hexcodes += char(hi);
        hexcodes += char(low);
        hexcodes += ' ';
        if (c < ' ') { // nicht druckbares Zeichen
            c = '.';
        }
        buchstaben += c;
        ++count;
        count %= ZEILENLAENGE;
        if (count == 0) {
            cout << buchstaben << " " << hexcodes << endl;
            buchstaben = "";
            hexcodes = "";
        }
    }
}
```



```

    if (count != 0) { // Rest ausgeben
        cout << buchstaben;
        for(size_t i=0; i < (ZEILENLAENGE-count); ++i) {
            cout << ' ';
        }
        cout << " " << hexcodes << endl;
    }
    cout << endl;
}

```

2.4 // Datei-Statistik

```

#include<iostream>
#include<cstdlib> // für exit()

#include<fstream>
#include<string>
using namespace std;

int main() {
    ifstream quelle;
    cout << "Dateiname :";
    string Quelldateiname;
    cin >> Quelldateiname;
    quelle.open(Quelldateiname.c_str());
    if (!quelle) { // muss existieren
        cerr << Quelldateiname
            << " kann nicht geöffnet werden!\n";
        exit(-1);
    }
    char c;
    unsigned long zeichenzahl = 0, wortzahl = 0, zeilenzahl = 0;
    bool wort = false;
    while (quelle.get(c)) {
        if (c == '\n') {
            ++zeilenzahl;
        }
        else {
            ++zeichenzahl;
        }
        // Anpassung auf Umlaute fehlt noch!
        if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')){
            // Wortanfang, oder c ist in einem Wort
            wort = true;
        }
        else {
            if(wort) {
                ++wortzahl; // Wortende überschritten
            }
            wort = false;
        }
    }
    cout << "Anzahl der Zeichen (ohne Zeilenendekennung) = "
        << zeichenzahl << endl;
}

```

```

    cout << "Anzahl der Worte = " << wortzahl << endl;
    cout << "Anzahl der Zeilen = " << zeilenzahl << endl;
}

```

Die Klammern um `(c >= 'A' && c <= 'Z')` usw. sind nicht unbedingt notwendig; sie dienen der besseren Lesbarkeit.

Kapitel 3

```

3.1 #include<iostream>
    using namespace std;

    int dauerInSekunden(int stunden, int minuten, int sekunden);

    int main() {
        int std = 3;
        int m = 37;
        int sec = 40;
        cout << std << " Stunden und " << m << " Minuten und "
            << sec << " Sekunden sind insgesamt "
            << dauerInSekunden(std, m, sec) << " Sekunden."
            << endl;
    }

    int dauerInSekunden(int stunden, int minuten, int sekunden) {
        return 3600 * stunden + 60 * minuten + sekunden;
    }

```

```

3.2 #include<iostream>
    #include<cmath> // wegen pow(), s.u.
    using namespace std;

    double power(double x, int y);

    int main() {
        cout << "x^y berechnen. Zahlen x und y eingeben (y ganzzahlig):";
        double x;
        int y;
        cin >> x >> y;
        cout << "x^y = " << power(x, y) << endl;
        cout << "pow(x,y) = " << pow(x, y) << endl; // aus <cmath>
    }

    // Die Funktion power() entspricht der Funktion pow() der C++-Bibliothek <cmath>.
    double power(double x, int y) {
        double ergebnis = 1;
        bool negativ = false;
        if(y < 0) {
            y = -y;
            negativ = true;
        }
        for(int i=0; i < y; ++i) {

```

```

    ergebnis *= x;
}
if(negativ) {
    ergebnis = 1.0/ergebnis;
}
return ergebnis;
}

```

3.3

```

#include<iostream>
using namespace std;

long fakultaet(int n);

int main() {
    cout << "Ganze Zahl >= 0 eingeben: ";
    int n;
    cin >> n;
    cout << n << "! = " << fakultaet(n) << endl;
}

long fakultaet(int n) {
    if(n < 2) {
        return 1;        // Rekursionsabbruch
    }
    return n*fakultaet(n-1);
}

```

3.4

```

#include<iostream>
using namespace std;

void bewegen(int n, int a, int b, int c) {
    while (n > 0) {
        bewegen(n - 1, a, c, b);
        cout << "Bringe eine Scheibe von " << a
              << " nach " << b << endl;
        --n;
        int t = a; a = c; c = t;
    }
}

int main() {
    cout << "Türme von Hanoi! Anzahl der Scheiben: ";
    int scheiben;
    cin >> scheiben;
    bewegen(scheiben, 1, 2, 3);
}

```

3.5

```

#include<iostream>
using namespace std;

void str_umkehr(string& s);

```

```

int main() {
    cout << "Reihenfolge der Zeichen umdrehen. Zeichenkette eingeben:";
    string str;
    cin >> str;
    str_umkehr(str);
    cout << str << endl;
}

void str_umkehr(string& s) { // dreht die Reihenfolge der Zeichen um
    int links = 0, rechts = s.length() - 1;
    while(links < rechts) {
        char temp = s[links];
        s[links++] = s[rechts];
        s[rechts--] = temp;
    }
}

```

3.6 Aus Platzgründen wird die Lösung nicht abgedruckt. Sie ist aber vollständig in den Beispielen enthalten (siehe [cppbuch/loesungen/k3/5.cpp](#)).

```

3.7 #include<iostream>
using namespace std;
bool istAlphanumerisch(const string& text); // Proptotyp

int main() {
    string einText;
    cout << "Zeichenfolge eingeben:";
    getline(cin, einText);
    if(istAlphanumerisch(einText)) {
        cout << "Die eingegebene Zeichenkette enthält "
              "nur Buchstaben und Ziffern." << endl;
    }
    else {
        cout << "Die eingegebene Zeichenkette enthält NICHT "
              "nur Buchstaben und Ziffern." << endl;
    }
}

bool istAlphanumerisch(const string& text) {
    bool ergebnis = true;
    for(size_t i = 0; i < text.length(); ++i) {
        char zeichen = text.at(i);
        bool istZiffer = zeichen >= '0' && zeichen <= '9';
        bool istBuchstabe = (zeichen >= 'A' && zeichen <= 'Z')
            || (zeichen >= 'a' && zeichen <= 'z');
        // Vorzeile: && bindet stärker, die Klammern sind nur zur besseren Lesbarkeit
        if(!istZiffer && !istBuchstabe) {
            ergebnis = false;
            break; // weitere Prüfungen sind nicht notwendig
        }
    }
    return ergebnis;
}

```

3.8 Die Lösung ist einfach, wenn wir bedenken, dass es sich um eine bloße *Text-ersetzung* handelt. `QUAD(x+1)` würde *ohne* die Klammern `x+1*x+1` und damit ein falsches arithmetisches Ergebnis liefern. *Mit* Klammern gibt es in diesem Fall keine Probleme: `((x+1)*(x+1))` (siehe jedoch Seite 130). Die äußeren Klammern sind wichtig, damit `QUAD(x)` in einem zusammengesetzten Ausdruck verwendet werden kann.

3.9 • *taschenrechner.h*

```
#ifndef TASCHENRECHNER_H
#define TASCHENRECHNER_H
long ausdruck(char& c);
long summand(char& c);
long faktor(char& c);
long zahl(char& c);
#endif
```

Diese Datei wird mit `#include` in *main.cpp* und *taschenrechner.cpp* eingebunden. Wegen der sehr großen Ähnlichkeit des Restes der Lösung mit der auf den Seiten 118 ff. abgedruckten wird hier aus Platzgründen auf eine Wiedergabe verzichtet. In den Beispielen (Verzeichnis *cppbuch/loesungen/k3/8*) ist das Programm vollständig vorhanden.

3.10 • *gettype.t*

```
#ifndef GETTYPE_T
#define GETTYPE_T
#include<string>
using std::string;

// Template
template<typename T>
string getType(T t) { return "unbekannter Typ";}

// Template-Spezialisierungen
template<> string getType(int t) { return "int";}
template<> string getType(unsigned int t) { return "unsigned int";}
template<> string getType(double t) { return "double";}
template<> string getType(char t) { return "char";}
template<> string getType(bool t) { return "bool";}
#endif
```

• *main.cpp*

```
#include<iostream>
#include"gettype.t"
using namespace std;

int main() {
    int i;
    cout << getType(i) << endl;
    unsigned int ui;
    cout << getType(ui) << endl;
```

```

float f; // nicht in getType() berücksichtigt!
cout << getType(f) << endl;
double d;
cout << getType(d) << endl;
char c;
cout << getType(c) << endl;
bool b;
cout << getType(b) << endl;
}

```

3.11 • *betrag.t*

```

#ifndef BETRAG_T
#define BETRAG_T
#include<iostream>
#include<cstdlib> // für exit()

// Template
template<typename T>
T betrag(T t) {
    return (t < 0) ? -t : t;
}

// Template-Spezialisierung
template<> char betrag(char c) {
    std::cerr << "Betrag von 'char' ist undefiniert" << std::endl;
    exit(1);
    return c; // damit der Compiler zufrieden ist (wg. exit() nicht erreichbar)
}

// Template-Spezialisierung
template<> bool betrag(bool b) {
    std::cerr << "Betrag von 'bool' ist undefiniert" << std::endl;
    exit(1);
    return b;
}
#endif

```

• *main.cpp*

```

#include<iostream>
#include"betrag.t"
using namespace std;
int main() {
    int i = -1;
    cout << "Der Betrag von " << i << " ist " << betrag(i) << endl;
    double d = -2.345;
    cout << "Der Betrag von " << d << " ist " << betrag(d) << endl;
    // Fehlermeldung provozieren
    bool b = true;
    cout << "Der Betrag von " << b << " ist " << betrag(b) << endl;
}

```

- 3.12 Der erste Fehler steckt in der Anweisung `temp = feld[0];`, weil diese Anweisung die Existenz von mindestens einem Vektorelement voraussetzt. Die Funktion würde

bei einem *leeren* Vektor versagen und möglicherweise »crashen«. Der zweite Fehler ist nicht ganz so leicht zu finden. Die Funktion sortiert einwandfrei, wenn alle Elemente verschieden sind, nicht aber, wenn es gleiche Elemente gibt und die auch noch die größten sind. Zum Beispiel wird die Folge 1200, 1200, 38, 1, 0, 3, 99, 1010, 4 nicht korrekt sortiert. Der Algorithmus verwendet die Überlegung: Nur *eine* Vertauschung ändert schon temp, weswegen es als Indikator genommen werden kann. Der Fehler: Dies gilt nicht, wenn nach der letzten Vertauschung temp genau den Wert hat, den auch `feld[0]` hat (größtes Element). Die Behauptung im Quellcode »// keine Vertauschung mehr« und auch die Argumentation im Aufgabentext sind also falsch.

- 3.13 Um Mehrfachberechnungen der Potenzen von x zu vermeiden, wird das Polynom durch geschickte Klammerung umformuliert:

$(((((\dots k_3)x + k_2)x + k_1)x + k_0)$ (sogenanntes Horner-Schema).

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

double polynom(const vector<double>& koeff, double x) {
    int n = koeff.size()-1;
    double ergebnis = koeff[n];
    for(int i = n-1; i >= 0 ; --i) {
        ergebnis *= x;
        ergebnis += koeff[i];
    }
    return ergebnis;
}

int main() {
    vector<double> koeffizienten(3);
    koeffizienten[0] = 1.1;
    koeffizienten[1] = 2.22;
    koeffizienten[2] = 13.0;
    cout << polynom(koeffizienten, 2.04) << endl;
    cout << polynom(koeffizienten, 3.033) << endl;
}
```

- 3.14 /* Dieses Programm listet sich selbst */

```
#include <string>
#include <iostream>
using namespace std;
char AS = 34;      // Anführungsstriche
char BS = 92;      // Backslash
char NZ = 10;      // neue Zeile
void c(const string& t) {
    cout << t << AS;
    unsigned int i = 0;
    while (i < t.length()) {
        if (t[i] == NZ) cout << BS << 'n'<<AS<< t[i] << AS;
        else cout << t[i];
        ++i;
    }
}
```

```

    }
    cout<<AS<<')<<';<<'c'<<'('<<'s'<<')<<';<<'}<<NZ;
}
int main(){ string s("/ * Dieses Programm listet sich selbst */\n"
#include <string>\n"
#include <iostream>\n"
using namespace std;\n"
"char AS = 34;      // Anführungsstriche\n"
"char BS = 92;      // Backslash\n"
"char NZ = 10;      // neue Zeile\n"
"void c(const string& t) {\n"
"    cout << t << AS;\n"
"    unsigned int i = 0;\n"
"    while (i < t.length()) {\n"
"        if (t[i] == NZ) cout << BS << 'n'<<AS<< t[i] << AS;\n"
"        else cout << t[i];\n"
"        ++i;\n"
"    }\n"
"    cout<<AS<<')<<';<<'c'<<'('<<'s'<<')<<';<<'}<<NZ;\n"
"}\n"
"int main(){ string s(""); c(s); }

```

- 3.15 Eine unsigned-Zahl ist immer größer oder gleich 0, deswegen kann der erste Teil der Bedingung nie wahr werden. Eine unsigned-Zahl kann niemals größer als `UINT_MAX` sein, weil `UINT_MAX` per Definition die größte unsigned-Zahl ist. Damit kann auch der zweite Teil der Bedingung nie wahr werden. Die Funktion ist sinnlos.

Kapitel 4

```

4.1 Rational add(long a, const Rational& b) {
    Rational r(a);
    r.add(b);
    return r;
}

Rational add(const Rational& a, long b) {
    return add(b, a); // Aufruf von add(long, const Rational&)
}

```

```

4.2 void ausgabeEinerRationalzahl(const Rational& r) {
    std::cout << r.Zaehler() << "/" << r.Nenner();
    std::cout << std::endl;
}

```

4.3 Lösungsbeispiel

• IntMenge.h

```

// Klasse zur Implementierung eines Datentyps für Mengen mit int-Elementen
#ifndef IntMenge_h
#define IntMenge_h
#include <cstdint> // size_t

```



```
#include<vector>

class IntMenge {
public:
    IntMenge();
    void hinzufuegen(int el);
    void entfernen(int el);
    bool istMitglied(int el) const;
    size_t size() const;
    void anzeigen() const;
    void loeschen(); // alle Elemente löschen
    int getMax() const; // größtes Element
    int getMin() const; // kleinstes Element
private:
    size_t anzahl;
    std::vector<int> vec;
    // Die Hilfsfunktion finden() gibt die Position des
    // Elements zurück. -1 bedeutet nicht vorhanden
    int finden(int el) const;
};
#endif
```

Die private Hilfsfunktion `finden(int el)` gibt die Position des Elements `el` zurück. Sie wird intern zur Vermeidung von Code-Duplizierung verwendet. Wenn es sie nicht gäbe, müssten `entfernen()` und `istMitglied()` mit einer Schleife versehen werden.

- *IntMenge.cpp*

```
#include "IntMenge.h"
#include<iostream>
#include<cassert>

IntMenge::IntMenge()
    : anzahl(0) {
}
```

Die folgende Methode `hinzufuegen()` nutzt aus, dass ein `vector` dynamisch mit `push_back()` vergrößerbar ist. Die Variable `anzahl` gibt die tatsächliche Anzahl der gespeicherten Elemente an. Sie kann kleiner als die Größe des Vektors sein, nämlich dann, wenn Elemente gelöscht worden sind.

```
void IntMenge::hinzufuegen(int el) {
    if(!istMitglied(el)) { // ansonsten ignorieren
        if(anzahl < vec.size()) {
            vec[anzahl] = el;
        }
        else { // Platz reicht nicht
            vec.push_back(el);
        }
        ++anzahl;
    }
}
```

Ein Element wirklich zu löschen, hieße den Vektor zu verkleinern – eine zeitaufwendige Operation. Da die Reihenfolge der Elemente in einer Menge nicht sortiert sein muss, bietet sich stattdessen an, das letzte Element an die Stelle des zu löschenden zu kopieren. Wenn dann noch `anzahl` um eins heruntergezählt wird, ist das vorherige letzte Element nicht mehr erreichbar, denn alle Schleifen in den folgenden Methoden haben `anzahl` als Grenze. Der freigewordene Platz steht für `hinzufuegen()` zur Verfügung. Nach dieser Logik ist auch das Löschen aller Elemente denkbar schnell und einfach: `anzahl` wird auf 0 gesetzt (siehe Methode `loeschen()`).

```
void IntMenge::entfernen(int el) {
    int wo = finden(el);
    if(wo > -1) {
        vec[wo] = vec[--anzahl]; // letztes Element umkopieren
    }
}

bool IntMenge::istMitglied(int el) const {
    return finden(el) > -1;
}

size_t IntMenge::size() const {
    return anzahl;
}

void IntMenge::anzeigen() const {
    for(size_t i=0; i < anzahl; ++i) {
        std::cout << vec[i] << " ";
    }
    std::cout << std::endl;
}

void IntMenge::loeschen() {
    anzahl = 0;
}

int IntMenge::finden(int el) const {
    for(size_t i=0; i < anzahl; ++i) {
        if(vec[i] == el)
            return i;
    }
    return -1; // nicht gefunden
}

int IntMenge::getMin() const {
    assert(anzahl > 0);
    int erg = vec[0];
    for(size_t i=1; i < anzahl; ++i) {
        if(vec[i] < erg)
            erg = vec[i];
    }
    return erg;
}
```

```
int IntMenge::getMax() const {
    assert(anzahl > 0);
    int erg = vec[0];
    for(size_t i=1; i < anzahl; ++i) {
        if(vec[i] > erg)
            erg = vec[i];
    }
    return erg;
}
```

Man kann sich noch einige Optimierungen vorstellen. Zum Beispiel könnte der erste Aufruf von `getMin()` oder `getMax()` sowohl Minimum als auch Maximum ermitteln, und die Werte könnten in entsprechenden Attributen gespeichert werden (Cache). Eine zweite Abfrage würde dann einen gespeicherten Wert zurückgeben und wäre damit sehr schnell. Ein erneutes Durchlaufen der Schleife wäre nur beim Hinzufügen oder Entfernen fällig und auch nur, wenn Minimum oder Maximum betroffen wären. Auch kann man sich überlegen, dass die Schleifen überhaupt zu aufwendig sind – dann bräuchte man allerdings eine andere Datenstruktur. Die Klasse `set` der C++-Bibliothek verwendet deshalb eine Variante des binären Suchbaums.

- 4.4 Die mehrseitige Lösung wird aus Platzgründen nicht abgedruckt. Sie ist aber vollständig in den Beispielen (Verzeichnis *cppbuch/loesungen/k4/4*) enthalten.
- 4.5 Man kann im `public`-Bereich eine Referenz auf `const` einfügen, die auf ein `private` Attribut verweist. Da man einer Referenz nichts zuweisen kann, muss sie im Konstruktor initialisiert werden.

```
#include<iostream>

class MeineKlasse {
public:
    MeineKlasse()
        : readonlyZahl(privateZahl), // Initialisierung der Referenz
          privateZahl(0) {
    }

    void aendern(int wert) {
        privateZahl = wert;
    }

    // public-Referenz auf Konstante, Initialisierung im Konstruktor
    const int& readonlyZahl;
private:
    int privateZahl;
};

using namespace std;

int main() {
    MeineKlasse objekt;
    // objekt.privateZahl = 999; // Fehler! Zugriff nicht möglich!
    // objekt.readonlyZahl = 999; // Fehler! Änderung nicht möglich!
```

```

    objekt.aendern(999);          // erlaubte Änderung
    // erlaubter direkter lesender Zugriff:
    cout << "objekt.readonlyZahl=" << objekt.readonlyZahl << endl; // 999
}

```

4.6 • *taschenrechner.h*

```

#ifndef TASCHENRECHNER_H
#define TASCHENRECHNER_H
#include<string>

class Taschenrechner {
public:
    Taschenrechner(const std::string&);
    const std::string& getAnfrage();
    long getErgebnis();
private:
    long ausdruck(char& c);
    long summand(char& c);
    long faktor(char& c);
    long zahl(char& c);
    void get(char& c);
    std::string anfrage;
    size_t position;
    long ergebnis;
};
#endif

```

• *taschenrechner.cpp*

```

#include"taschenrechner.h"
#include<cctype>
#include<iostream>

Taschenrechner::Taschenrechner(const std::string& str)
    : anfrage(str), position(0), ergebnis(0L) {
    char c;
    get(c); // 1. Zeichen lesen
    ergebnis = ausdruck(c);
}

const std::string& Taschenrechner::getAnfrage() {
    return anfrage;
}

long Taschenrechner::getErgebnis() {
    return ergebnis;
}

void Taschenrechner::get(char& c) {
    do {
        if(position >= anfrage.length()) {
            c = '#'; // ungültiges Zeichen, d.h. Abbruch
        }
    }
}

```

```

        else {
            c = anfrage[position++];
        }
    } while(c == ' '); // Leerzeichen ignorieren
}

long Taschenrechner::ausdruck(char& c) { // Übergabe per Referenz!
// Der weggelassene Rest ist wie auf Seite 119, nur dass get(c);
// statt cin.get(c); verwendet wird.
// ...
    return a;
}

```

Aus Platzgründen und weil die Struktur nach vorstehendem Muster klar ist, wurden die restlichen Funktionen weggelassen. In den Beispielen (Verzeichnis *cppbuch/loesungen/k4/6*) ist das Programm vollständig vorhanden.

Kapitel 5

- 5.1 Ja. Aus der Gleichheit von $(kosten+i)$ und $(i+kosten)$ und aus der Kenntnis, dass der Compiler *stats* die Umwandlung in die Zeigerdarstellung von $[]$ vornimmt, folgt, dass man genauso gut $i[kosten]$ statt $kosten[i]$ formulieren kann. Es ist jedoch absolut unüblich und erschwert die Lesbarkeit des Programms.
- 5.2 $sizeof(int)*dim1*dim2 = 24$, $Bytenummer = (i*dim2+j)*sizeof(int)$. Daraus ergibt sich, dass $dim1$ nur zur Berechnung des Speicherplatzes gebraucht wird, aber nicht zur Adressberechnung, zu der jedoch alle weiteren Dimensionen benötigt werden.
- 5.3 Es muss $m = p$, $r = n$ und $s = q$ gelten, damit die Matrizenmultiplikation definiert ist. Daher benötigt man nur noch drei Konstanten. Für die Funktion `tabellenausgabe2D()` siehe Seite 212.

```

int main() {
    // Initialisierung (Beispiel)
    const int N = 2, M = 3, Q = 4;
    int a[N][M] = {{1,2,3}, {4,5,6}};
    int b[M][Q] = {{1,2,3,0}, {4,1,1,5}, {1,7,1,4}};
    int c[N][Q];
    for(int i = 0; i < N; ++i) { // Multiplikation
        for(int j = 0; j < Q; ++j) {
            c[i][j] = 0;
            for(int k = 0; k < M; ++k) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    // Ergebnis ausgeben oder weiterrechnen
    tabellenausgabe2D(a, N);
    cout << "multipliziert mit " << endl;
    tabellenausgabe2D(b, M);
    cout << "ergibt" << endl;
    tabellenausgabe2D(c, N);
}

```

```
// ...
}
```

- 5.4 Da eine echte dreidimensionale Ausgabe in der Ebene nicht möglich ist, werden n (DIM2 x DIM3)-Matrizen ausgegeben.

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

template<typename Feldtyp>
void Tabellenausgabe3D(Feldtyp T, size_t n) {
    const size_t DIM2 = sizeof T[0] /sizeof T[0][0];
    const size_t DIM3 = sizeof T[0][0] /sizeof T[0][0][0];
    for(size_t i = 0; i < n; ++i) {
        for(size_t j = 0; j < DIM2; ++j) {
            for(size_t k = 0; k < DIM3; ++k)
                cout << T[i][j][k] << ' ';
            cout << endl;
        }
        cout << endl;
    }
    cout << endl;
}

int main() {
    const int N = 2, M = 3, Q = 4;
    int mat3D[N][M][Q]; // 3D-Matrix
    // Mit Werten füllen
    int m = 0;
    for(int i = 0; i < N; ++i) {
        for(int j = 0; j < M; ++j) {
            for(int k = 0; k < Q; ++k)
                mat3D[i][j][k] = ++m;
        }
    }
    Tabellenausgabe3D(mat3D, N);
}
```

- 5.5 Die Funktion entspricht der Funktion strcpy() der C++-Standardbibliothek.

```
void strcpy(char *ziel, const char *quelle) {
    // kopiert den Inhalt von quelle in den String ziel
    // (und überschreibt den vorherigen Inhalt dabei).
    while((*ziel++ = *quelle++));
}
```

- 5.6
- ```
char* strduplikat(const char *s) {
 // liefert einen Zeiger auf den neu erzeugten String.
 // ACHTUNG: Der Aufrufer ist für das delete verantwortlich!
 char* neu = new char[strlen(s)+1];
 strcpy(neu, s); // wie strcpy() von oben
 return neu;
}
```

```

5.7 // Vergleichsfunktion für C-String-Array-Elemente
int scmp(const void *a, const void *b) {
 // Umwandlung in einen const-Zeiger auf einen C-String, dh. auf const char *
 const char *pa = *static_cast<const char* const*>(a);
 const char *pb = *static_cast<const char* const*>(b);
 return strcmp(pa, pb);
}

```

Quicksort wird ähnlich wie im Textbeispiel aufgerufen. Da wir ein Feld von Zeigern vor uns haben, wird als Elementgröße die Größe eines Zeigers auf char übergeben: `qsort(sfeld, size, sizeof(char*), scmp);`.

Alternative Lösung mit der Standardfunktion `std::sort()`:

```

#include<iostream>
#include<cstring>
#include<string>
#include<algorithm> // enthält sort()
using namespace std;

// Vergleichsfunktion für C-String-Array-Elemente
bool scmp(const char *a, const char *b) {
 return strcmp(a, b) < 0;
}

int main() {
 const char* sfeld[] = {"eins", "zwei", "drei", "vier", "fünf",
 "sechs", "sieben", "acht", "neun", "zehn"};
 size_t anzahlElemente = sizeof(sfeld)/sizeof(sfeld[0]);
 std::sort(sfeld, sfeld + anzahlElemente, scmp);
 // ALPHABETISCHE Ausgabe des sortierten Feldes:
 for(size_t i = 0; i < anzahlElemente; ++i)
 cout << ' ' << sfeld[i];
 cout << endl;

 // Entsprechend für C++-Strings
 string strings[] = {"eins", "zwei", "drei", "vier", "fünf",
 "sechs", "sieben", "acht", "neun", "zehn"};
 size_t anzahl = sizeof(sfeld)/sizeof(sfeld[0]);
 // bei C++-Strings ist keine Vergleichsfunktion notwendig
 std::sort(strings, strings + anzahl);
 // ALPHABETISCHE Ausgabe des sortierten Feldes:
 for(size_t i = 0; i < anzahl; ++i)
 cout << ' ' << strings[i];
 cout << endl;
}

```

```

5.8 void LeerzeichenEntfernen(char* s) {
 char* q = s;
 do {
 if(*s != ' ') {
 *q++ = *s;
 }
 }
}

```

```
 } while(*s++);
}
```

5.9

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

int main(int argc, char* argv[]) {
 cout << "Dateien ausgeben" << endl;
 if(argc == 1) {
 cout << "Keine Dateinamen in der Kommandozeile gefunden.\n"
 << "Gebrauch: " << argv[0] // Programmname
 << " datei1 datei2 usw." << endl;
 return 0;
 }
 int nr = 0;
 while(argv[++nr] != 0) {
 ifstream quelle;
 quelle.open(argv[nr], ios::binary|ios::in);
 cout << "Datei " << argv[nr];
 if(!quelle) { // Fehlerabfrage
 cout << " nicht gefunden." << endl;
 continue; // weiter bei while
 }
 cout << ":" << endl;
 char ch;
 while(quelle.get(ch)) {
 cout << ch; // zeichenweise ausgeben
 }
 quelle.close();
 }
}
```

## 5.10 Ausgabe von Namen in einer Datei

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

bool istBuchstabe(char c) { // vgl. isalpha(), Seite 875
 return c >= 'A' && c <= 'Z'
 || c >= 'a' && c <= 'z'
 || c == '_';
}

bool istAlphanumerisch(char c) { // vgl. isalnum(), Seite 875
 return c >= '0' && c <= '9'
 || istBuchstabe(c);
}

int main(int argc, char* argv[]) {
 if(argc == 1) {
 cout << "Kein Dateiname in der Kommandozeile gefunden."
```



```

 " Gebrauch: " << argv[0] // Programmname
 << " dateiname" << endl;
 return 0;
}
ifstream quelle(argv[1]);
if(!quelle) { // Fehlerabfrage
 cout << "Datei " << argv[1] << " nicht gefunden." << endl;
 return 0;
}
char ch;
bool namengefunden = false;
while(quelle.get(ch)) {
 if(istBuchstabe(ch)) {
 cout << ch;
 namengefunden = true;
 }
 else if(namengefunden && istAlphanumerisch(ch)) {
 cout << ch;
 }
 else if(namengefunden) {
 namengefunden = false;
 cout << endl;
 }
}
quelle.close();
}

```

## Kapitel 6

```

6.1 void MeinString::insert(size_t pos, const MeinString& m) {
 // m vor pos einfügen
 if(pos > len) {
 pos = len;
 }
 reserve(len + m.len);
 // Teil hinter pos verschieben
 size_t neuesende = len + m.len;
 for(size_t anz = 0; anz <= len-pos; ++anz) {
 start[neuesende] = start[neuesende-m.len];
 --neuesende;
 }
 // m einfügen
 const char* temp = m.start;
 while(*temp) {
 start[pos++] = *temp++;
 }
 len = len + m.len; // Verwaltungsinformation aktualisieren
}

```

### 6.2 • *format.h*

```

#ifndef FORMAT_H

```

```
#define FORMAT_H
#include<string>
using std::string;

class Format {
public:
 Format(int weite, int nachk);
 string toString(double d) const;
private:
 int weite;
 int nachkommastellen;
};
#endif
```

• *format.cpp*

```
#include "format.h"
#include<iostream>
using namespace std;

Format::Format(int w, int nk)
 : weite(w), nachkommastellen(nk) {
 if(nk < 0) nk = 0;
 if(nk > 16) nk = 16;
 if(w < nk) w = nk+1;
}

string Format::toString(double d) const {
 string ergebnis;
 bool negativ = false;
 if(d < 0.0) {
 negativ = true;
 d = -d;
 }
 // Rundung
 double rund = 0.5;
 for(int i=0; i < nachkommastellen; ++i)
 rund /= 10.0;
 d += rund;
 // Mit der folgenden Normierung (d.h. Zahl beginnt mit 0,...) wird erreicht, dass
 // die Anzahl der Stellen vor dem Komma bekannt ist (Stellenwert).
 int stellenwert = 0;
 // Zahl normieren, falls >=1
 while(d >= 1.0) {
 ++stellenwert;
 d /= 10.0;
 }
 if(stellenwert == 0) {
 ergebnis += '0'; // wenigstens eine 0 vor dem Komma
 }

 // Die Zahl wird sukzessive mit 10 multipliziert, die jeweils erste Ziffer
 // zunächst ermittelt (zif), dann abgetrennt und an den Ergebnis-String
```

```

// gehängt usw.
do {
 if(stellenwert == 0) {
 ergebnis += ','; // Komma
 }
 d *= 10.0;
 int zif = (int)d;
 d -= zif;
 ergebnis += (char)zif + (int)'0';
 --stellenwert;
} while(nachkommastellen + stellenwert > 0);
if(negativ) {
 ergebnis = '-' + ergebnis;
}
int diff = weite - ergebnis.length();
for(int i=0; i < diff; ++i) {
 ergebnis = " " + ergebnis;
}
return ergebnis;
}

```

### 6.3 • *teilnehmer.h*

Bei der Speicherung in einem `vector<Teilnehmer*>` müssen alle verbundenen Teilnehmer im selben Gültigkeitsbereich sein! Der Grund: Wenn die Lebensdauer unterschiedlich ist, können ungültige Referenzen entstehen. Beispiel:

```

Teilnehmer otto("Otto");
{
 Teilnehmer andrea("Andrea");
 otto.lerntKennen(andrea); // alles bestens
}
otto.druckeBekannte(); // ups! Andrea ist futsch!

```

Aus diesem Grund ist es günstiger, nur die Namen zu speichern.

```

#ifndef TEILNEHMER_H
#define TEILNEHMER_H
#include<string>
#include<vector>
using std::string;
using std::vector;

class Teilnehmer {
public:
 Teilnehmer(const string& name);
 void lerntKennen(Teilnehmer& tn);
 bool kennt(const Teilnehmer& tn) const;
 void druckeBekannte() const;
 const string& gibNamen() const;
private:
 string name;
 vector<string> dieBekannten;
};

```

```
#endif
```

• *teilnehmer.cpp*

```
#include "teilnehmer.h"
#include <iostream>
using std::cout;
using std::endl;

Teilnehmer::Teilnehmer(const string& n)
 : name(n) {

}

void Teilnehmer::lerntKennen(Teilnehmer& tn) {
 if(&tn != this // 'sich selbst kennenlernen' ignorieren
 && !kennt(tn)) { // wenn noch unbekannt, eintragen
 dieBekannten.push_back(tn.gibNamen());
 tn.lerntKennen(*this); // wechselseitig kennenlernen
 }
}

bool Teilnehmer::kennt(const Teilnehmer& tn) const {
 bool erg = false;
 for(size_t i = 0; i < dieBekannten.size(); ++i) {
 if(tn.gibNamen() == dieBekannten.at(i)) {
 erg = true;
 break;
 }
 }
 return erg;
}

void Teilnehmer::druckeBekannte() const {
 for(size_t i = 0; i < dieBekannten.size(); ++i) {
 cout << " " << dieBekannten.at(i);
 }
 cout << endl;
}

const string& Teilnehmer::gibNamen() const {
 return name;
}
```

## Kapitel 7

7.1 Nein. Die Funktion kann nicht mehr von `GraphObj` geerbt werden, ohne dass `Strecke` abstrakt wird. Für die Klasse `Strecke` muss eine überladene Elementfunktion `flaeche()` mit dem Rückgabewert 0 geschrieben werden.

7.2 • *person.h*

```
#ifndef PERSON_H
#define PERSON_H
```

```

#include<string>
using std::string;

class Person {
public:
 Person(const string& n, const string& v)
 : nachname(n), vorname(v) {
 }
 const string& getNachname() const { return nachname;}
 const string& getVorname() const { return vorname;}
 virtual string toString() const = 0;
 virtual ~Person() {}
private:
 string nachname;
 string vorname;
};
// Die Standardimplementierung einer rein virtuellen Methode
// muss nach [ISOC++] außerhalb der Klassendefinition stehen:
inline string Person::toString() const {
 return vorname + " " + nachname;
}
#endif

```

- *student.h*

```

#ifndef STUDENT_H
#define STUDENT_H
#include "person.h"
#include<string>
using std::string;

class StudentIn : public Person {
public:
 StudentIn(const string& name, const string& vorname,
 const string& matnr)
 : Person(name, vorname), matrikelnummer(matnr) {
 }
 const string& getMatrikelnummer() const {
 return matrikelnummer;
 }
 virtual string toString() const {
 return "Student/in " + Person::toString()
 + ", Mat.Nr.: " + matrikelnummer;
 }
 virtual ~StudentIn() {}
private:
 string matrikelnummer;
};
#endif

```

- *prof.h*

```

#ifndef PROF_H
#define PROF_H

```

```

#include "person.h"
#include <string>
using std::string;

class ProfessorIn : public Person {
public:
 ProfessorIn(const string& nachname, const string& vorname,
 const string& lgb)
 : Person(nachname, vorname), Lehrgebiet(lgb) {
 }
 const string& getLehrgebiet() const {
 return Lehrgebiet;
 }
 virtual string toString() const {
 return "Prof. " + Person::toString()
 + ", Lehrgebiet: " + Lehrgebiet;
 }
 virtual ~ProfessorIn() {}
private:
 string Lehrgebiet;
};
#endif

```

- 7.3 Da im obigen Programm Zeiger auf Person verwendet werden, erfordert ein Zugriff auf Methoden, die nicht in Person deklariert sind, eine Typumwandlung. Beispiel:

```

cout << "Die Matrikelnummer von "
 << diePersonen[0]->getNachname() << " ist "
 << ((StudentIn*)diePersonen[0])->getMatrikelnummer() // !
 << endl;

```

Die Typumwandlung in den Typ `StudentIn*` funktioniert natürlich nur, wenn man genau weiß, dass der Zeiger an der Stelle `[0]` auf ein Objekt des dynamischen Typs `StudentIn` verweist. Was aber, wenn man es nicht genau weiß? Dazu geben die Abschnitte 7.9 und 7.10 Auskunft.

- 7.4
- ```

cout << "Die Matrikelnummern mit dynamic_cast: " << endl;
for(size_t i = 0; i < diePersonen.size(); ++i) {
    cout << diePersonen[i]->getVorname() << ": ";
    StudentIn* ps = dynamic_cast<StudentIn*>(diePersonen[i]);
    if(ps) {
        cout << ps->getMatrikelnummer() << endl;
    }
    else {
        cout << " hat keine Matrikelnummer." << endl;
    }
}

```

- 7.5
- ```

cout << endl << "Die Matrikelnummern mit typeid: " << endl;
for(size_t i = 0; i < diePersonen.size(); ++i) {
 cout << diePersonen[i]->getVorname();
 if(typeid(StudentIn) == typeid(*diePersonen[i])) {

```

```

 cout << ": "
 << ((StudentIn*)(diePersonen[i]))->getMatrikelnummer()
 << endl;
 }
 else {
 cout << " (interner Typ: "
 << typeid(*diePersonen[i]).name()
 << ") hat keine Matrikelnummer." << endl;
 }
}

```

## Kapitel 9

- 9.1 Die Referenz auf den Rational-Parameter in der Deklaration darf nicht const sein, weil das Objekt verändert wird:

```

// Deklaration als globale Funktion
std::istream& operator>>(std::istream&, Rational&);

// Implementation
std::istream& operator>>(std::istream& eingabe, Rational& r){
 // cerr wird gewählt, damit die Abfragen auch dann
 // auf dem Bildschirm erscheinen, wenn die Standard-
 // Ausgabe in eine Datei zur Dokumentation geleitet wird.
 int z, n;
 std::cerr << "Zähler :";
 eingabe >> z;
 std::cerr << "Nenner :";
 eingabe >> n;
 assert(n != 0); // nicht sehr benutzungsfreundlich ...
 r.set(z, n);
 r.kuerzen();
 return eingabe;
}

```

Anmerkung: Hier wurde die Methode `eingabe()` mit dem Operator nachgebildet. Der bessere Programmierstil ist, die Funktionen der Ein- und Ausgabe zu trennen, sodass die Aufforderung zur Zahleneingabe nicht Bestandteil des Eingabeoperators ist.

- 9.2 Der Operator `+=` verändert das Objekt selbst, denn `a += b;` ist nur eine Abkürzung für `a = a+b;`. Daher kann er, vordergründig betrachtet, als Elementfunktion mit nur einem Argument und Rückgabetypp `void` deklariert werden:

```
void operator+=(Rational);
```

Die Implementierung könnte wie folgt aussehen:

```

void Rational::operator+=(Rational b) { // nicht optimal
 zaehler = zaehler*b.nenner + b.zaehler*nenner;
 nenner = nenner*b.nenner;
 kuerzen();
}

```

Um Verkettungen wie  $c = a += b$ , die zu  $c = a.operator+=(b)$  aufgelöst werden sowie die Verwendung innerhalb des binären `operator+()` zu erlauben, muss ein Objekt des passenden Datentyps zurückgegeben werden, also ein Objekt der Klasse `Rational` (statt `void` wie vorher). Um die Konstruktion von temporären Objekten durch den Kopierkonstruktor bei der ErgebnISRückgabe zu vermeiden, wird die Referenz auf das Zielobjekt zurückgegeben. Die Referenz auf `const` in der Parameterliste erspart die Kopie beim Eintritt in die Funktion.

```
Rational& Rational::operator+=(const Rational& b) {
 zaehler = zaehler*b.nenner + b.zaehler*nenner;
 nenner = nenner*b.nenner;
 kuerzen();
 return *this;
}
```

### 9.3 Deklaration in *ratioop.h* als Elementfunktion:

```
Rational& operator--(const Rational&);
Rational& operator*=(const Rational&);
Rational& operator/=(const Rational&);
```

Ebenfalls in *ratioop.h* werden die binäre Operatoren als globale Funktionen deklariert. Dabei wird die Empfehlung von Seite 168 beachtet, den ersten Parameter per Wert zu übergeben, weil in der Funktion eine Kopie gebraucht wird. Das `const` bei dem Rückgabetypp verhindert unsinnige Anweisungen wie  $(a + b) = c$ .

```
// globale Operatoren
const Rational operator+(Rational, const Rational&);
const Rational operator-(Rational, const Rational&);
const Rational operator*(Rational, const Rational&);
const Rational operator/(Rational, const Rational&);
```

### Definition in *ratioop.cpp* als Elementfunktion:

```
Rational& Rational::operator--(const Rational& b) {
 zaehler = zaehler*b.nenner - b.zaehler*nenner;
 nenner = nenner*b.nenner;
 kuerzen();
 return *this;
}
```

```
Rational& Rational::operator*=(const Rational& b) {
 zaehler *= b.zaehler;
 nenner *= b.nenner;
 kuerzen();
 return *this;
}
```

```
Rational& Rational::operator/=(const Rational& b) {
 zaehler *= b.nenner;
 nenner *= b.zaehler;
 kuerzen();
 return *this;
}
```



Definition der globalen Funktionen in *ratioop.cpp*:

```
const Rational operator+(Rational a, const Rational& b) {
 return a += b;
}

const Rational operator-(Rational a, const Rational& b) {
 return a -= b;
}

const Rational operator*(Rational a, const Rational& b) {
 return a *= b;
}

const Rational operator/(Rational a, const Rational& b) {
 return a /= b;
}
```

#### 9.4 Deklaration in *ratioop.h* als globale Funktion:

```
bool operator==(const Rational&, const Rational&);
```

Definition in *ratioop.cpp* als globale Funktion:

```
bool operator==(const Rational& a, const Rational& b) {
 return a.getZaehler() == b.getZaehler()
 && a.getNenner() == b.getNenner();
}
```

Es wird hier angenommen, dass beide Zahlen in der gekürzten Darstellung vorliegen, weil dies durch die Elementfunktionen erzwungen wird. Andernfalls müssten beide Argumente vor dem Vergleich gekürzt werden.

#### 9.5 `operator=()` darf *nichts* tun. Schließlich darf die `SerialNr` als Konstante eines Objekts nicht verändert werden. Der Sinn des Operators besteht nur darin, Zuweisungsoperationen im Programm zu erlauben, ohne dass der Compiler meckert. Dies ist wichtig, wenn von der Klasse `NummeriertesObjekt` geerbt wird, weil bei der Zuweisung eines Objekts der abgeleiteten Klasse die Zuweisungsoperatoren der Elemente der Klasse inklusive der anonymen Subobjekte aufgerufen werden.

```
NummeriertesObjekt& operator=(const NummeriertesObjekt&) {
 return *this;
}
```

#### 9.6 Deklaration in *meinstring.h*

```
// als Elementfunktion
MeinString& operator=(const MeinString&); // Zuweisung
MeinString& operator=(const char *); // Zuweisung

// Indexoperator:
const char& operator[](std::size_t position) const;
// Indexoperator. Die Referenz erlaubt Ändern des Zeichens.
char& operator[](std::size_t position);
```

```
// global:
std::ostream& operator<<(std::ostream&, const MeinString&);
```

Implementierung in *meinstring.cpp*:

```
#include "meinstring.h"
#include <stdexcept>
#include <cstring>

namespace {
 void bereichPruefen(bool bedingung) {
 if(!bedingung) {
 throw std::out_of_range("MeinString: Bereichsüberschreitung");
 }
 }
}

MeinString& MeinString::operator=(const MeinString& m) {
 reserve_only(m.len);
 strcpy(start, m.start);
 len = m.len;
 return *this;
}

MeinString& MeinString::operator=(const char *s) {
 size_t temp = strlen(s);
 reserve_only(temp);
 strcpy(start, s);
 len = temp;
 return *this;
}

char& MeinString::operator[](size_t pos) { // Zeichen per Referenz holen
 bereichPruefen(pos >= 0 && pos<= len); // Nullbyte lesen ist erlaubt
 return start[pos];
}

const char& MeinString::operator[](size_t pos) const { // Zeichen holen
 bereichPruefen(pos >= 0 && pos<= len); // Nullbyte lesen ist erlaubt
 return start[pos];
}
```

In Analogie zum C++-Standard-Entwurf ist das Lesen des Nullbytes erlaubt, anders als bei der Funktion `at()`. Weil für nichtkonstante `MeinString`-Objekte die nichtkonstante Variante von `operator[]()` genommen wird, ergibt sich aus der Tatsache, dass auch eine Referenz auf das Nullbyte zurückgegeben werden kann, ein Schönheitsfehler: `operator[]()` erlaubt das Beschreiben des Nullbytes. `at()` kann deshalb nicht ohne Weiteres durch `char& operator[](int)` ersetzt werden, wenn nicht schreibend auf das Nullbyte zugegriffen werden darf. `operator[]()` entsprechend auch für diese Fälle abzusichern, ist mit wenig Aufwand nur möglich, wenn jede andere Methode das Nullbyte auf Veränderung prüft, was etwas Laufzeit kostet.

```
// Ausgabeoperator (globale Funktion)
std::ostream& operator<<(std::ostream& os, const MeinString& m) {
 os << m.c_str();
 return os;
}
```

### 9.7 Deklaration in *meinstring.h*

```
// als Elementfunktion
MeinString& operator+=(const MeinString&); // Verkettung
// global
MeinString operator+(MeinString, const MeinString&);
```

Implementierung in *meinstring.cpp*:

```
MeinString& MeinString::operator+=(const MeinString& m) { // Verkettung
 char *p = new char[len + m.len + 1]; // neuen Platz beschaffen
 strcpy(p, start); // Teil 1 kopieren
 strcpy(p + len, m.start); // Teil 2 kopieren
 delete [] start; // alten Platz freigeben
 len += m.len; // Verwaltungsinformation aktualisieren
 start = p;
 return *this;
}

// Verkettung
MeinString operator+(MeinString a, const MeinString& b) {
 return a += b;
}
```

Eine ausführliche Diskussion des Plus-Operators und seiner Optimierungsmöglichkeiten gibt es in Abschnitt 22.1.

- 9.8 Ein Rückgabetypp `Datum&` erspart den impliziten Aufruf des Kopierkonstruktors.
- 9.9 *Nein!* Die lokale Variable `temp` ist nach Verlassen der Operatorfunktion nicht mehr existent. Wenn weitere Erläuterungen nötig sein sollten: Schlagen Sie sie auf Seite 112 nach. In der vorhergehenden Aufgabe wird ein schon *vor* dem Eintritt in die Operatorfunktion existierendes Objekt zurückgegeben.
- 9.10 Die folgenden Operatoren sind in *datum.h* zu deklarieren. Die Implementierung gehört nach *datum.cpp*. `#include<iostream>` nicht vergessen!

```
std::ostream& operator<<(std::ostream& os, const Datum& d) {
 os << d.tag() << '.' << d.monat() << '.' << d.jahr();
 return os;
}
```

- 9.11
- ```
bool operator==(const Datum& a, const Datum& b) {
    return a.tag() == b.tag()
        && a.monat() == b.monat()
        && a.jahr() == b.jahr();
}
```

```

bool operator!=(const Datum& a, const Datum& b) {
    return !(a == b);
}
bool operator<(const Datum& a, const Datum& b) {
    return  a.jahr() < b.jahr()
        || a.jahr() == b.jahr() && a.monat() < b.monat()
        || a.jahr() == b.jahr()
            && a.monat() == b.monat() && a.tag() < b.tag();
}

```

9.12

```

int datumDifferenz(const Datum& a, const Datum& b) {
    if(a == b) {          // kurzer Prozess bei Gleichheit
        return 0;
    }
    bool richtigeReihenfolge = a < b;
    Datum frueher = a;
    Datum spaeter = b;
    if(!richtigeReihenfolge) { // ggf. vertauschen
        frueher = b;
        spaeter = a;
    }
    int Differenz = 0;
    while(frueher != spaeter) { // nicht optimiert (tagweises Hochzählen)
        ++Differenz;
        ++frueher;
    }
    return richtigeReihenfolge ? Differenz : -Differenz;
}

```

9.13 Das Ergebnis ist der 19.1.2038 (2147483647 Sekunden seit dem 1.1.1970), falls `time_t` einem 32-Bit-`int` entspricht. Dies ist auf vielen Unix-Systemen der Fall.

9.14 • *ungueltigesdatumexception.h*:

```

#ifndef UNGUELTIGESDATUMEXCEPTION_H
#define UNGUELTIGESDATUMEXCEPTION_H
#include<stdexcept>
#include<string>

class UngueltigesDatumException : public std::runtime_error {
public:
    UngueltigesDatumException(int t, int m, int j)
        : std::runtime_error(toString(t, m, j)) {
    }
private:
    static std::string toString(int tag, int monat, int jahr) {
        std::string t = std::to_string(tag);
        std::string m = std::to_string(monat);
        std::string j = std::to_string(jahr);
        return t + "." + m + "." + j + " ist kein gültiges Datum.";
    }
};
#endif

```

Die Deklaration der Methode `set()` in *datum.h*:

```
void set(int t, int m, int j);
```

Die Methode `set()` in *datum.cpp* lautet:

```
void Datum::set(int t, int m, int j) {
    if(!istGueltigesDatum(t, m, j)) {
        throw UngueltigesDatumException(t, m, j);
    }
    tag_ = t;
    monat_ = m;
    jahr_ = j;
}
```

```
9.15 std::string Datum::toString() const {
    std::string temp("tt.mm.jjjj");
        // implizite Umwandlung in char
    temp[0] = tag_/10 + '0';
    temp[1] = tag_%10 + '0';
    temp[3] = monat_/10 + '0';
    temp[4] = monat_%10 + '0';
    int pos = 9;           // letzte Jahresziffer
    int j = jahr_;
    while(j > 0) {
        temp[pos] = j % 10 + '0'; // letzte Ziffer
        j = j/10;                // letzte Ziffer abtrennen
        --pos;
    }
    return temp;
}
```

```
9.16 template<typename T>
void Matrix<T>::swap(Matrix<T>& rhs) { // Verwendung in op*= unten
    super::swap(rhs);
    std::swap(yDim, rhs.yDim);
}

template<typename T>
Matrix<T>& Matrix<T>::operator*=(const Matrix<T>& b) {
    if(spalten() != b.zeilen())
        throw "Falsche Dimension in Matrix*=";
    Matrix<T> erg(zeilen(), b.spalten());
    for(size_t i = 0; i < zeilen(); ++i) {
        for(size_t j=0; j < b.spalten(); ++j) {
            erg[i][j]= T(0);
            for(size_t k=0; k < spalten(); ++k) {
                erg[i][j] += super::operator[](i)[k] * b[k][j];
            }
        }
    }
    swap(erg);           // *this mit erg vertauschen
    return *this;
}
```

- 9.17 Innerhalb des Operators wird der vorhandene Kurzform-Operator für die Multiplikation aufgerufen.

```
template<typename T>
Matrix<T> operator*(Matrix<T> a, const Matrix<T>& b) {
    return a *= b;    // Matrix<T>::operator*=(const Matrix<T>& b)
}
```

Manche mögen wegen des Aufwandes den Aufruf des Kopierkonstruktors bei der Rückgabe von a bemängeln. Andererseits ist bei genauer Betrachtung der Aufwand gegenüber dem Gesamtaufwand der Multiplikation für sehr große Matrizen tatsächlich vernachlässigbar: Falls wir der Einfachheit halber große quadratische Matrizen mit n Zeilen und n Spalten betrachten, ist der Aufwand für den Kopierkonstruktor $\propto n^2$, der Aufwand zur Multiplikation jedoch $\propto n^3$. Der tatsächliche Aufwand wird jedoch geringer sein, weil der Compiler den Aufruf des Kopierkonstruktors bei der Rückgabe temporärer Objekte wegoptimieren kann. Alternativ kann man den Konstruktoraufruf durch die in Abschnitt 22.1 ff. gezeigten Techniken selbst wegoptimieren – oder besser: Man setzt gleich eine der fertigen Bibliotheken ein.

- 9.18 Die hier vorgestellte Lösung benutzt keine Schleife, die Frage kann also mit »Ja« beantwortet werden. Im Lösungsvorschlag werden die Matrixelemente, die ja vom Typ `mathVektor<T>` sind, mit `v`, einem `mathVektor<T>` initialisiert. `v` wiederum wird beim Aufruf `v.init(Wert)`; durch `Vektor<T>::init (const T&)` initialisiert.

```
template<typename T>
void Matrix<T>::init(const T& Wert) {
    mathVektor<T> v(Spalten()); // Hilfsvektor v definieren und initialisieren
    v.init(Wert);
    // Die Matrix ist ein Vektor (von Vektoren), dessen Elemente nun initialisiert werden.
    Vektor<mathVektor<T> >::init(v);
}
```

Eine konzeptionell einfachere und vermutlich verständlichere Lösung wäre eine geschachtelte Schleife über alle Elemente. Ein Laufzeitnachteil ergibt sich nicht, weil die Schleife in der vorgestellten Lösung ebenfalls vorhanden ist, wenn auch versteckt. Überdies erspart sie die Erzeugung des temporären Vektors `v`.

Kapitel 11

- 11.1 `Liste(const Liste& liste)` // Kopierkonstruktor
`: anfang(0), anzahl(0) {`
 `if(liste.size() > 0) {`
 `iterator I = liste.begin();`
 `push_front(*I++); // erstes Element anlegen`
 `ListElement *letztes = anfang;`
 `while(I != liste.end()) {`
 `// Elemente am Ende einfügen, damit die`
 `// Reihenfolge erhalten bleibt`
 `letztes->naechstes = new ListElement(*I++, 0);`
 `letztes = letztes->naechstes;`
 `++anzahl;`
 `}`
 `}`

```

    }
}

Liste& operator=(Liste temp) { // Zuweisungsoperator
    std::swap(temp.anfang, anfang);
    std::swap(temp.anzahl, anzahl);
    return *this;
}

~Liste() { // Destruktor
    clear();
}

iterator erase(iterator p) {
    if(empty()) {
        return iterator(); // leere Liste
    }
    ListElement* zuLoeschen = p.aktuellesElement;
    // Vorgänger suchen
    ListElement* vorgaenger = anfang;
    if(zuLoeschen != anfang) {
        while( vorgaenger->naechstes != zuLoeschen) {
            vorgaenger = vorgaenger->naechstes;
        }
        // Zeiger verbiegen
        vorgaenger->naechstes = zuLoeschen->naechstes;
    }
    else { // am Anfang löschen
        anfang = zuLoeschen->naechstes; // Zeiger verbiegen
    }
    delete zuLoeschen;
    --anzahl;
    return ++p; // Nachfolger zurückgeben
}

void pop_front() {
    erase(begin());
}

bool empty() const {
    return anfang == 0;
}

size_t size() const {
    return anzahl;
}

void clear() {
    while(!empty()) {
        pop_front();
    }
}

```

Kapitel 13

```

13.1 #ifndef ABLAGE_H
#define ABLAGE_H
#include <boost/thread.hpp>

namespace {
    boost::mutex ausgabeMutex;
}

class Ablage {
public:
    Ablage(int platz)
        : kapazitaet(platz), inhalt(new int[platz]),
          anzahl(0), lesePos(-1), schreibPos(0) {
    }
    ~Ablage() {
        delete [] inhalt;
    }
    int get() {
        boost::unique_lock<boost::mutex> lock(objektMutex);
        while(anzahl == 0) { // leer
            cond.wait(lock);
        }
        --anzahl;
        cond.notify_all();
        lesePos = (lesePos + 1) % kapazitaet;
        return inhalt[lesePos];
    }
    void put(int wert) {
        boost::unique_lock<boost::mutex> lock(objektMutex);
        while(anzahl == kapazitaet) { // voll
            cond.wait(lock);
        }
        inhalt[schreibPos] = wert;
        ++anzahl;
        schreibPos = (schreibPos + 1) % kapazitaet;
        cond.notify_all();
    }
private:
    int kapazitaet;
    int* const inhalt;
    int anzahl;
    // Aufbau als Ringpuffer (FIFO)
    int lesePos; // letzte gelesene Position
    int schreibPos; // nächste zu schreibende Position
    boost::mutex objektMutex;
    boost::condition_variable cond;
    // wegen Zeigerattribut inhalt:
    Ablage(const Ablage&); // Kopie verbieten
    Ablage& operator=(const Ablage&); // Zuweisung verbieten
};
#endif

```


Kapitel 24

24.1 Aus Platzgründen wird die Lösung nicht abgedruckt. Sie ist vollständig in den Beispielen enthalten (siehe Verzeichnis *cppbuch/loesungen/k24/1*).

24.2 • *heap.t*

```
#ifndef HEAP_T
#define HEAP_T
#include<algorithm>
#include<vector>
#include<utility>
using std::vector;

template<class T, class Compare = std::less<T> >
class Heap {
public:
    Heap(const Compare& cmp = Compare())
        : anz(0), comp(cmp), v(vector<T>(1)), last(v.begin()) {
    }

    void push(const T& t) {
        if(anz == v.size()) {
            v.resize(anz+100);
            last = v.begin() + anz; // neu bestimmen
        }
        *last = t;
        push_heap(v.begin(), ++last, comp);
        ++anz;
    }

    void pop() {
        pop_heap(v.begin(), last--, comp);
        --anz;
    }

    const T& top() const { return *v.begin(); }

    bool empty() const { return anz == 0; }

    size_t size() const { return anz; }

    vector<T> toSortedVector() const {
        vector<T> temp(anz);
        for(size_t i = 0; i < anz; ++i) {
            temp[i] = v[i];
        }
        sort_heap(temp.begin(), temp.end(), comp);
        return temp;
    }
private:
    size_t anz;
    Compare comp;
    vector<T> v;
}
```

```

    typename vector<T>::iterator last;
};
#endif

```

• Anwendungsbeispiel *main.cpp*

```

#include<iostream>
#include"heap.t"
using namespace std;

int main() {
    Heap<pair<int, string> > promis;
    promis.push(make_pair(7, "Jack Nicholson"));
    promis.push(make_pair(10, "Bill Clinton"));
    promis.push(make_pair(7, "Thomas Gottschalk"));
    promis.push(make_pair(8, "Brad Pitt"));
    promis.push(make_pair(8, "Peter Jackson"));
    promis.push(pair<int, string>(10, "Tina Turner")); // mal ohne make_pair

    cout << "Sortiert:" << endl;
    vector<pair<int, string> > vs = promis.toSortedVector();
    for(size_t i=0; i < vs.size(); ++i) {
        cout << vs[i].second << ", Priorität "
            << vs[i].first << endl;
    }
    cout << "Leeren:" << endl;
    while(!promis.empty()) {
        cout << promis.top().second << ", Rang "
            << promis.top().first
            << " size=" << promis.size()
            << endl;
        promis.pop();
    }
}

```

24.3 #include<iostream>
 #include<cmath>
 #include<complex>
 using namespace std;

```

int main() {
    cout << "Quadratische Gleichung  $x^2+x+p*x+q = 0$ \n";
    cout << "Koeffizienten p, q eingeben:";
    double p, q;
    cin >> p >> q;
    double Diskriminante = p*p/4.0 - q;

    cout << "Lösung :\n";
    if (Diskriminante >= 0.0) {
        double x1 = -p/2.0 + sqrt(Diskriminante);
        double x2 = -p/2.0 - sqrt(Diskriminante);
        cout << "x1= " << x1 << " x2= " << x2 << endl;
    }
}

```

```

    else {
        complex<double> ergebnis(-p/2, sqrt(-Diskriminante));
        cout << "x1 = " << ergebnis << endl;
        cout << "x2 = " << conj(ergebnis) << endl;
    }
}

```

Kapitel 28

```

28.1 #include<iostream>
#include<stack>
using namespace std;

void bewegen(int n, int a, int b, int c) {
    stack<int> s;
    int t;          // zum Vertauschen der Werte
    // ersten Aufruf transformieren
    while (n > 0) {
        // aktuelle Daten sichern
        s.push(n); s.push(a); s.push(b); s.push(c);
        // Aufruf mit neuen Daten simulieren
        --n; t = b; b = c; c = t;
    }
    // Haupt-Schleife
    while (!s.empty()) {
        c = s.top(); s.pop(); // Daten wiederherstellen
        b = s.top(); s.pop();
        a = s.top(); s.pop();
        n = s.top(); s.pop();
        cout << "Bringe eine Scheibe von " << a
              << " nach " << b << endl;
        --n; t = a; a = c; c = t;
        while (n > 0) {
            // aktuelle Daten sichern
            s.push(n); s.push(a); s.push(b); s.push(c);
            // Aufruf mit neuen Daten simulieren
            --n; t = b; b = c; c = t;
        }
    }
}

int main() {
    cout << "Türme von Hanoi! Anzahl der Scheiben: ";
    int scheiben;
    cin >> scheiben;
    bewegen(scheiben, 1, 2, 3);
}

```

```

28.2 #include<iostream>
#include<utility>
#include<queue>

```

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

int main() {
    priority_queue<pair<int, string> > promis;
    promis.push(make_pair(7, "Jack Nicholson"));
    promis.push(make_pair(10, "Bill Clinton"));
    promis.push(make_pair(7, "Thomas Gottschalk"));
    promis.push(make_pair(8, "Brad Pitt"));
    promis.push(make_pair(8, "Peter Jackson"));
    promis.push(pair<int, string>(10, "Tina Turner"));
    while(!promis.empty()) {
        cout << promis.top().second << ", Priorität "
              << promis.top().first << endl;
        promis.pop();
    }
}
```

```
28.3 priority_queue<pair<int, string>,
        deque<pair<int, string> >,
        greater<pair<int, string> > > promis;
```

```
28.4 #include<iostream>
#include<utility>
#include<map>
#include<string>
using namespace std;

int main() {
    multimap<int, string, greater<int> > promis;
    // multimap<int, string> promis; // umgekehrte Sortierung
    promis.insert(make_pair(7, "Jack Nicholson"));
    // ... usw. wie in Lösung 28.2
    for(multimap<int, string>::iterator iter = promis.begin();
        iter != promis.end(); ++iter) {
        cout << (*iter).second << ", Priorität "
              << (*iter).first << endl;
    }
}
```

Kapitel 30

30.1 `count()` würde nicht funktionieren, weil es ein `pair`-Objekt als Parameter verlangt, es aber in der Aufgabe nur um den Rang, also nur einen Teil der Paar-Kombination geht. Mit `count_if()`, das ein Prädikat verlangt (vgl. Seite 661), ist das Problem zu lösen, weil das Prädikat beliebig gestaltet werden kann. Das Prädikat ist ein Funktionsobjekt und vergleicht nur, ob der Rang der gewünschte ist – der Name wird ignoriert:

```
// gleicherrang.h
#ifndef GLEICHERRANG_H
#define GLEICHERRANG_H
#include<utility>
#include<string>

class GleicherRang {
public:
    GleicherRang(int r) : rang(r) { }
    bool operator()(const std::pair<int, std::string>& p) const {
        return p.first == rang;
    }
private:
    int rang;
};
#endif
```

Abgesehen von #include "gleicherrang.h" wird das Programm der Lösung 28.4 nur noch um folgendes Stück erweitert:

```
int gesucht = 8;
cout << "Es gibt "
    << count_if(promis.begin(), promis.end(),
                GleicherRang(gesucht))
    << " Einträge mit Rang " << gesucht << endl;
```

- 30.2 equal_range() arbeitet nur auf sortierten Containern und braucht daher die Information, welches von zwei Elementen das größere ist. In diesem speziellen Fall wird dabei nur der Rang verglichen. Aus denselben Gründen wie in der vorhergehenden Lösung benötigt equal_range() ein Funktionsobjekt, das den Vergleich erledigt:

- rangvergleich.h

```
#ifndef RANGVERGLEICH_H
#define RANGVERGLEICH_H
#include<utility>
#include<string>

class Rangvergleich {
public:
    bool operator()(const std::pair<int, std::string>& p1,
                    const std::pair<int, std::string>& p2) const {
        return p1.first > p2.first;
    }
};
#endif
```

Abgesehen vom Inkludieren der Header-Datei wird das Programm der Lösung 28.4 nur noch um folgendes Stück erweitert:

```
// nur der Rang interessiert, siehe rangvergleich.h
pair<int, string> gesuchtesPaar(8, "Dummy");
cout << "Es gibt folgende Einträge mit Rang "
    << gesuchtesPaar.first << ":" << endl;
```

```

pair<multimap<int, string, greater<int> >::iterator,
    multimap<int, string, greater<int> >::iterator>
bereich = equal_range(promis.begin(), promis.end(),
    gesuchtesPaar, Rangvergleich());

for(multimap<int, string>::iterator iter = bereich.first;
    iter != bereich.second; ++iter) {
    cout << (*iter).second << endl;
}

```

A.7 Installation der DVD-Software für Windows

A.7.1 Installation des Compilers und der Entwicklungsumgebung

Die einfachste Möglichkeit ist die Installation von der DVD, auf der die verwendete Software in komprimierter Form vorliegt. Loggen Sie sich zur Installation als Administrator ein. Anschließend klicken Sie die Datei *installcomp.exe* von der DVD an. Danach melden Sie sich ab und wieder an, damit die Pfadeinstellungen wirksam werden. Bei der Installation werden die folgenden Verzeichnisse angelegt bzw. überschrieben:

- **C:\MinGW**: Dieses Verzeichnis enthält den GNU C++-Compiler 4.5.2 und zugehörige Programme sowie das Datenbankprogramm SQLite. Auch sind einige Dienstprogramme dabei. Platzbedarf etwa 320 MB.
- **C:\CodeBlocks**: Verzeichnis mit der Entwicklungsumgebung. Platzbedarf etwa 44 MB. Es wird auch eine Verknüpfung auf dem Desktop angelegt.
- **C:\cppbuchiincludes**: Hier sind einige für das Compilieren der Beispiele notwendige Dateien abgelegt.

De-Installation

Die Software wird vom Rechner entfernt, indem die genannten Verzeichnisse gelöscht werden. Die Desktop-Verknüpfung muss manuell gelöscht werden, ebenso die Einträge

C:\MinGW\msys\1.0\bin; C:\MinGW\bin; C:\MinGW\lib; C:\Codeblocks

in der PATH-Umgebungsvariablen (nicht zwingend).

A.7.2 Installation der Boost-Bibliothek

Die Installation wird erst ab Kapitel 12 gebraucht, ist also für den Einstieg in die C++-Programmierung nicht notwendig. Die oben beschriebene Installation des Compilers muss abgeschlossen sein. Loggen Sie sich zur als Administrator ein. Anschließend klicken Sie die Datei *installboost.exe* von der DVD an. Danach melden Sie sich ab und wieder an, damit die Pfadeinstellungen wirksam werden. Bei der Installation wird das Verzeichnis

`C:\Boost` angelegt bzw. überschrieben. Platzbedarf etwa 1,2 GB. Während der Installation kann der Platzbedarf wegen der temporären Dateien kurzfristig noch größer werden. Boost kann auch auf einem anderen Laufwerk wie `D:\` oder `E:\` installiert werden. In diesem Fall muss die Datei `C:\cppbuchincludes\make\include.mak` entsprechend angepasst werden. Wegen der Größe der Bibliothek dauert die Installation einige Minuten.

De-Installation

Die Software wird vom Rechner entfernt, indem das Installationsverzeichnis `C:\Boost` gelöscht wird. Der Eintrag `C:\Boost\stage\lib` in der PATH-Umgebungsvariablen muss manuell gelöscht werden (nicht zwingend).

A.7.3 Installation von Qt

Die Installation wird erst ab Kapitel 14 gebraucht, ist also für den Einstieg in die C++-Programmierung nicht notwendig.

1. Führen Sie die Datei `qt-win-opensource-4.7.2-mingw.exe` (Verzeichnis `win/qt`) aus und folgen Sie den Anweisungen. Die Fragen sollten Sie mit ja (Fehlermeldung wegen MinGW ignorieren)
o (open source) und
y (Lizenz akzeptieren) beantworten.
2. Ergänzen Sie die PATH-Umgebungsvariable um das Verzeichnis `C:\Qt\4.7.2\bin`. Der Eintrag wird nach Abmelden und Wiederanmelden wirksam.

Als Alternative bietet sich die vollständige Entwicklungsumgebung Qt SDK an, die Sie von <http://qt.nokia.com/> herunterladen können.

De-Installation

Die Software wird mit den Windows-Betriebsmitteln vom Rechner entfernt, das heißt, Programmdeinstallation über die Systemsteuerung.

A.7.4 Codeblocks einrichten

Klicken Sie das Code::Blocks-Symbol auf dem Desktop an. Zuerst wird der Compiler abgefragt. Einfach GNU GCC anklicken und mit OK bestätigen. Die aufpoppenden Tipps- und Skript-Fenster schließen. Für manche Programme werden die Boost-Library und die Dateien im Verzeichnis `C:\cppbuchinclude` benötigt. Deswegen wird Code::Blocks dafür eingerichtet. Ich gehe davon aus, dass Sie Boost durch Entpacken der Datei `boost.tgz` installiert haben. In der Menüleiste »Settings« klicken und »Compiler und debugger« wählen. Unter dem oberen Reiter »Compiler settings« gibt es ein wenig darunter den Reiter »Compiler Flags«. Dort anklicken:

- Produce debugging symbols [-g]
- Enable all compiler warnings [-Wall]
- Have g++ follow the coming C++0x ISO C++ language standard [-std=c++0x]

Dann den Reiter »Search directories« anklicken und darunter den Reiter »Compiler« wählen. Unter der Fläche »Add« (bzw. »Hinzufügen«) anklicken und das Verzeichnis

`C:\cppbuchincludes/include`

eintragen – oder mit dem Suchbutton rechts vom Eingabefeld ermitteln. /home/user ist ein Platzhalter, bitte für Ihr System anpassen. Als Nächstes auf dieselbe Art

```
C:/Boost
```

eintragen. Im nächsten Schritt werden dem Linker (Reiter »Linker settings«) die Bibliotheken mitgeteilt. Dazu im Feld »Link Libraries« mit dem »Add«- oder »Hinzufügen«-Button die folgenden Dateien eintragen:

```
C:/Boost/stage/lib/libboost_regex-mgw45-mt-1_45.dll.a  
C:/Boost/stage/lib/libboost_filesystem-mgw45-mt-1_45.dll.a  
C:/Boost/stage/lib/libboost_system-mgw45-mt-1_45.dll.a  
C:/Boost/stage/lib/libboost_thread-mgw45-mt-1_45.dll.a  
C:/Boost/stage/lib/libboost_unit_test_framework-mgw45-mt-1_45.dll.a
```

Es gibt viel mehr Boost-Libraries, aber nur die angegebenen werden von einigen der Beispiele benötigt. Jetzt unten mit OK bestätigen. Damit sind Sie für die weiteren Beispiele gerüstet, wenn es sich nur um einzelne Dateien handelt.

Das erste Projekt

Das Starten einer ersten einfachen Programmdatei wird auf Seite 38 beschrieben. Es gibt aber auch Programme, die aus mehreren Dateien bestehen. Für diese Dateien muss ein sogenanntes Projekt angelegt werden. Dazu klicken Sie im »Start here«-Fenster von Code::Blocks auf »Create a new Project«. Im erscheinenden Fenster gibt es eine große Auswahl verschiedener Projekttypen. Für die Beispiele dieses Buchs genügen »Console application« und »Qt4 project«. Bitte wählen Sie »Console application« und dann »Next« und »C++« im erscheinenden Fenster. Als erstes Beispiel wird das Projekt im Verzeichnis *cppbuch/k4/ratio* gewählt. Um das Programm zu erzeugen, könnten Sie direkt in das Verzeichnis gehen und `make` eingeben, aber hier geht es um die Anlage des Projekts in Code::Blocks. Geben Sie als Titel »ratio« an.

In der nächsten Zeile suchen Sie das Verzeichnis *cppbuch/k4* und tragen es ein. Mit »Next« und »Finish« bestätigen. Normalerweise wird automatisch eine Datei *main.cpp* angelegt, aber hier soll die vorhandene genutzt werden. Deswegen die Warnung mit »Nein« beantworten. Links im Fenster können Sie den Bereich »Sources« expandieren und sehen dann *main.cpp*. Ein Doppelklick holt die Datei in den Editor. Das Projekt ist aber noch nicht vollständig; deswegen wird mit Rechtsklick auf den Projekt-Namen »ratio« und »add files...« die Datei *rational.cpp* ausgewählt. Bestätigen Sie mit OK, und Sie sehen auch diese Datei links im Bereich. Mit der Taste F9 können Sie alle Dateien übersetzen und ausführen. Wegen einer beabsichtigten Division durch 0 gibt es einen Abbruch, wie Sie sehen. Wenn Sie in *main.cpp* die letzten Zeilen löschen, beendet sich das Programm regulär.

Letzlich können Sie aber auch mit einem beliebigen ASCII-Editor (Wordpad) die Dateien bearbeiten und mit `make` die Übersetzung anstoßen. Fehlermeldungen werden dann auf der Konsole statt in der IDE angezeigt.



Weiterführende Informationen

Weitere Informationen zur Bedienung von Code::Blocks bitte ich, dem Manual (Verzeichnis *win\codeblocks* der DVD) und der Internetseite <http://www.codeblocks.de> zu entnehmen. Dort sind besonders die Rubriken Forum und Wiki interessant.



A.7.5 Integration von Qt in ein Code::Blocks-Projekt

Die Integration von Qt in ein Code::Blocks-Projekt wird an einem schon vorhandenen Beispiel gezeigt, damit Sie nicht so viel Tipparbeit haben. Die Hinweise sind leicht auf ein neu anzulegendes Projekts übertragbar. Nach dem Start von Code::Blocks »Create a new project« anwählen, im erscheinenden Fenster weiter unten »QT4 project« anklicken. Dann geben Sie den Projekt-Namen »label« (keinen anderen, weil existierende Dateien verwendet werden) und das Verzeichnis *...cppbuch\k14* an, in dem das Projekt gespeichert werden soll. Die Punkte sind durch den Rest des vollständigen Pfadnamens zu ersetzen. Mit »Next« kommen Sie zum nächsten Fenster, in dem Sie angeben, wo Qt installiert ist. Wenn Qt installiert und im Pfad ist, können Sie das Feld `$(#qt4)` so belassen.

Falls Sie Qt nicht von der DVD, sondern manuell installiert haben sollten, kann es sein, dass CodeBlocks sich beschwert, weil es *QtCore4.lib* nicht findet. Dann machen Sie einfach eine Kopie von *C:\Qt\4.7.2\lib\QtCore4.dll*, benennen sie in *QtCore4.lib* um, und bringen sie nach *C:\Qt\4.7.2\lib*.

Mit »Next« beenden Sie die Eingaben. Die nächste Frage beantworten Sie bitte mit *Nein*, damit die vorhandene Datei *main.cpp* nicht überschrieben wird! Nehmen Sie nun die folgenden Einstellungen vor:

- Unter »Project« → »Properties« den Reiter »Project settings« wählen und das Kästchen neben dem Text »This is a custom Makefile« aktivieren. Damit wird zur Compilation das von *qmake* erzeugte Makefile ausgeführt.
- Danach den Reiter »Build target« anklicken und mit »Rename« das Target »Debug« in »debug« umbenennen. Das von *qmake* erzeugte Makefile enthält »debug« als Target.
- Im selben Fenster bei »Output filename« bitte *debug\label.exe* eintragen (also *nicht: bin\Debug\label.exe!*). Mit OK bestätigen.
- Unter »Project« → »Build options« ganz links »debug« anklicken und bei »Pre/post build steps« in das obere Feld den Text

```
qmake -project
qmake
```

eintragen. Die erste Anweisung erzeugt eine Steuerungsdatei für das Projekt, die zweite ein Makefile. Mit OK bestätigen.

Wenn Sie links »Sources« expandieren und auf *main.cpp* klicken, wird die Datei im Editor angezeigt. Übersetzung und Ausführung des Programms können nun wie üblich über die Menüleiste (Build) oder die Taste F9 gestartet werden. Diese Einstellungen müssen für jedes Qt-Projekt vorgenommen werden!

A.7.6 Bei Verzicht auf die automatische Installation

Die folgenden Anweisungen gelten nur für den Fall, dass Sie fertigen Installationsdateien nicht benutzen wollen oder wissen möchten, wie die einzelnen Installationsschritte aussehen. Damit die Programme erreichbar sind, müssen sie im Pfad sein – dies sollten Sie als Nächstes erledigen.

Pfad einstellen: Windows 7

1. Als Administrator einloggen. Dann »Start« → »Systemsteuerung« → »System und Sicherheit« → »System« und dort den links »Erweiterte Systemeinstellungen« anklicken. Auf der erscheinenden Registerkarte unten »Umgebungsvariablen« anklicken. Dann im Fenster unten bei den Systemvariablen die Variable »Path« wählen und dann »Bearbeiten« anklicken. Den Pfad um die benötigten Verzeichnisse ergänzen. Er sollte am *Anfang* (!) enthalten:

```
C:\MinGW\msys\1.0\bin; C:\MinGW\bin; C:\MinGW\Lib; C:\Codeblocks
```

Nach Installation von Boost und Qt kommen hinzu

```
; C:\Boost\stage\lib; C:\Qt\4.7.2\bin
```

Bearbeiten Sie die Pfadangaben sehr sorgfältig und löschen Sie nicht schon vorhandene Einträge wie etwa `%SystemRoot%\system32` usw.

Alternativ kann auch eine in Prozentzeichen eingeschlossene Umgebungsvariable angegeben werden, sofern sie definiert ist, zum Beispiel `%MINGW_HOME%\bin`.

Danach mit zweimal »OK« beenden.

2. Damit die Änderung wirksam wird, jetzt abmelden und erneut als Administrator anmelden.

Pfad einstellen: Windows XP

Klicken Sie »Start« → »Systemsteuerung« → »System« und dort den Reiter »Erweitert« an. Auf der erscheinenden Registerkarte unten »Umgebungsvariablen« anklicken. Im Fenster unten bei den Systemvariablen die Variable »Path« und dann »Bearbeiten« anklicken. Dann weiter wie oben bei Windows 7 beschrieben.

Pfad einstellen: Windows Vista

Als Administrator einloggen. Dann »Start« → »Systemsteuerung« → »System« → »Erweiterte Systemeinstellungen« und dort den Reiter »Erweitert« anklicken. Dann weiter wie oben bei Windows 7 beschrieben.

Installation des Compilers

Entpacken Sie die Datei `win/mingw/mingw.tgz` von der DVD mit einem geeigneten Programm nach C:. Passen Sie den Pfad an wie oben beschrieben.

Installation der IDE Code::Blocks

Klicken Sie die Datei `codeblocks-10.05-setup.exe` im Verzeichnis `win/codeblocks/` der DVD an und folgen Sie den Anweisungen.

Installation der Boost-Library

Gehen Sie in das Verzeichnis *win\boost* der DVD. Folgen Sie den Anweisungen in der Datei *INSTALL.txt*.

Beispieldateien entpacken

Um mit den Beispieldateien zu arbeiten, kopieren Sie die Datei *cppbuch.tgz* aus dem Verzeichnis *win* der DVD in das Verzeichnis *Eigene Dateien* (Windows XP) oder *Dokumente* (Windows 7 oder Windows Vista). Öffnen Sie ein Shell-Fenster und gehen Sie mit *cd*-Befehlen in dieses Verzeichnis. Die Datei wird mit

```
tar xvfz cppbuch.tgz
```

entpackt. Wenn Sie danach zum Beispiel in das Verzeichnis *cppbuch/k1* gehen und das Kommando *make* eintippen, werden alle Beispiele in diesem Verzeichnis übersetzt. Die entstehenden Programme, erkennbar an der Endung *.exe*, werden durch Eingabe des Namens in das Shell-Fenster aufgerufen.

A.8 Installation der DVD-Software für Linux

A.8.1 Installation des Compilers

Wenn Sie *g++ --version* in ein Shell-Fenster eintippen und Sie die Meldung »command not found« bekommen, ist der Compiler nicht installiert. Andernfalls sollte er sich mit einer Versionsnummer 4.4.x oder höher melden. Installieren Sie sich den C++-Compiler gegebenenfalls von der Betriebssystem-DVD oder mit dem Software-Update-Tool des Betriebssystems, falls er nicht vorhanden ist. Für einige wenige Beispiele wird jedoch mindestens die Version 4.5 des GNU C++-Compilers benötigt. Falls bereits die Version 4.5 oder neuer auf Ihrem System vorhanden ist, können Sie den Rest des Abschnitts überspringen und direkt zum Abschnitt *Installation von Code::Blocks für Linux* (A.8.3) gehen!

Um einen Konflikt mit dem C++-Compiler des Betriebssystems zu vermeiden, wird er separat installiert.



Hinweis

Wenn Ihnen die folgenden Schritte (noch) zu kompliziert erscheinen, lesen Sie einfach unten beim Abschnitt *Installation von Code::Blocks für Linux* (A.8.3) weiter. Sie verzichten damit nur auf den Test einiger Beispiele, die von neueren C++-Techniken Gebrauch machen. Sie können die Installation jedoch zu einem späteren Zeitpunkt nachholen.

Vergewissern Sie sich bitte, ob auf Ihrem System die Pakete GMP Version 4.2 oder höher (<http://gmplib.org/>) und MPFR Version 2.3 oder höher (<http://www.mpfr.org/>) installiert sind. Die Pakete *gmp-devel* und *mpfr-devel* müssen auch vorliegen. Wenn nicht, instal-

lieren Sie diese Pakete mit Ihrem System-Werkzeug zur Softwareinstallation. Falls Ihnen diese Pakete nicht vorliegen, kopieren Sie sich die entsprechenden Dateien aus dem Verzeichnis *linux/gcc* der DVD in das Verzeichnis */usr/local*. Die Pakete werden zunächst entpackt:

Endung *.tar.gz*:

```
tar xvfz XXX.tar.gz
```

XXX steht für den ersten Teil des Dateinamens. Endung *.tar.bz2*:

```
bzip2 -d tar xvfz XXX.tar.bz2
```

```
tar xvf XXX.tar
```

Anschließend gehen Sie mit `cd XXX` in das betreffende Verzeichnis geben ein:

```
./configure
```

```
make install
```

```
ldconfig
```

Anschließend prüfen Sie das Paket MPC (<http://www.multiprecision.org/>) auf dieselbe Weise. Die Versionsnummer muss 0.8 oder größer sein.

Zur Installation des Compilers loggen Sie sich als root ein und kopieren die mit *gcc*- beginnenden Dateien aus dem Verzeichnis *linux/gcc* der DVD in das Verzeichnis */usr/local*. Die Steuerungsdatei zur Installation (Makefile) wird mit den folgenden Befehlen erzeugt (xxx ist ein Platzhalter):

```
bzip2 -d gcc-core-4.5-xxx.tar.bz2
bzip2 -d gcc-g++-4.5-xxx.tar.bz2
tar xvf gcc-core-4.5-xxx.tar
tar xvf gcc-g++-4.5-xxx.tar
cd gcc-4.5-xxx
./configure --prefix=/usr/local/gcc45
```

Der Parameter *prefix* sorgt dafür, dass der Compiler im Verzeichnis */usr/local/gcc45* installiert wird. Wird *prefix* usw. weggelassen, wird der Compiler der neue C++-System-compiler, was vielleicht nicht erwünscht ist. Anschließend können Übersetzung und Installation mit

```
make install
```

gestartet werden. Der Prozess kann recht lange dauern, abhängig von der Maschine. Wenn Sie einen Dual- oder Quad-Core-Rechner haben, können Sie den Vorgang erheblich beschleunigen, indem Sie die Option `-j 4` für einen Quad-Core-Rechner bzw. `-j 2` für einen Dual-Core-Rechner angeben, also etwa

```
make -j4 install
```

Loggen Sie sich aus und melden Sie sich als normaler Benutzer wieder an.

A.8.2 Installation von Boost

Auf meinem System brach die Installation ab, weil Python nicht gefunden wurde. Nach Installation der Pakete *python* und *python-devel* lief alles wie geplant. Loggen Sie sich als root ein und kopieren die Datei *boost_1_45_0.tar.bz2* aus dem Verzeichnis *linux/boost* der DVD in das Verzeichnis */usr/local*. Dort entpacken Sie die Datei mit `tar --bzip2 -xf boost_1_45_0.tar.bz2`

Gehen Sie in das entstandenen Verzeichnis mit `cd boost_1_45_0` und rufen Sie dort

```
./bootstrap.sh -help
```

auf, um die Optionen zu sehen. In der Regel können Sie die Voreinstellungen beibehalten. Rufen Sie dann

```
./bootstrap.sh
```

```
./bjam
```

um die Übersetzung zu starten. Der Vorgang dauert recht lange, ein idealer Moment für eine Kaffeepause. Die erzeugten Libraries werden im Unterverzeichnis *stage/lib* abgelegt. Der nächste Schritt

```
./bjam install
```

bringt unter anderem die Libraries nach */usr/local/lib* und kopiert die Headerdateien nach */usr/include/boost*. Das ist alles.

A.8.3 Installation von Code::Blocks

Zur Installation unter Linux führen Sie bitte die folgenden Schritte durch:

1. Loggen Sie sich als Administrator (root) ein.
2. Prüfen Sie, ob *wxWidgets* installiert ist. Wenn nicht, installieren Sie es mit dem System-Tool zur Installation. Suchen Sie nach »wx« (unter SuSE 11.x heißen die Pakete *wxGTK* und *wxGTK-devel*, weil sie auf der GTK-Bibliothek basieren). Sie finden eine *wxWidgets*-Version auf der DVD im Verzeichnis *linux/codeblocks*. Andere Portierungen gibt es bei <http://www.wxwidgets.org/downloads/>. Falls notwendig, können Sie *wxWidgets* mit dem üblichen Verfahren installieren, nachdem Sie die Datei von der DVD nach */usr/local* kopiert haben:

```
bzip2 -d wxGTK-XXX.tar.bz2
tar xvf wxGTK-XXX.tar
cd wxGTK-XXX
./configure
make install
ldconfig
```

libtool, *automake* und *zip* müssen ebenfalls installiert sein.

3. Anschließend kopieren Sie die Datei *codeblocks.tgz* zum Beispiel nach */usr/local*. Dann geben Sie ein:

```
cd /usr/local
tar xvfz codeblocks.tgz
cd codeblocks
./bootstrap
make
make install
ldconfig
```

Die durch `make` angestoßene Übersetzung kann einige Zeit dauern. Nach Abschluss der Installation kann Code::Blocks gestartet werden, indem als Kommando *codeblocks* eingegeben wird.

Alternative: Wenn Sie *subversion* installiert haben, können Sie sich die jeweils neueste Version von Code::Blocks herunterladen. Gehen Sie dazu nach */usr/local* und geben dort ein:

```
svn checkout svn://svn.berlios.de/codeblocks/trunk
```

Das entstehende Verzeichnis *trunk* benennen Sie in *codeblocks* um, damit Sie später wissen, was das Verzeichnis enthält. Gehen Sie mit `cd codeblocks` in das Verzeichnis und rufen die folgenden Kommandos auf:

```
./bootstrap
./configure
make
make install
ldconfig
```

A.8.4 Code::Blocks einrichten

Starten Sie Code::Blocks durch Eingabe von *codeblocks* in einer Konsole. Zuerst wird der Compiler abgefragt. Einfach GNU GCC anklicken und mit OK bestätigen. Die aufpoppenden Tipps- und Skript-Fenster schließen. Für manche Programme werden die Boost-Library und die Dateien im Verzeichnis *cppbuch/include* benötigt. Deswegen wird Code::Blocks dafür eingerichtet. Ich gehe davon aus, dass Sie Boost wie oben beschrieben installiert haben. In der Menüleiste »Settings« klicken und »Compiler und debugger« wählen. Unter dem oberen Reiter »Compiler settings« gibt es ein wenig darunter den Reiter »Compiler Flags«. Dort anklicken:

- Produce debugging symbols [-g]
- Enable all compiler warnings [-Wall]
- Have g++ follow the coming C++0x ISO C++ language standard [-std=c++0x]

Dann den Reiter »Search directories« anklicken und darunter den Reiter »Compiler« wählen. Unter der Fläche »Add« (beziehungsweise »Hinzufügen«) anklicken und das Verzeichnis */home/user/cppbuch/include* eintragen – oder mit dem Suchbutton rechts vom Eingabefeld ermitteln. */home/user* ist ein Platzhalter, bitte für Ihr System anpassen. Als Nächstes auf dieselbe Art

```
/usr/local/include
```

eintragen. Im nächsten Schritt werden dem Linker (Reiter »Linker settings«) die Bibliotheken mitgeteilt. Dazu im Feld »Link Libraries« mit dem »Add«- oder »Hinzufügen«-Button die folgenden Dateien eintragen:

```
/usr/local/lib/libboost_regex.so
/usr/local/lib/libboost_filesystem.so
/usr/local/lib/libboost_system.so
/usr/local/lib/libboost_thread.so
/usr/local/lib/libboost_unit_test_framework.so
```

Es gibt viel mehr Boost-Libraries, aber nur die angegebenen werden von einigen der Beispiele benötigt. Jetzt unten mit OK bestätigen. Damit sind Sie für die weiteren Beispiele gerüstet, wenn es sich nur um einzelne Dateien handelt. Wie Sie mit Code::Blocks ein erstes Programm starten können, lesen Sie auf Seite [939](#).

A.8.5 Beispieldateien entpacken

Um mit den Beispieldateien zu arbeiten, kopieren Sie die Datei *cppbuch.tgz* aus dem Verzeichnis *linux* der DVD in Ihr Home-Verzeichnis. Öffnen Sie ein Shell-Fenster und gehen Sie mit `cd`-Befehlen in dieses Verzeichnis. Die Datei wird mit

```
tar xvfz cppbuch.tgz
```

entpackt. Wenn Sie danach zum Beispiel in das Verzeichnis *cppbuch/k1* gehen und das Kommando `make` eintippen, werden alle Beispiele in diesem Verzeichnis übersetzt. Die entstehenden Programme, erkennbar an der Endung *.exe*, werden durch Eingabe des Namens in das Shell-Fenster aufgerufen.

A.8.6 Installation von Qt4

Die Installation hängt von der verwendeten Linux-Distribution ab. Am einfachsten ist die Verwendung eines Linux-Systems, das Qt4 bereits enthält, einschließlich *libqt4-devel*. Bei Suse-Linux geschieht die Installation mit dem Programm Yast2. Nach Neustart des Users ist der Pfad wirksam, und ein Programm kann mit

```
qmake -project
qmake
make
```

übersetzt werden. Wie Sie Qt in ein Code::Blocks-Projekt integrieren, lesen Sie unten in Abschnitt A.8.7.



Alternative

Als Alternative bietet sich die vollständige Entwicklungsumgebung Qt SDK an, die Sie von <http://qt.nokia.com/> herunterladen können.

Lokale Installation von Qt4

Wenn Sie wollen, können Sie eine eigene Installation von Qt vornehmen, wenn Sie zum Beispiel eine neuere Qt-Version ausprobieren wollen, ohne mit der vorinstallierten Qt-Version Ihres Linux-Systems in Konflikt zu geraten. Dazu kopieren Sie die Datei *qt-everywhere-opensource-src-4.7.2.tar.gz* von der DVD in ein beliebiges Verzeichnis – oder Sie laden sich eine aktuellere Version von der Qt-Internetseite (<http://www.qt.nokia.com/>) herunter. Wenn Qt zum Beispiel in */home/user/programme/qt* installiert werden soll, geben Sie Folgendes ein:

```
tar xvfz qt-everywhere-opensource-src-4.7.2.tar.gz
cd qt-everywhere-opensource-src-4.7.2/
./configure -prefix /home/user/programme/qt
```

Geben Sie auf die Fragen `o` für Open Source ein, und `yes`, um die Lizenzbedingungen zu akzeptieren. Dann folgen die Kommandos

```
gmake
gmake install
```

Die Übersetzung dauert *lange*! Wenn Sie einen Dual- oder Quad-Core-Rechner haben, empfiehlt es sich, `gmake -j2` bzw. `gmake -j4` einzugeben. Dann nutzt `gmake` alle Prozessoren und spart Zeit. Um die Qt-Version möglichst einfach ansprechen zu können, ist es sinnvoll, den Pfad entsprechend festzulegen. Im Fall einer Bash-Shell tragen Sie in die Datei `.bashrc` des Home-Verzeichnisses ein:

```
PATH=./home/user/programme/qt/bin:${PATH}
export PATH
```

Nach Aus- und wieder Einloggen ist der Pfad wirksam.

A.8.7 Integration von Qt in ein Code::Blocks-Projekt

Die Anleitung ähnelt dem Abschnitt [A.7.5](#), aber im Detail zeigen sich doch einige Unterschiede in den Implementierungen von Qt für Windows und Linux, weswegen die 1:1-Anwendung des angegebenen Abschnitts nicht möglich ist.

Die Integration von Qt in ein Code::Blocks-Projekt wird an einem schon vorhandenen Beispiel gezeigt, damit Sie nicht so viel Tipparbeit haben. Die Hinweise sind leicht auf ein neu anzulegendes Projekt übertragbar. Nach dem Start von Code::Blocks Datei → New → Project anwählen, im erscheinenden Fenster weiter unten »QT4 project« anklicken und oben rechts mit »Go« bestätigen.

Dann geben Sie den Projekt-Namen »label« (keinen anderen, weil existierende Dateien verwendet werden!) und das Verzeichnis `...cppbuch\k14` an, in dem das Projekt gespeichert werden soll. Die Punkte sind durch den Rest des vollständigen Pfadnamens zu ersetzen. Mit »Weiter« kommen Sie zum nächsten Fenster, in dem Sie angeben, wo Qt installiert ist. Wenn Qt auf Ihrem System vorhanden ist, ist normalerweise `/usr` einzutragen. Wenn das nicht funktioniert, können Sie mit `qmake -query "QT_INSTALL_PREFIX"` den Ort erfragen. Bei einer lokalen Installation nach obiger Anleitung könnte der Ort zum Beispiel `/home/user/programme/qt` sein.

Mit »Weiter« und »Fertigstellen« beenden Sie die Eingaben. Die nächste Frage beantworten Sie bitte mit *Nein*, damit die vorhandene Datei `main.cpp` nicht überschrieben wird! Nehmen Sie nun die folgenden Einstellungen vor:

- Unter »Project« → »Properties« den Reiter »Project settings« wählen und das Kästchen neben dem Text »This is a custom Makefile« aktivieren. Damit wird zur Compilation das von `qmake` erzeugte Makefile ausgeführt.
- Danach den Reiter »Build target« anklicken und mit »Rename« das Target »Debug« in »first« umbenennen. Das von `qmake` erzeugte Makefile enthält »first« als Target.
- Im selben Fenster bei »Output filename« nur `label` eintragen (*nicht*: `bin/Debug/label!`). Mit OK bestätigen.
- Unter »Project« → »Build options« ganz links »first« anklicken und bei »Pre/post build steps« in das obere Feld den Text

```
qmake -project
qmake
```

eintragen. Die erste Anweisung erzeugt eine Steuerungsdatei für das Projekt, die zweite ein Makefile.

- Rechts den Reiter »Make commands« anklicken und in die Zeile »Clean project/target« rechts `$make -f $makefile clean` eintragen (d.h. `$target` löschen). Mit OK bestätigen. Wenn Sie links »Sources« expandieren und auf *main.cpp* klicken, wird die Datei im Editor angezeigt. Übersetzung und Ausführung des Programms können nun wie üblich über die Menüleiste (Build) oder die Taste F9 gestartet werden. Diese Einstellungen müssen für *jedes* Qt-Projekt vorgenommen werden!

Glossar

Dieses Glossar enthält Kurzdefinitionen der wichtigsten Begriffe der objektorientierten Programmierung und verwandter Bereiche. Das Glossar ist zum Nachschlagen und zur Wiederauffrischung der Begriffe gedacht, nicht zur Einführung. Entsprechende Textstellen können über das Stichwortverzeichnis gefunden werden.

Abstrakter Datentyp

Ein Abstrakter Datentyp fasst *Daten* und *die Funktionen*, mit denen die Daten bearbeitet werden dürfen, zusammen. Der Sinn liegt darin, den richtigen Gebrauch der Daten sicherzustellen. Mit »Funktion« ist hier *nicht* die konkrete Implementierung gemeint, das heißt, *wie* die Funktion im Einzelnen auf die Daten wirkt. Zur Benutzung eines Abstrakten Datentyps reicht die Spezifikation der Zugriffsoperation aus. Ferner sind logisch zusammengehörige Dinge an einem Ort konzentriert. Ein Abstrakter Datentyp ist ein $\rightarrow$ *Typ* zusammen mit $\rightarrow$ *Datenkapselung*. Eine $\rightarrow$ *Klasse* in C++ ist ein Abstrakter Datentyp.

Abstrakte Klasse

Eine abstrakte Klasse ist eine $\rightarrow$ *Klasse*, von der es keine $\rightarrow$ *Instanzen* gibt. Abstrakte Klassen definieren $\rightarrow$ *Schnittstellen*, die durch abgeleitete Klassen implementiert werden müssen.

Aggregation

Die Aggregation ist ein Spezialfall der $\rightarrow$ *Assoziation*, der Enthaltensein (Teil-Ganzes-Beziehung) beschreibt. Wenn im Ganzen nur Verweise auf die Teile existieren, sind diese nicht existentiell abhängig vom Ganzen. Andernfalls spricht man auch von Komposition. In diesem Fall werden die Teile zusammen mit dem Ganzen zerstört. Zur Umsetzung in C++ siehe Seite [585](#).

Attribut

Attribute beschreiben die Eigenschaften eines Objekts. Der aktuelle Zustand eines Objekts wird durch die Werte der Attribute beschrieben. Zum Beispiel kann *Farbe* ein Attribut sein; ein möglicher Attributwert wäre *rot*.

Ausnahme

Eine Ausnahme (englisch *exception*) ist die Verletzung der $\rightarrow$ *Vorbedingung* einer Operation ($\rightarrow$ *Methode*) einer Klasse. C++ bietet die Möglichkeit, Ausnahmen zu erkennen und zu behandeln (*exception handling*).

Assoziation

Die Assoziation ist eine gerichtete Beziehung zwischen Klassen. Sie kann in eine Richtung verweisen (A kennt B, aber nicht umgekehrt) oder bidirektional sein (A kennt B und B kennt A).

Behälterklasse

Datenstruktur zur Speicherung von Objekten. Beispiele: Array, Liste, Vektor

Bindung

$\rightarrow$ *dynamische Bindung*, $\rightarrow$ *statische Bindung*

Botschaft

In der rein objektorientierten Programmierung wird davon ausgegangen, dass ein laufendes Programm(-system) aus einer Menge von Objekten besteht, die miteinander über Botschaften (englisch *messages*) kommunizieren. Ein genauerer Begriff als Botschaft ist *Aufforderung*, weil das empfangende Objekt etwas tun soll. Ein Objekt, das eine Aufforderung erhält, führt eine dazu passende Operation ($\rightarrow$ *Methode*) aus, die in der $\rightarrow$ *Klasse* beschrieben ist.

Client

Im informationstechnischen Sprachgebrauch heißen Dinge (Objekte, Rechner, ...), die eine Dienstleistung erbringen, *Server*. Die Dienstleistung wird erbracht für einen *Client* (deutsch: Klient, Kunde), der selbst ein Rechner oder Objekt sein kann.

Container

$\rightarrow$ *Behälterklasse*

Daten

Der Zustand eines Objekts wird durch seine Daten beschrieben. Die Daten sind die Werte der $\rightarrow$ *Attribute* eines Objekts.

Datenkapselung

Datenkapselung ist das »Verstecken« der Daten eines Objekts vor direkten Zugriffen. Zugriffe sind nur über die öffentliche → *Schnittstelle* der Datenkapsel (→ *Abstrakter Datentyp*) möglich. Datenbezogene Fehler sind damit leicht lokalisierbar. In C++ wird Datenkapselung mit der Zugriffsspezifikation `private` realisiert.

Definition

Eine *Definition* liegt vor, wenn *mehr als nur der Name eingeführt wird*, zum Beispiel wenn Speicherplatz angelegt werden muss für Daten oder Code oder die innere Struktur eines Datentyps beschrieben wird, aus der sich der benötigte Speicherplatz ergibt. Weil *auch* ein Name eingeführt wird, ist eine Definition immer auch eine Deklaration. Die Umkehrung gilt nicht.

Deklaration

Eine *Deklaration* teilt dem Compiler mit, dass eine Funktion (oder eine Variable) mit diesem Aussehen irgendwo definiert ist. Damit kennt er den Namen bereits, wenn er auf einen Aufruf der Funktion stößt, und ist in der Lage, eine Syntaxprüfung vorzunehmen. Eine Deklaration führt einen Namen in ein Programm ein und gibt dem Namen eine Bedeutung. Eine Deklaration kann gleichzeitig eine → *Definition* sein.

Delegation

Ein Objekt wird durch den Aufruf einer Methode aufgefordert, eine Dienstleistung zu erbringen. Diese Aufforderung kann an ein weiteres Objekt zur Bearbeitung weitergeleitet (= delegiert) werden.

Distribution

Die Zusammenstellung von Programm(en) samt Dokumentation und anderen Dateien (Bilder, Audiodateien) in einer zur Verteilung geeigneten, in der Regel gepackten Form, wird Distribution genannt.

Dynamische Bindung

Wenn sich erst während des Programmlaufs ergibt, welche Methode für ein Objekt aufgerufen werden soll, kann das Binden noch nicht zur Compiler- oder zur Link-Zeit erfolgen, sondern eben erst später (englisch *late binding*). In C++ wird im Allgemeinen während der Übersetzung sichergestellt, dass nur zulässige Aufrufe einer Methode möglich sind (Ausnahme: siehe Beispiele zum `dynamic_cast`). In anderen Sprachen, zum Beispiel *Smalltalk*, wird die Überprüfung ausschließlich erst zur Laufzeit vorgenommen, wodurch auf Kosten einiger Aspekte der Programmsicherheit eine größere Flexibilität ermöglicht wird.

Frühe Bindung

→ *statisches Binden*

GNU

Das 1984 begonnene GNU-Projekt (<http://www.gnu.org/>) hatte die Entwicklung eines freien Unix-ähnlichen Betriebssystems zum Ziel. Auch wenn dieses Ziel nicht erreicht wurde, ist dabei jede Menge freier Software entstanden, siehe auch → *Open Source*.

Identität

Ein Objekt besitzt eine *Identität*, die es unterscheidbar macht von einem beliebigen anderen Objekt, selbst wenn beide gleiche Daten enthalten. Die Identität zu einem bestimmten Zeitpunkt wird durch eine eindeutige Position im Speicher gewährleistet; zwei Objekte können niemals dieselbe Adresse haben, es sei denn, ein Objekt ist im anderen enthalten. C++ hat keine Sprachmittel für die Identität. Die Adresse als Identitätsmerkmal gilt nur für ein → *vollständiges Objekt* und dann auch bei Mehrfachvererbung. Falls dies nicht ausreichend sein sollte, kann die Identität durch ein eigens für diesen Zweck vorgesehenes Element des Objekts definiert werden, zum Beispiel durch eine Seriennummer.

Initialisierung

Wenn ein Objekt *während der Erzeugung* mit Anfangsdaten versehen wird, heißt der Vorgang *Initialisierung*. Die Initialisierung ist die Aufgabe eines Konstruktors. Sie ist von der → *Zuweisung* zu unterscheiden.

Instanz

Eine Instanz einer Klasse ist eine andere Bezeichnung für ein → *Objekt*. Die Erzeugung eines Objekts wird auch Instanziierung genannt.

Interface

→ *Schnittstelle*

Kapselung

→ *Datenkapselung*

Klasse

Eine Klasse definiert die Merkmale (Daten) und das Verhalten (Operationen, Methoden) einer Menge von Objekten. Eine Klasse ist ein Datentyp, genauer: ein → *Abstrakter Datentyp*. In C++ gilt die Umkehrung (ein Datentyp ist eine Klasse) *nicht*, weil die Grunddatentypen (zum Beispiel `int`) und darauf aufbauende zusammengesetzte Typen (zum Beispiel C-Array) nicht als Klasse implementiert sind. Eine Klasse definiert die Struktur aller nach ihrem Muster erzeugten Objekte, entweder direkt oder indirekt durch → *Vererbung*.

Klassifikation

Klassifikation ist ein Verfahren, um Gemeinsamkeiten von Dingen herauszufinden und auszudrücken. Von Unterschieden wird abstrahiert. In C++ wird ein Satz gleicher Merkmale und Verhaltensweisen durch die → *Klasse* beschrieben.

Komplexität

→ *Zeitkomplexität*

Lexikografischer Vergleich

Ein lexikografischer Vergleich ist ein Vergleich, wie er bei der Sortierung von Begriffen in einem Lexikon verwendet wird. Danach entscheiden die jeweils ersten beiden Elemente zweier zu vergleichender Folgen, welche Folge als kleiner aufgefasst wird. Falls jedoch die jeweils ersten Elemente *gleich* sind, werden die jeweils zweiten Elemente zum Vergleich herangezogen usw. Zum Beispiel würde bei den Zeichenfolgen »Objekt« und »Oberklasse« erst der dritte Buchstabe über die Sortierung entscheiden.

Linken

Beim *statischen* Linken werden die Bibliotheksmodule zu dem ausführbaren Programm gebunden. Die so erzeugte ausführbare Datei ist dementsprechend größer. *Vorteil*: Sie kann auf einen anderen Computer derselben Bauart und mit demselben Betriebssystemtyp kopiert werden und funktioniert dort wie auf dem Originalsystem. *Nachteil*: Wenn N Programme dieselbe statische Bibliothek benötigen, wird N mal der zugehörige Speicherplatz gebraucht, wenn die Programme gleichzeitig laufen.

Dynamisches Linken heißt, dass Bibliotheksmodule *nicht* in der ausführbaren Datei enthalten sind, sondern erst bei Ausführung des Programms dazugebunden werden. *Vorteil*: Wenn beliebig viele Programme dieselbe dynamische Bibliothek benötigen, wird der zugehörige Speicherplatz nur einmal gebraucht. *Nachteil*: Die ausführbare Datei funktioniert auf einem anderen Computer derselben Bauart und mit demselben Betriebssystemtyp nur, wenn die dynamische Bibliothek dort installiert ist. Wie dynamische und statische Bibliotheken erzeugt werden, lesen Sie in Abschnitt 23.4.

L-Wert

Ein L-Wert (Links-Wert, (englisch *lvalue*)) ist ein Ausdruck, der im Kontext seines Auftretens als (symbolische) *Adresse* eines Objekts oder einer Funktion aufgefasst werden kann. Ein L-Wert kann auf der linken Seite einer Zuweisung auftreten, daher der Name (siehe Seite 62). Ein L-Wert kann auch auf der rechten Seite einer Zuweisung auftreten, ein R-Wert jedoch *nur* auf der rechten Seite. In der Regel muss ein Objekt, das verändert werden soll, als L-Wert vorliegen. Alle Ausdrücke, die keine L-Werte sind, sind R-Werte (englisch *rvalue*)¹. Ein R-Wert repräsentiert den Wert eines Objekts, nicht seine Adresse. Dazu gehört die rechte Seite einer Zuweisung, aber auch eine temporäre Kopie, zum Beispiel vom Kopierkonstruktor bei der Rückgabe eines Funktionswerts erzeugt. Von einem L-Wert kann die Adresse ermittelt werden, von einem R-Wert nicht. Einige Beispiele:

- Der Indexoperator (siehe Seite 326) liefert einen L-Wert zurück:

```
a[i] = 5;
```

- Wenn ein L-Wert in einem Kontext auftritt, wo ein R-Wert erwartet wird, wird er in einen R-Wert umgewandelt:

¹ Nach [ISO C++], Abschnitt 3.10, ist die Unterscheidung noch differenzierter. Da gibt es noch die Wertkategorien *glvalue*, *prvalue* und *xvalue*.

```
x = a[i];
```

- Eine binärer Operator liefert einen R-Wert zurück, in diesem Fall ein temporäres Ergebnis, das *c* zugewiesen wird:

```
c = operator+(a, b);
```

Mehrfachverbung

Eine Klasse kann in C++ von mehr als einer Klasse erben (→ *Vererbung*).

Methode

Methode ist eine andere Bezeichnung für eine Operation, die auf den Daten eines → *Objekts* ausgeführt werden kann. In C++ heißen Methoden auch Elementfunktionen (englisch *member functions*), um auszudrücken, dass eine Methode ein Element einer Klasse ist. Sie unterscheidet sich von einer normalen Funktion auch dadurch, dass sie Zugriff auf die internen Daten des → *Objekts* hat.

Nachbedingung

Die Nachbedingung ist die Spezifikation einer → *Methode*, eines Programmschritts oder einer Funktion. Sie beschreibt ausgehend vom Zustand des Programms *vor* Ausführung (→ *Vorbedingung*), welchen Zustand ein Programm *nach* der Ausführung hat.

MIME (Multipurpose Internet Mail Extension)

erlaubt es, Multimedia-Inhalte binären Formats an E-Mails zu binden. Der Inhaltstyp, oft MIME-Typ genannt, bestimmt die Art der Verarbeitung. Es können bei der IANA [\[MIME\]](#) registrierte Typen oder selbst definierte sein, die der vorgeschriebenen Syntax genügen. Der MIME-Typ wird nicht nur bei E-Mails, sondern auch in anderen Zusammenhängen verwendet (z.B. Browser).

Nachricht

→ *Botschaft*

Oberklasse

Es kann verschiedene Klassen geben, die gemeinsame Anteile enthalten. Diese Anteile können »herausgezogen« werden und bilden eine Oberklasse. Die Klassen werden dann als Spezialisierung der Oberklasse aufgefasst, weil sie nur noch die Unterschiede beschreiben. Zum Beispiel haben eine Tanne und eine Eiche die gemeinsame Eigenschaft, ein Baum zu sein mit all seinen Merkmalen. In einer Beschreibung (Klasse) für eine Tanne genügt es, auf die Oberklasse »Baum« zu verweisen (→ *Vererbung*) und nur die Besonderheit »Nadeln« anzugeben. Eine Oberklasse ist eine durch → *Klassifikation* gewonnene Abstraktion in der Form einer *ist-ein*-Beziehung. Ein Tanne »ist ein« Baum – eine Eiche auch. Eine Oberklasse kann selbst wieder von einer weiteren Oberklasse erben. Manchmal wird eine Oberklasse oder die oberste Oberklasse »Basisklasse« (englisch *base class*) genannt.

Objekt

Ein Objekt ist die konkrete Ausprägung des durch eine $\rightarrow$ *Klasse* definierten Datentyps. Es hat einen inneren Zustand, der durch Attribute in Form von anderen Objekten oder Elementen der in der Programmiersprache vorgegebenen Datentypen dargestellt wird. Der Zustand kann sich durch Aktivitäten des Objekts ändern, also durch Ausführen von Operationen auf Objektdaten. Jedes Objekt hat eine Identität, sodass auch gleiche Objekte unterscheidbar sind. Im Ablauf eines Programms werden Objekte erzeugt (und wieder gelöscht), die aufgrund des Empfangs einer $\rightarrow$ *Botschaft* hin aktiv werden. Die Menge aller möglichen Botschaften für ein Objekt heißt $\rightarrow$ *Schnittstelle*.

Open Source

Bei Open Source-Software sind die Quellen, wie der Name sagt, frei zugänglich. Open Source-Software ist meistens kostenlos. Sie kann modifiziert, kopiert und weitergegeben werden. Allerdings heißt Open Source nicht, dass alles erlaubt ist. Was erlaubt ist, ist nicht einheitlich geregelt, sodass es für verschiedene Open Source-Software unterschiedliche Lizenzen gibt. Die Bekannteste ist unter <http://www.gnu.org/licenses/> erhältliche »GNU General Public License«.

Operation

$\rightarrow$ *Methode*, $\rightarrow$ *Botschaft*

POD (plain old data type)

Dieser Begriff kennzeichnete Datentypen, die es auch in der Sprache C geben kann (daher »plain old«), die also ohne C++-Eigenschaften auskommen. Beispiel:

```
struct Punkt_POD {          struct Punkt_keinPOD {
    int x;                   int x;
    int y;                   int y;
};                           Punkt_keinPOD(int x, int y);
                             };

```

Die ausführliche Definition finden Sie in [ISOC++], Kapitel 9.

Polymorphismus

Die Fähigkeit von Programmelementen, sich zur *Laufzeit* auf $\rightarrow$ *Objekte* verschiedener $\rightarrow$ *Klassen* beziehen zu können, heißt Polymorphismus. Anders formuliert: Erst zur Laufzeit eines Programms wird die zu dem jeweiligen Objekt passende Realisierung einer Operation ermittelt. In C++ müssen diese Klassen in einer Vererbungsbeziehung stehen ($\rightarrow$ *Vererbung*). Mit Polymorphismus eng verknüpft ist der Begriff $\rightarrow$ *dynamische Bindung*.

Resource Acquisition Is Initialization (RAII)

Ein Objekt wird durch den Konstruktor initialisiert. RAII ist das Prinzip, Ressourcen durch die Initialisierung (das heißt durch den Konstruktor) zu belegen. Gleichzeitig ist damit die Freigabe der Ressource durch den Destruktor verbunden. Die Vorteile: Die Freigabe geschieht erst am Ende der Lebensdauer des Objekts. Sie erfolgt automatisch auch im Fehlerfall (Exception), ohne dass sie gesondert programmiert werden muss (wegen der

Freigabe durch den Destruktor). Dieses Prinzip wird oft vorteilhaft eingesetzt. Beispiele: automatisches Schließen von Dateien, exception-sichere Beschaffung von Ressourcen mit `shared_ptr` (Seite 567), Synchronisation mit Mutex-Variablen (Seite 426 ff.), Unit-Test-Fixtures (Seite 534), `sentry`-Objekte zur Absicherung von Ein-/Ausgabeoperationen (Seiten 835, 836 und andere).

R-Wert

→ *L-Wert*

Schnittstelle

Als (öffentliche) Schnittstelle (englisch *(public) interface*) bezeichnet man die Menge von Aufforderungen, auf die ein → *Objekt* reagieren kann. In C++ werden Schnittstellen durch die → *Deklarationen* der `public`- → *Methoden* beschrieben.

Server

→ *Client*

Signatur

Die Signatur besteht aus der Kombination des Funktionsnamens mit der Reihenfolge und den Typen der Parameterliste. Anhand der Signatur kann der Compiler überladene Funktionen erkennen. Manche betrachten auch den Rückgabetyt als Teil der Signatur.

Spätes Binden

→ *Dynamische Bindung*

Statische Bindung

Ein Funktionsaufruf muss an eine Folge von auszuführenden Anweisungen gebunden werden. Der Aufruf einer Funktion (oder Methode, Operation) heißt statisch gebunden, wenn bereits der Compiler oder der Binder (Linker) die Funktion einbindet, also *vor dem Programmstart*. Die Typverträglichkeit und Zulässigkeit von Funktionsaufrufen kann damit sehr früh geprüft werden. Siehe auch → *Dynamische Bindung*.

Subtyp

Ein Subtyp ist ein abgeleiteter → *Typ*, wobei ein Objekt eines Subtyps jederzeit an die Stelle eines Objekts der → *Oberklasse* treten kann. Ein abgeleiteter Typ, der nicht vollständig das Verhalten der Oberklasse zeigt, ist kein echter Subtyp.

Typ

Ein Typ ist die Menge aller Objekte in einem System, die auf dieselbe Art auf eine Menge von → *Botschaften* reagieren. Diese Objekte haben dieselbe öffentliche → *Schnittstelle*. In C++ wird ein Typ durch eine → *Klasse* beschrieben.

UML

Die Unified Modeling Language (UML) ist eine weit verbreitete grafische Beschreibungssprache für Klassen, Objekte und noch mehr. Sie wird vornehmlich in der Phase des Softwareentwurfs eingesetzt. Grundlagen und Anwendung sind zum Beispiel in [Oe] beschrieben.

Unterklasse

Eine Klasse, zu der eine $\rightarrow$ *Oberklasse* existiert, heißt Unterklasse bezüglich dieser Oberklasse. Wenn ein Objekt der Unterklasse stets an die Stelle eines Oberklassenobjekts treten kann, ist die Unterklasse ein $\rightarrow$ *Subtyp* der Oberklasse. Eine Unterklasse heißt auch »abgeleitete Klasse« ((englisch *derived class*)).

Vererbung

Vererbung wird definiert durch eine Beziehung zu einer $\rightarrow$ *Oberklasse*, um deren Merkmale und Verhaltensweisen zu übernehmen. Eine Klasse »erbt« von Oberklassen, indem die direkten Oberklassen in der Klassendefinition angegeben werden. Gleichzeitig wird damit von allen Oberklassen der Oberklasse geerbt, sofern sie existieren. Der Aufruf einer Operation für ein Objekt lässt nicht erkennen, ob sie der Klasse des Objekts oder einer Oberklasse zuzuordnen ist, also geerbt wurde.

Vertrag

Eine $\rightarrow$ *Methode* gewährleistet die Einhaltung ihrer Spezifikation, wenn der Aufrufer die $\rightarrow$ *Vorbedingung* einhält (zum Beispiel Aufruf der Methode mit korrekten Parametern). Die Methode erfüllt damit einen Vertrag.

Vollständiges Objekt

Unter einem »vollständigen Objekt« wird ein Objekt verstanden, das nicht als Subobjekt dient, also nicht in einem anderen Objekt durch Vererbung enthalten ist.

Vorbedingung

Die Vorbedingung beschreibt den Zustand eines Programms, der notwendig ist, um den nächsten Programmschritt korrekt durchführen zu können. Der Programmschritt kann der Aufruf einer $\rightarrow$ *Methode* sein.

Zeitkomplexität

Der Begriff Komplexität aus der Informatik gibt an, wie die von einem Algorithmus benötigte Zeit und der benötigte Speicherplatz von der Anzahl der im Container gespeicherten Elemente abhängen. Meistens interessiert nur die benötigte Zeit (Zeitkomplexität). Von den Eigenschaften der Maschine, Betriebssystem und der Programmiersprache wird dabei abstrahiert. Letztere gehen mit einem konstanten Faktor ein. Die Zeitkomplexität von Algorithmen wird üblicherweise in der O-Notation angegeben. Dabei meint $O(n)$, dass der Zeitaufwand proportional zur Anzahl n der Elemente (eines Containers) ist. $O(\log n)$ bedeutet, dass der Zeitaufwand proportional zum Logarithmus von n ist, und $O(1)$, dass der Zeitaufwand konstant bzw. unabhängig von n ist.

Zustand

Der Zustand eines Objekts ist definiert durch die Menge der Werte seiner $\rightarrow$ *Attribute*.

Zuweisung

Eine Zuweisung weist ein Objekt dem anderen zu und ändert damit dessen Wert. Im Unterschied zur $\rightarrow$ *Initialisierung* muss das zu ändernde Objekt vor der Zuweisung bereits existieren.

Zusicherung

Eine Zusicherung (englisch *assertion*) ist eine logische Bedingung, die erfüllt sein muss. Zusicherungen dienen der Verifikation von Programmen, das heißt dem Nachweis, dass ein Programm seiner Spezifikation entspricht. $\rightarrow$ *Vorbedingungen* und $\rightarrow$ *Nachbedingungen* sind Beispiele für Zusicherungen.

Literatur- verzeichnis

- [Abr] Dave Abrahams: *Want Speed? Pass by Value*.
<http://cpp-next.com/archive/2009/08/want-speed-pass-by-value/>
- [ALSU] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullmann: *Compiler*. Pearson 2008
- [Alex] Andrei Alexandrescu: *Modern C++ Design*. Addison-Wesley 2001
- [asio] Christopher Kohlhoff: *Boost.Asio*.
http://www.boost.org/doc/libs/1_45_0/doc/html/boost_asio.html
oder die lokale Dokumentation Ihres Boost-Systems
- [Bal] Heide Balzert: *Lehrbuch der Objektmodellierung*. Spektrum Akademischer Verlag 2004
- [Beck] Kent Beck: *Extreme Programming Explained: Embrace Change*. Addison-Wesley 2004
- [BeckP] Pete Becker: *The C++ Standard Library Extensions*. Addison-Wesley 2007
- [BlSu] Jasmin Blanchette, Mark Summerfield: *C++ GUI-Programming with Qt 4*. Prentice Hall 2008
- [boost] Boost-Homepage: <http://www.boost.org/>
- [Br] Ulrich Breymann: *Assignment and Polymorphism*, Journal of Object-Oriented Programming 13, No. 11, März 2001, S. 20-24
- [CB] Code::Blocks-Homepage <http://www.codeblocks.org/>
- [CE] Krzysztof Czarnecki, Ulrich Eisenecker: *Generative Programming*. Addison-Wesley 2000
- [CERT] CERT C++ Secure Coding Standard,
<https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637>
- [CLR] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*. MIT Press 2009 (auch in deutscher Sprache erhältlich)

- [Date] C. J. Date: *An Introduction to Database Systems*. Addison-Wesley 2003
- [Ecma] Standard ECMA-262, <http://www.ecma-international.org/publications/standards/Ecma-262.htm>
- [ES] Margaret A. Ellis, Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley 1990
- [Fiel] R. Fielding et al.: *Hypertext Transfer Protocol – HTTP/1.1*, <http://www.ietf.org/rfc/rfc2616.txt>, 1999
- [Fri] Jeffrey E.F. Friedl: *Mastering Regular Expressions*. 3. Auflage, O'Reilly 2006
- [Gamma] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*. Addison-Wesley 1995
- [GCC] Homepage der GNU Compiler Collection: <http://gcc.gnu.org>
- [GNU] GNU Autoconf und Verwandte, <http://www.gnu.org/software/autoconf/#family>
- [GrJ] Douglas Gregor, Jaakko Järvi: *Variadic Templates for C++0x*, in Journal of Object Technology, vol. 7, no. 2, February 2008, pp. 31-51, http://www.jot.fm/issues/issue_2008_02/article2/
- [Her] Helmut Herold: *Linux/Unix-Systemprogrammierung*. Addison-Wesley 2004
- [HeNy] Mats Henricson, Erik Nyquist: *Programming in C++ – Rules and Recommendations* (auf der DVD vorhanden)
- [Hof06] Douglas R. Hofstadter: *Gödel, Escher, Bach. Ein Endloses Geflochtenes Band*. Klett-Cotta 2006
- [ISOC] ISO/IEC 9899-201x: *C Standard*, Committee Draft, 16. November 2010. <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1539.pdf>
- [ISOC++] ISO/IEC JTC 1/SC22/WG21: *Programming Language C++*, Final Draft International Standard, 11. April 2011. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3290.pdf>
- [JSF] JSF AV C++ Coding Standards, Lockheed Martin 2005, <http://www.research.att.com/~bs/JSF-AV-rules.pdf> (auf der DVD vorhanden)
- [KL] Klaus Kreft, Angelika Langer: *Standard C++ IOStreams and Locales*. Addison Wesley 2008
- [Lis] Barbara Liskov: *Data Abstraction and Hierarchy*, Addendum to Proc. of the ACM Conference on Object-Orientated Programming Systems, Languages, and Applications (OOPSLA '87). SIGPLAN Notices 23, 5 (May 1988)
- [make] GNU make, <http://www.gnu.org/software/make/manual/make.html>
- [Mar] Rudolf Marty: *Methodik der Programmierung in Pascal*, Springer 1994
- [Mey] Bertrand Meyer: *Touch of Class*. Springer 2009
- [Mill] Peter Miller: *Recursive Make Considered Harmful*, <http://miller.emu.id.au/pmiller/books/rmch/>
- [MIME] IANA – Internet Assigned Numbers Authority: *MIME Media Types*, <http://www.iana.org/assignments/media-types/index.html>

- [Neun] Alexander Neundorf: *Why the KDE project switched to CMake – and how*, <http://lwn.net/Articles/188693/>. Siehe auch http://www.linuxmagazin.de/heft_abo/ausgaben/2007/02/mal_ausspannen
- [NTP] D. Mills: *RFC 4330: Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI*. <http://www.ietf.org/rfc/rfc4330.txt>
- [Oe] Bernd Oestereich: *Analyse und Design mit UML 2.3*. Oldenbourg 2009
- [OON] The Object-Oriented Numerics Page, <http://oonumerics.org/oon/>
- [Port] IANA – Internet Assigned Numbers Authority: *PORT NUMBERS*, <http://www.iana.org/assignments/port-numbers>
- [Roz] Gennadiy Rozental: *Boost Test Library*. http://www.boost.org/doc/libs/1_45_0/libs/test/doc/html/index.html oder die lokale Dokumentation Ihres Boost-Systems
- [ScM] Scott Meyers: *Effective C++*, 3. Auflage. Addison-Wesley 2005
- [ScMb] Scott Meyers: *Mehr Effectiv C++ programmieren*. Addison-Wesley 1997
- [SFP] Ben Collins-Sussmann, Brian W. Fitzpatrick, C. Michael Pilato: *Version Control with Subversion*, <http://svnbook.red-bean.com/> (auf der DVD vorhanden)
- [SL] Andreas Spillner, Tilo Linz: *Basiswissen Softwaretest*, Dpunkt 2010.
- [SQLite] SQLite Homepage: <http://www.sqlite.org/>
- [SSRB] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann: *Pattern-Oriented Software Architecture – Volume 2, Patterns for Concurrent and Networked Objects*. Wiley 2000
- [Str] Bjarne Stroustrup: *The C++-Programming Language*, Addison-Wesley 2000 (auch in deutscher Übersetzung erhältlich)
- [Str94] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley 1994
- [svn] Subversion Homepage: <http://subversion.tigris.org/>
- [thread] Anthony Williams: *Thread*. http://www.boost.org/doc/libs/1_45_0/doc/html/thread.html oder die lokale Dokumentation Ihres Boost-Systems
- [TR1] Draft Technical Report on C++ Library Extensions, <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf>
- [Unic] Homepage der Unicode-Organisation, <http://www.unicode.org/>
- [Unr] Erwin Unruh, Primzahlenprogramm, <http://www.erwin-unruh.de/primorig.html>
- [URI] T. Berners-Lee, R. Fielding, L. Masinter: *Uniform Resource Identifier (URI): Generic Syntax*, <http://www.ietf.org/rfc/rfc3986.txt>
- [VaJo] David Vandevoorde, Nicolai M. Josittis: *C++ Templates – The Complete Guide*. Addison-Wesley 2003
- [vdL] Peter van der Linden: *Expert C Programming*. SunSoft Press/Prentice Hall 1994

- [Ve95] Todd Veldhuizen: *Using C++ template metaprograms*. *C++ Report* 7, No. 4, May 1995, Seiten 36–43.
Siehe auch »Blitz«-Projekt im Internet: <http://oonumerics.org/blitz/>
- [Ve95a] Todd Veldhuizen: *Expression Templates*. *C++ Report* 7, No. 5, May 1995, Seiten 26–31.

**Hinweis**

Internet-Verweise unterliegen häufig Änderungen. Es kann daher sein, dass die angegebenen Links nach Druck des Buchs nicht mehr alle auffindbar sind. Der letzte Zugriff auf alle Links fand am 4. Juni 2011 statt.

Register

Symbole

* 45, 186, 403
*= 45, 332
+, += 45
++ 45, 334, 399, 403
+= 45
, 76, 210
-, -= 45
->* 230
-- 45, 336
.* 230
... 306
/ 45
/* ... */ 31
// 31
/= 45
:: 59, 152, 269
::* 230
|| 55
|= 45
! 45, 55
!= 54, 55, 403
; 33, 66
<, <= 45, 54
< 45
<< 45, 96, 221, 322, 377
<<= 45
=, == 45, 54, 55
>, >= 45, 54
>> 45, 94, 221, 381, 735
>>= 45
?: 67
[] 190, 191, 326
[][]
 Matrixklasse 359
 Zeigerdarstellung 212
132
%, %= 45
& Adress-Operator 56
&, &= Bit-Operatoren 45
&& log. UND 55
&& Shell 602
&& R-Wert 591
\\ 32, 52, 130
\\0 193, 196–198
\\", \\a, \\b 52
\\f, \\n, \\r, \\t, \\v 52, 94
\\x 52
\\ 52
~ 45
^ 45
\$, \$^, \$@ 521
@D, @F 607
" 34, 193, 194

A

- abgeleitete Klasse 257, 266, 269
 - und virtueller Destruktor 281
- Abhängigkeit (make) 519
 - automatische Ermittlung 602
- abort() 878
- abs() 50, 695, 696, 863, 876, 878
- Abstrakter Datentyp 148, 949
- abstrakte Klasse 275, 949
- Abstraktion 258
- accumulate() 650
- acos() 696, 863, 876
- acosh() 696
- Adapter, Iterator- 811
- Additionsoperator 319
- adjacent_difference() 653
- adjacent_find() 679
- adjustfield 379
- Adresse 186
 - symbolische 34
- Adressierung
 - indirekte 870
- Adressoperator 187
- advance() 810
- Aggregation 585, 949
- Aktualparameter 104
- <algorithm> 623, 815
- Algorithmus 28, 399
 - accumulate() 650
 - adjacent_difference() 653
 - adjacent_find() 679
 - binary_search() 681
 - copy() 717
 - copy_backward() 717
 - copy_if() 718
 - copy_n() 719
 - count() 661
 - count_if() 661
 - equal() 694
 - equal_range() 682
 - fill() 648
 - fill_n() 648
 - find() 674
 - find_end() 678
 - find_first_of() 675
 - for_each() 716
 - generate(), _n() 648
 - includes() 684
 - inner_product() 651
 - inplace_merge() 673
 - iota() 649
 - is_heap() 692
 - iter_swap() 719
 - lexicographical_compare() 665
 - lower_bound() 682
 - make_heap() 691
 - max() 726
 - max_element() 655
 - merge() 671
 - mergesort() 672
 - min() 726
 - min_element() 655
 - minmax() 726
 - minmax_element() 655
 - mismatch() 692
 - next_permutation() 664
 - nth_element() 669
 - partial_sort(), -_copy 669
 - partial_sum() 653
 - partition() 666
 - pop_heap() 689
 - prev_permutation() 663
 - push_heap() 690
 - random_shuffle() 657
 - remove(), _if(), _copy(), _copy_if() 723
 - replace(), _if(), _copy(), _copy_if() 722
 - reverse(), _copy() 660
 - rotate(), _copy() 656
 - search() 677
 - search_n() 680
 - set_difference() 686
 - set_intersection() 686
 - set_symmetric_difference() 687
 - set_union() 685
 - sort() 667
 - sort_heap() 691
 - stable_partition() 666
 - stable_sort() 667
 - swap() 719
 - swap_ranges() 720

- transform() 720
 - unique(), _copy() 658
 - upper_bound() 682
 - Alias-Name 55, 187, 189
 - *this 209
 - alignof() 355
 - allgemeiner Konstruktor 155
 - allocator 744, 855
 - Anführungszeichen 34, 193, 194
 - Anker 412
 - anonymer Namespace 125
 - ANSI-Sequenzen zur Bildschirm-
ansteuerung 52
 - Anweisung 61
 - any() 803
 - app 390
 - append() 843
 - application/x-www-form-urlencoded 488,
490
 - apply() 860
 - Äquivalenzbeziehung 682
 - Äquivalenzklasse 526
 - arg() 695, 696
 - argc 208
 - Argumente *siehe* Parameter
 - argv[] 208
 - Arität (Template) 253
 - Arithmetik mit Iteratoren 402
 - Arithmetik mit Zeigern 192
 - arithmetische Operatoren 45
 - Array
 - char 195
 - von C-Strings 199
 - dynamisches 201, 213, 214, 324
 - Freigabe 204
 - als Funktionsparameter 211
 - mehrdimensionales 209, 213, 214
 - Matrixklasse 360
 - valarray 857
 - vs. Zeiger 190
 - <array> 764, 780
 - Array2d 215, 506
 - ASCII 825
 - Dateien 221
 - Tabelle 53, 887
 - asctime() 881
 - asin() 696, 863, 876
 - asinh() 696
 - assert() 133
 - mit Exception 309
 - assign() 770, 844
 - Assoziation (UML) 582
 - Assoziativität von Operatoren 56, 890
 - at() 82, 86, 771, 775, 781, 784, 843
 - atan() 696, 863, 876
 - atan2() 863, 876
 - atanh() 696
 - ate 390
 - atexit() 878
 - atof(), atoi(), atol() 628, 878
 - atoi() 643
 - Atom-Uhr 485
 - Attribute 150, 950
 - Aufforderung 29, 950
 - Aufzählungstyp 79
 - Ausdruck
 - Auswertung 56
 - Definition 42
 - mathematischer 49, 116
 - Ausgabe 93, 96, 377
 - benutzerdefinierter Typen 377
 - Datei- 96, 375
 - Formatierung 377
 - Weite der 378
 - Ausgabeoperator 322, 377
 - Ausnahme 950
 - Ausnahmebehandlung 303
 - Ausrichtung an Speichergrenzen 355
 - auto 90, 408, 562
 - automatische Variable 124
 - make 521
 - Autotools (GNU) 616
- ## B
- back() 771, 773, 775, 778, 781, 843
 - back_inserter(), _insert_iterator 812
 - Backslash
 - Zeichenkonstante \ 52
 - Zeilenfortsetzung 130
 - Backspace 52
 - bad() 389
 - bad_alloc 308

- badbit 388
 - bad_cast 293, 308
 - bad_typeid 296, 308
 - base() 811
 - basefield 379
 - basic_-Streamklassen 375
 - basic_string 841
 - Basisklasse 257
 - und virtueller Destruktor 281
 - Konstruktor 287
 - Subobjekt 290
 - virtuelle 289
 - Bedingungsdruck 64
 - make 613
 - Bedingungsoperator ?: 67
 - beg 391
 - begin() 404, 766
 - Belegungsgrad 792
 - benutzerdefinierte
 - Datentypen 79, 88
 - Klassen 322
 - Typen (Ausgabe) 377
 - Typen (Eingabe) 735
 - Benutzungszählung 851
 - Bereichsnotation 767
 - Bereichsoperator :: 59, 152, 269
 - namespace 142
 - Bibliothek
 - C++- 742
 - C- 142, 873
 - Bibliotheksmodul 124
 - dynamisch 613
 - statisch 612
 - bidirectional_iterator 807
 - Bildschirmsteuerung mit ANSI-
 - Sequenzen 52
 - /bin/sh 522
 - Binärdatei 222
 - binäre Ein-/Ausgabe 220
 - binärer Operator 319
 - optimiert 596
 - binäres Prädikat 659, 693, 816
 - binäre Zahlendarstellung 46
 - binary 97, 390
 - binary_negate 753
 - binary_search() 681
 - bind 754
 - Binden 124
 - dynamisches *siehe* dynamisches Binden
 - statisches *siehe* statisches Binden
 - Bit
 - Operatoren 45, 46
 - Verschiebung 46
 - pro Zahl 42
 - Bitfelder 91
 - <bitset> 764
 - bitset 801
 - bitweises ODER, UND, XOR 45
 - Blitz (Matrix) 705
 - Block 31, 33, 58, 59, 62, 172
 - und dyn. Objekte 204
 - bool 54
 - boolalpha 55, 379, 384
 - Boost
 - create_directory() 730
 - exists() 728
 - Filesystem 727
 - is_any_of() 625
 - is_directory() 728
 - lexical_cast() 627
 - Matrix 705
 - remove_all() 728
 - split() 625
 - Boost.Asio 477
 - break 68, 77
 - bsearch() 878
 - Bubble-Sort 82
 - Bucket 792
 - bucket() 797, 800
 - Byte 51
 - Byte-Reihenfolge 485
- C**
- C++-Schlüsselwörter 887
 - C-Arrays 189
 - C-Funktionen einbinden 144
 - C-Header 873
 - C-Header-Datei 142
 - C-String 193
 - call wrapper 430
 - Callback-Funktion 225, 756

- calloc() 351
- capacity() 771, 843
- case 68
- <cassert> 133, 874
- cast *siehe* Typumwandlung
- catch 304
- cbegin() 766
- <cctype> 720, 874
- cdecl 228
- ceil() 876
- cend() 766
- cerr 94, 376
- <cerrno> 875
- <cfloat> 873
- char 51, 192
- char* 193
- char* const vs. const* char 189
- chmod 730
- <chrono> 760
- cin 34, 94, 376, 389
- <ciso646> 873
- class 151, 265
- class (bei Template-Parametern) 135
- clear() 389, 770, 785, 789, 843
- Client-Server
 - Beziehung 152
 - und callback 225
- <climits> 43, 874
- <locale> 874
- clock(), clock_t 882
- clog 376
- close() 97
- Closure 347
- CMake 619
- <cmath> 49, 695, 743, 873, 875
- code bloat 619
- Code::Blocks 37
- codecvt 832
- Code-Formatierer 640
- collate 633, 830
- combine() 823
- compare() 830, 846
- Compilationsmodell 620
- Compiler 32, 34, 123, 149, 151
 - befehle 891
 - direktiven 122, 128
 - und Templates 138
 - Typumwandlung 164
- <complex> 695
- Computerarithmetik 49
- configure 616
- conj() 696
- connect() 454
- const 51
 - Elementfunktionen 151, 170
 - globale Konstante 126
- const_cast<>() 294
- const char* vs. char* const 189
- const_iterator 765
- const_local_iterator 797, 800
- const_pointer 770
- constraint, Vererben von 283
- const_reference 765
- const_reverse_iterator 769, 811
- const& *siehe* Referenz auf const
- Container 398
 - implizite Datentypen 765
- Container-Methoden 766
- container_type 776, 777
- continue 77
- copy elision 589
- copy semantics 591
- copy() 717, 843
- copy_backward() 717
- copy_if() 718
- copy_n() 719
- cos(), cosh() 696, 863, 876
- count() 785, 797, 800, 803
 - Algorithmus 661
 - set 789
- count_if() 661
- cout 33, 94, 376
- __cplusplus 144
- crbegin(), crend() 769
- create_directory() (Boost.Filesystem) 730
- critical section 427
- CRLF 488
- <csetjmp> 874
- cshift() 860
- <csignal> 874
- <csdarg> 876
- <csddef> 47, 188, 877

`<cstdint>` 758
`<cstdio>` 728, 877
`<cstdlib>` 116, 224, 743, 875, 877
`c_str()` 97, 234, 843
`<cstring>` 194, 226, 233, 235, 879
`<ctime>` 335, 881
`ctype` 632, 634, 830, 831
`cur` 391
`curr_symbol` 834
`<cwctype>` 875
`CXXFLAGS` 521

D

dangling pointer 203
 data race 427
`data()` 771, 781, 843
 Datagramm 483
`__DATE__` 134
 Datei
 ASCII 221
 binär 222
 Ein-/Ausgabe 96, 375
 kopieren 97
 löschen 728
 Öffnungsarten 390
 öffnen 97
 Positionierung 391
 schließen 97
 umbenennen 729
 Dateizugriffsrechte 730
 Daten
 als Attributwerte 950
 static-Element- 242
 Datenbankanbindung 503
 Datenkapselung 884, 951
 Datensatz 88
 Datentypen 34, 41
 abstrakte *siehe* Abstrakter
 Datentyp
 benutzerdefinierte 79
 erster Klasse 299
 int und unsigned 66
 logische 54
 parametrisierte 134, 246
 polymorphe 279
 strukturierte 88

 zusammengesetzte 79
`date_order()` 837
 Datum
 gültiges 336
 Klasse 334
 regulärer Ausdruck 710
`dec` 379, 384
`decimal_point()` 833, 834
`default` 68
 default constructor 154
 Default-Parameter *siehe* vorgegebene
 Parameter
`= default` 182
`#define` 129, 130
 Definition 126, 951
 von static-Elementdaten 242
 von Objekten 155
 Deklaration 33, 34, 37, 126, 951
 einer Funktion 103
 Funktionszeiger 223
 Lesen einer D. 226
 in for-Schleifen 74
 Deklarationsanweisung 61
 Dekrementierung 45
 Dekrementoperator 336
 Delegation 299
 delegierender Konstruktor 182
`delete` 200, 203, 245, 280, 281
 überladen 347
`delete [ ]` 204, 885
`= delete` 182, 565
`<deque>` 764
`deque` 775
 Dereferenzierung 186, 224
 Design by Contract 315
 Destruktor 171, 288, 884
 implizite Deklaration 171
 und `exit()` 173
 virtueller 280, 886
`detach()` 425
 Dezimalpunkt 47, 833
 Dialog 465
`difference_type` 765
 Differenz
 Menge 686
 symmetrische Menge 687

difftime() 882
digits, digits10 725
distance() 810
Distribution 523, 951
div(), div_t 878
divides 753
Division durch 0 311
dll 613
DNS 475
do while 71
Dokumentation 541
domain_error 308
double 47, 192
 als Laufvariable? 75
 korrekter Vergleich 561
downcast 293, 367
doxygen 541
Dubletten entfernen 658
Durchschnitt (Menge) 686
dynamic_cast<>() 293
dynamisches Array 324
dynamisches Binden 223, 270, 951
dynamische Datenobjekte 200
dynamischer Typ 295

E

e, E 47
Eclipse 39
effizienter binärer Operator 596
egrep 409, 636
Ein- und Ausgabe 93
Einbinden von C-Funktionen 144
Eingabe 380
 Datei- 96, 375
 von Strings 95
 benutzerdefinierter Typen 735
Einschränkung *siehe* constraint
Elementdaten, Zeiger auf 231
Elementfunktion 148, 260
 als Funktionsobjekt 757
 Zeiger auf 230
Ellipse 254, 306, 876
else 63
emplace() 768
emplace_back() 771, 773, 775
emplace_front() 773, 775

empty() 766, 776, 778, 779, 843
end 391
end() 404, 766
#endif 130
endl 43, 49, 52, 384
ends 384
ENTER 32, 94
»enthält«-Beziehung 289
enum 79
Enumeration 79
Environment, env[] 208
EOF 382
eof() 305, 382, 389
eofbit 388
epsilon() 725
equal() 694
equalsIgnoreCase() 634
equal_range() 682, 785, 789, 797, 800
equal_to 753
erase() 770, 784, 788, 796, 799, 844
ereignisgesteuerte Programmierung 452
Ergebnisrückgabe 104
errno 302, 728, 875
Exception 950
 arithmetische Fehler 311
 und Destruktor 304
 Handling 303
 Hierarchie 307
 Speicherleck durch Exc. 567
<exception> 308, 309
exception 307
Exception-Sicherheit 315
exists() (Boost.Filesystem) 728
exit() 116, 878
 und Destruktor 173
Exklusiv-Oder (Menge) 687
exp() 696, 863, 876
explicit 161
explizite Instanziierung von
 Templates 621, 622
Exponent 47, 48
Expression Template 599
extern 124, 126, 128
extern "C" 144
extern template 621
external linkage 126

F

f, F 47
fabs() 696, 876
Facette 829
fail() 389, 390
failbit 388, 736
fakultaet() 102
Fallunterscheidung 67
false 54
falsename() 833
Fehlerbehandlung 301
 Ein- und Ausgabe 387
Fibonacci 654
__FILE__ 133
fill() 378, 648
fill_n() 648
find() 785, 797, 800
 Algorithmus 674
 set 789
 string 845
find_...-Methoden (string) 846
findstring 613
find_end() 678
find_first_of() 675
fixed 379, 380, 384
Fixture 534
flache Kopie 236
flags() 378
flip() 772, 803
float 47
float.h 47
floatfield 379
floor() 876
flush 384
flush() 380
fmod() 876
fmtflags 378
for 73
 Kurzform 767
foreach (make) 608
for_each() 716
Formalparameter 104
Formateinstellungen für Streams 379
Formatierung der Ausgabe 377
forward() 750
forward_iterator 807

forward_list 744
frac_digits() 834
Fragmentierung (Speicher) 205
Framework 453
free store 200
free() 350
frexp() 876
friend 241
front() 771, 773, 775, 777, 781, 843
front_inserter(), _insert_iterator 812
<fstream> 96
fstream 375, 392
Füllzeichen 378
function 756
<functional> 753
Funktion 102
 mit Gedächtnis 105
 klassenspezifische 242
 mathematische 49, 875
 Parameterübergabe
 per Referenz 111
 per Wert 107
 per Zeiger 205
 rein virtuelle 275
 mit Definition 276
 static 242
 Überschreiben in abgeleiteten
 Klassen 268
 Überschreiben virtueller
 Funktionen 273, 886
 variable Parameterzahl 113
 virtuelle *siehe* virtuelle
 Funktionen
 vorgegebene Parameterwerte
 113
Funktionsobjekte 344, 386
 function 756
 mem_fn 757
Funktionsspezifikation 121
Funktions-Template 134
Funktor *siehe* Funktionsobjekte

G
g++ 36
Ganzzahlen 42
garbage collection 205

- gegenseitige Abhängigkeit von
 - Klassen 180
 - Genauigkeit 48
 - Generalisierung 258
 - generate(), generate_n() 648
 - generische Programmierung 398
 - GET 489
 - get() 94, 221, 381, 382
 - getaddrinfo() 476
 - getenv() 878
 - getline() für Strings 95, 848
 - getline(char*, ...) 382
 - getloc() 395
 - get_monthname() 837
 - getnameinfo() 476
 - get_temporary_buffer() 855
 - get_time(), get_weekday(), get_year() 837
 - GGT 70
 - schnell 166
 - Gleichheitsoperator bei Vererbung 368
 - Gleichverteilung 714
 - Gleitkommazahl 51
 - Syntax 47
 - global 59
 - Namensraum 142
 - Variable 125, 128
 - gmtime() 882
 - GNU 952
 - GNU Autotools 616
 - good() 389
 - goodbit 388
 - Grafische Benutzungsschnittstellen 451
 - greater, greater_equal 753
 - greedy (regex-Auswertung) 412
 - Grenzwerte von Zahltypen 725
 - Groß- und Kleinschreibung 32, 37
 - größter gemeinsamer Teiler 70
 - grouping() 833, 834
 - gslice 866
 - gslice_array 869
 - GTK+ 451
 - guard 428
 - Gültigkeitsbereich 115
 - Block 58
 - Datei 125
 - Funktion 104
 - Klassen 151
 - und new 203
 - GUI 451
- ## H
- hängender Zeiger *siehe* Zeiger,
 - hängender
 - has_facet() 823
 - hash() 830
 - Hash-Funktion 791, 793
 - hasher 794, 799
 - has_infinity 725
 - »hat«-Beziehung 585
 - Header 34
 - C++-Standard 142
 - Header (Http) 488
 - Header-Datei 122, 123
 - Inhalt 127
 - Heap 688
 - hex 379, 384
 - Hexadezimalzahl 44
 - Host Byte Order 485
- ## I
- iconv 828
 - IDE 36
 - Identität von Objekten 150, 952
 - if 63
 - #ifdef, #ifndef 129
 - ifeq 613
 - ifstream 96, 375
 - ignore() 382
 - imag() 695, 696
 - Implementation 123
 - Implementationsdatei, Inhalt 127
 - Implementationsvererbung 297
 - implizite Deklaration
 - Destruktor 171
 - Konstruktor 154
 - Zuweisungsoperator 328, 365
 - in 97, 390
 - #include 32, 122, 128
 - includes() 684
 - Indexoperator 81, 190, 191, 212, 326
 - indirect_array 870
 - indirekte Adressierung 870

- `infinity()` 725
- Initialisierung
 - array 781
 - und virtuelle Basisklassen 291
 - C-Array 192, 210
 - Konstante in Objekten 156, 243
 - globaler Konstanten 126, 128
 - mit konstruktor-interner Liste 156, 243, 263
 - mit Liste (Container) 767
 - von Objekten 84, 154, 263
 - mit {}-Liste 155
 - von Referenzen 910
 - Reihenfolge 156, 235
 - in for-Schleife 74
 - von static-Elementdaten 242
 - struct 89
 - vector 84
 - und Vererbung 263
 - und Zuweisung 84, 158
- `initializer_list` 767
- Inklusions-Modell 620
- Inkrementierung 45
- Inkrementoperator 334
- `inline` 139, 152
- `inner_product()` 651
- `inplace_merge()` 673
- `input_iterator` 806
- `insert()` 770, 784, 788, 791, 796, 799, 844
- `insertion()` 813
- `insert_iterator` 812
- Installation der DVD-Software 937
- Instanz 30, 952
- Instanziierung von Templates 248
 - explizite 621, 622
 - ökonomische (bei vielen Dateien) 619
- `int` 33, 42
- `int-Parameter in Templates` 249
- `int2string()` 629
- integral promotion 58
- Integrierte Entwicklungsumgebungen 37
- Interface (UML) 580
- `internal` 379, 384
- `internal linkage` 126
- Internet-Anbindung 473
- Interrupt (Thread) 434
- Intervall 767
- `INT_MAX` 43
- Introspektion 453
- `invalid_argument` 308
- `IOException` 574
- `<iomanip>` 384, 385
- `<ios>` 384
- `ios` 375, 387, 390
- `ios_base` 375
- `ios_base::binary`, `ios_base::in` 97
- `ios_base-Fehlerstatusbits` 388
- `ios_base-Flags` 378, 379
- `ios_base-Formateinstellungen` 379
- `ios_base-Manipulatoren` 384
- `ios::failure` 389
- `ios-Flags zur Dateipositionierung` 391
- `ios-Methoden` 378, 380, 389
- `iostate` 387
- `<iostream>` 34, 376, 377, 381, 384
- `iota()` 649
- IP-Adresse (regulärer Ausdruck) 712
- IPv4, IPv6 475
- `is()` 831
- `isalnum()`, `isalpha()` 829, 875
- `is_any_of()` 625
- `isblank()` 875
- `is_bounded` 725
- `isctrl()` 829, 875
- `isdigit()` 120, 160, 829, 875
- `is_directory()` (Boost.Filesystem) 728
- `is_exact` 725
- `isgraph()` 829, 875
- `is_heap()`, `is_heap_until()` 692
- `is_iec559`, `is_integer` 725
- `islower()` 829, 875
- `is_modulo` 725
- ISO 10646 826
- ISO-8859-1, ISO-8859-15 825
- `isprint()` 829, 875
- `ispunct()` 829
- `is_signed` 725
- `isspace()` 829, 875
- ist-ein*-Beziehung 257, 267, 282

istream 375, 380
IStream-Iterator 813
istream::seekg(), tellg() 391
istream::ws 384
istreamstream 375, 393
isupper() 829, 875
isxdigit() 829, 875
Iterator 399, 403, 805
 Adapter 811
 Bidirectional 807
 Forward 807
 Input 806
 Insert 812
 Output 806
 Random Access 807
 Reverse 811
 Stream 813
 Zustand 404
<iterator> 805
iterator 765
iterator_category 806
Iterator-Tags 807
iter_swap() 719

J

Jahr 881
join() 422, 425
Jota 649

K

Kardinalität 582
Kategorie (locale) 829
key_comp() 785, 789
key_compare 783, 788
key_equal 794, 799
key_type 783, 788, 794, 799
Klammerregeln 56
Klasse 29, 149, 952
 abgeleitete *siehe* abgeleitete Klasse
 abstrakte 275
 Basis- *siehe* Basisklasse
 Deklaration 151
 konkrete 275
 Ober- *siehe* Oberklasse
 für einen Ort 150

Unter- *siehe* Unterklasse
 für rationale Zahlen 162
Klassenname 296
klassenspezifische
 Daten 242
 Funktionen 242
 Konstante 245
Klassen-Template 246
Klassifikation 258, 952
Kleinschreibung 32
Kollisionsbehandlung 792
Kommandointerpreter 519, 522
Kommandozeilenparameter 208
Kommaoperator 76, 210
Kommentar 31
komplexe Zahlen 695
Komplexität 957
Komposition 585
konkrete Klasse 275
Konsole auf UTF-8 einstellen 825
Konstante 50
 globale 126, 128
 klassenspezifische 245
konstante Objekte 170, 886
Konstruktor 151, 154
 allgemeiner *siehe* allgemeiner Konstruktor
 implizite Deklaration 154
 delegierender 182
 Kopier- *siehe* Kopierkonstruktor
 vorgegebene
 Parameterwerte 155
 Typumwandlungs- *siehe* Typumwandlungskonstruktor
Kontrollabstraktion 399
Kontrollstrukturen 61
Konvertieren von Datentypen *siehe* Typumwandlung
Kopie, flache/tiefe 236
Kopieren
 von Dateien 97
 von Objekten 236
 von Zeichenketten 198
Kopierkonstruktor 158, 326, 884
 Vermeidung des Aufrufs 159
Kreuzreferenzliste 645

kritischer Bereich 427

Kurzform-Operatoren 44

L

l, L 47

L-Wert 62, 190, 195, 326, 953

Länge eines Vektors 652

Lambda-Funktionen 346

LANG 822

late binding 223

Laufvariable 74

Laufzeit 186

 und Funktionszeiger 223

 und new 200

 und Polymorphie 270

 Typinformation 295

ldd 614

ldexp() 876

ldiv(), ldiv_t 878

LD_LIBRARY_PATH 615

left 379, 384

length() 842

length_error 308

lexical_cast() 627

lexicographical_compare() 665

lexikografischer Vergleich 751, 767, 953

<limits> 47, 725

__LINE__ 133

Linken 125, 142

 dynamisches 613, 953

 internes, externes 126

 statisches 612, 953

Linker 36

linksassoziativ 56

list 772

<list> 745, 764

Liste

 Initialisierungs- 156, 243

 Initialisierungs- (bei C-Arrays)
 210

Liste (Klasse) 404

Literal 194

 Zahlen- 51

 Zeichen- 51

load factor 792

load_factor() 797, 800

<locale> 821

localtime() 335, 882

local_iterator 794, 797, 799, 800

lock_guard 428

log(), log10() 696, 863, 876

logical_and, _not, _or 753

logic_error 308

logischer Datentyp 54

logische Fehler 309

logische Negation 55

logisches UND bzw. ODER 55

lokal (Block) 58

lokale Objekte 189

long 42

long double 47

lower_bound() 682, 785, 789

lvalue *siehe* L-Wert

M

main() 31, 33, 115

MAKE 607

make 517

 automatische Ermittlung von

 Abhängigkeiten 602

 parallelisieren 611

 rekursiv 606

 Variable 521

Makefile 124, 519

make_heap() 691

make_pair() 751

make_tupel() 752

Makro 130

malloc() 350

Manipulatoren 383

Mantisse 48

<map> 764

map 782

mapped_type 783, 794

mask_array 869

match_results 415

mathematischer Ausdruck 49

mathematische Funktionen 875

Matrix

 C-Array 209

 C-Array, dynamisch 213

 Klasse 215

Klasse, mit zusammenhängendem
 Speicher 697
 Vektor von Vektoren 359
max() 725, 726, 860
max_bucket_count() 797, 800
max_element() 655
max_exponent, max_exponent10 725
max_load_factor() 797, 800
max_size() 766
mehrdimensionales Array 209
mehrdimensionale Matrizen 359
Mehrfachvererbung 259, 285, 288
MeinString (Klasse) 233
member function *siehe* Element-
 funktion
memchr() 880
memcmp(), memcpy(), memmove(), 880
<memory> 745, 855
memory leak 204, 567
memset() 881
mem_fn() 757
Mengenoperationen auf sortierten
 Strukturen 684
merge() 671, 774
mergesort() 672
messages 830, 839
Meta-Objekt-Compiler 454
Methode 30, 148, 954
 Regeln zur Konstruktion
 von Prototypen 557
MIME 954
min() 725, 726, 860
min_element() 655
min_exponent, min_exponent10 725
MinGW 518
minmax() 726
minmax_element() 655
minus 753
Minute 881
mischen 671
mismatch() 692
mkdir() 730
mktime() 882
modf() 876
modulare Gestaltung 122
Modulo 45

modulus 753
Monat 881
monetary 830, 833
money_get 834
moneypunct 834
money_put 836
Monitor 439
move semantics 591
move() 596, 748
moving constructor 594
M_PI 344, 695, 734
multimap 787
multiplies 753
Multiplikationsoperator 332
Multiplizität (UML) 582
multiset 791
mutable 170

N

Nachbedingung 121, 954
Nachkommastellen 47
 precision 734
Name 37
 einer Klasse 296
name() 296, 823
Namenskonflikte bei Mehrfach-
 vererbung 288
Namenskonventionen 36
Nameserver 475
Namespace
 anonym 125
 in Header-Dateien 133
 Verzeichnisstruktur 605
namespace 32, 141
namespace std 60, 142
narrow() 832
nationale Sprachumgebung 822
NDEBUG 133, 309
negate 753
Negation
 bitweise 45, 46
 logische 55
Negationsoperator (überladen) 390
negative_sign() 834
neg_format() 834
Network Byte Order 485

Netzwerkprogrammierung 473
 neue Zeile 49, 52, 63
 new 200, 203, 245
 Fehlerbehandlung 312
 überladen 347
 <new> 308, 854
 new Placement-Form 854
 new_handler 312
 next_permutation() 664
 noboolalpha 384
 noexcept 306
 none() 803
 norm() 695, 696
 Normalverteilung 715
 noshowbase, -point, -pos 384
 noskipws 384
 not1, not2 753
 Notation für Intervalle 767
 not_equal_to 753
 nothrow 314
 notify_one(), _all() 433, 576
 noutitbuf, nouppercase 384
 npos 842
 NRVO *siehe* RVO
 nth_element() 669
 NTP – Network Time Protocol 485
 NULL 188, 877
 und new 314
 nullptr 188
 numeric 830, 832
 numeric_limits 47, 725
 numerische Auslöschung 49
 numerische Umwandlung 625, 629, 847
 num_get, num_put 832
 NummeriertesObjekt
 Klasse 242
 numpunct 833

O

O-Notation 957
 Oberklasse 257, 268, 954
 erben von 259
 Subobjekt einer 263
 Subtyp einer 266
 Zugriffsrechte vererben 264
 Oberklassenkonstruktor 260, 263

object slicing 267
 Objekt 28, 149, 151, 955
 dynamisches 200
 als Funktions- 344
 Identität *siehe* Identität von
 Objekten
 Initialisierung 154, 263
 konstantes 170, 886
 temporäres 161
 Übergabe per Wert 159
 verkettete Objekte 202
 verwitwetes 204
 vollständiges 290, 291, 957
 Objektcode 36
 Objekterzeugung 151
 Objekthierarchie 289
 Objektorientierung 147
 oct 379, 384
 ODER
 bitweises 45
 logisches 55
 Öffnungsarten für Streams 390
 offsetof 877
 ofstream 96, 375
 Oktalzahl 44
 omanip 385
 one definition rule 127, 884
 open() 96
 OpenMP 578
 Open Source 955
 Operator
 arithmetischer 45
 binärer 319
 Bit- 45
 für char 54
 als Funktion 318
 Kurzform 46
 für logische Datentypen 55
 Präzedenz 56, 890
 relationale 45, 55, 747
 Syntax 318
 Typumwandlungs- 337
 überladen (<<) 377
 überladen (>>) 735
 unärer 319
 für ganze Zahlen 44

- operator!() 390
 - operator delete() 347
 - operator new() 347
 - operator string() 338
 - operator()() 344
 - operator*() 333, 339, 403
 - operator*=() 332
 - operator++() 334, 335, 403
 - operator++(int) 563
 - operator+=() 321
 - operator->() 339
 - operator|() 801
 - operator|=() 802
 - operator!==() 403, 803
 - operator<<() 322, 377, 801, 802
 - operator<=() 802
 - operator=() 329
 - operator==() 403, 803
 - virtual 368
 - operator>>() 381, 801, 803
 - operator>=() 802
 - operator[]() 362, 365, 771, 775, 781, 807
 - operator&() 801
 - operator&=() 802
 - operator^=() 802
 - operator^() 801
 - operator~() 803
 - Optimierung durch Vermeiden
 - temporärer Objekte 159
 - Ort (Klasse) 150
 - ostream 322, 375, 377
 - OStream-Iterator 813
 - ostream::endl, ends, flush() 384
 - ostream::seekp(), tellp() 391
 - ostreamstream 375, 393
 - out 390
 - out_of_range 308
 - output_iterator 806
 - overflow 44, 49
 - overflow_error 308
- P**
- pair 750
 - Parameter einer Funktion 103
 - Parameterübergabe
 - per Referenz 111
 - per Wert 107
 - per Zeiger 205
 - parametrisierte Datentypen 134, 246
 - »part-of«-Beziehung 585
 - partial_sort(), _copy 669
 - partial_sum() 653
 - partielle Spezialisierung von
 - Templates 806
 - partition() 666
 - patsubst 523
 - peek() 382
 - perfect forwarding 750
 - Performance 587
 - Permutationen 663
 - PHONY 520
 - π 51, 695
 - Placement new/delete 854
 - plus 753
 - POD (plain old data type) 955
 - pointer 770, 783, 788, 794, 799
 - Pointer, smarte 339
 - polar() 696
 - polymorpher Typ 279, 295
 - polymorphe Zuweisung 366
 - Polymorphismus 270, 955
 - pop() 776, 778, 779
 - pop_back() 771, 773, 775
 - pop_front() 773, 775
 - pop_heap() 689
 - portabel 827
 - pos_format() 834
 - Positionierung innerhalb einer Datei 391
 - positive_sign() 834
 - POSIX 822
 - POST 494
 - postcondition *siehe* Nachbedingung
 - Postfix-Operator 334
 - pos_type 391
 - pow() 696, 863
 - Prädikat
 - Algorithmus mit P. 816
 - binäres 659, 693, 816
 - unäres 661
 - Präfix-Operator 334
 - Präprozessor 122, 128, 602
 - Präzedenz von Operatoren 56, 890

precision() 380
precondition *siehe* Vorbedingung
prev_permutation() 663
PRINT (Makro) 132
printf() 253
Priority-Queue 778
private 151, 264
private Vererbung 297
Programm
 ausführbares 36
 Strukturierung 101
Programmierrichtlinien 884
Projekte 124
protected 264
protected-Vererbung 299
Prototyp
 Funktions- 102
 einer Methode 150
 Regeln zur Konstruktion 557
ptrdiff_t 877
public 151, 260, 264
push() 776, 778, 779
push_back() 771, 773, 775
 vector 85
push_front() 773, 775
push_heap() 690
put() 96, 221, 377
putback() 382

Q

qsort() 224, 878
Qt 453
 cout 455
QThread 469
Quantifizierer 412
Queue 777
<queue> 745, 764, 777
quicksort() 135

R

R-Wert 62, 190, 953
 Referenz 591
race condition 427, 448
radix 725
RAII 396, 428, 534, 567, 955

rand() 712, 878
Random 712
random_access_iterator 807
random_shuffle() 657
RAND_MAX 878
range_error 308
<ratio> 758
rationale Zahlen (Klasse) 162
raw_storage_iterator 855
rbegin() 769, 811, 842
rdstate() 389
read() 220
real() 695, 696
realloc() 351
Rechengenauigkeit 48, 75
rechtsassoziativ 56
reelle Zahlen 47
reentrant 448
ref(), reference_wrapper 430, 761
reference
 bitset 801
 Container 765
 vector<bool> 772
reference counting 851
Referenz 55, 119
 auf Basisklasse 273
 auf const 111, 158
 auf istream 381, 735
 auf Oberklasse 266, 272
 auf ostream 322, 377
 Parameterübergabe per 111
 Rückgabe per 327
 auf R-Wert 591
 oder Zeiger? 885
Referenzsemantik 237, 587
<regex> 308, 416
regex_iterator 415
regex_match() 417
regex_replace() 418, 638
regex_search() 417, 636
reguläre Ausdrücke 409
Reihenfolge
 Auswertungs- 56
 Berechnungs- 49
 der Initialisierung 156, 235
 umdrehen 660

- rein virtuelle Funktion 275
 - mit Definition 276
- reinterpret_cast<>() 192, 220, 295
- Rekursion 108
 - Template-Metaprogrammierung 252, 255
- rekursiver Abstieg 116, 117
- rekursiver Make-Aufruf 606
- relationale Operatoren 45, 55, 747
- rel_ops 747
- remove()
 - Algorithmus 723
 - Datei/Verzeichnis (C) 728
 - Liste 773
- remove_all() (Boost.Filesystem) 728
- remove_if() 723, 773
- rename() Datei/Verzeichnis (C) 729
- rend() 769, 811, 842
- replace() 722, 845
- replace_copy(), replace_if(), replace_copy_if() 722
- reserve() 239, 771, 842
- reset() 803
- resetiosflags() 385
- resize() 771, 773, 776, 842
- return 34, 159
- return value optimization (RVO) 589
- return_temporary_buffer() 855
- reverse() (list) 773
- reverse(), _copy() 660
- Reverse-Iterator 811
- reverse_iterator 769
- Reversible Container 768
- rfind() 845
- right 379, 384
- rotate(), rotate_copy() 656
- round_error(), _style() 725
- RTTI 295
- runtime_error 308
- rvalue *siehe* R-Wert
- RVO 589
- S**
- scan_is(), scan_not() 831
- Schleifen 69
 - do while 71
 - for 73
 - und Strings 196
- Tabellensuche 84
- terminierung 71
- while 69
- Schlüsselwörter 887
- Schnittmenge 686
- Schnittstelle 123, 956
 - einer Funktion 106
- Regeln zur Konstruktion 557
- scientific 379, 380, 384
- scope 59
- scoped locking 428
- search() 677
- search_n() 680
- sed 610
- seekg(), seekp() 391
- Seiteneffekt 64, 103, 197, 199
 - im Makro 134
- Seitenvorschub 52
- Sekunde 881
- Selektion 63
- sentinel 84, 192
- sentry 396
- Sequenz 63
- Sequenz-Methoden 770
- Server-Client
 - Beziehung 152
 - und callback 225
- <set> 764
- set 787
- set() 803
- setbase() 385
- set_difference() 686
- setf() 378, 379
- setfill() 385
- set_intersection() 686
- setiosflags() 385
- setjmp() 874
- setprecision() 385
- setstate() 389
- set_symmetric_difference() 687
- set_terminate() 309
- set_union() 685
- set_new_handler() 313

- setw() 385
- shared_ptr 344, 851
 - für Arrays 567
- SHELL 522
- shift() 860
- short 42
- showbase, showpoint, showpos 379, 384
- showSequence() 648
- shrink_to_fit() 239, 842
- Sichtbarkeit 58
 - dateiübergreifend 124
- Sichtbarkeitsbereich (namespace) 141
- Signal 454, 457
- Signalton 52
- Signatur 114, 260, 269, 270, 273
- signed char 51
- sin(), sinh() 696, 863, 876
- single entry/ single exit 77
- Singleton 705
- size() 81, 86, 766
- sizeof 191
 - nicht bei dynamischen Arrays 214
- size_t 47, 877
- size_type 765
- Skalarprodukt 651
- skipws 379, 384
- sleep() 424
- sleep_for(), sleep_until() 760
- slice 864
- slice_array 866
- Slot 454, 457
- Smart Pointer 339
 - und Exceptions 567
- Socket 478
- Sommerzeit 881
- Sonderzeichen 53
- sort() 667
- Sortieren 667
 - stabiles 667
 - durch Verschmelzen 672
- Sortierung 224
- sort_heap() 691
- Speicherklasse 124
- Speicherleck 204, 567
- Speicherplatzfreigabe 188
- Speicherverwaltung (eigene) 355
- Spezialisierung
 - von Klassen 258
 - von Templates 137
- Spezifikation einer Funktion 121
- splice() 774
- split() 624
- Sprachumgebung 822
- SQL 503
- sqrt() 50, 696, 863, 876
- srand() 712, 878
- SSH 547
- <sstream> 393
- stable_partition() 666
- stable_sort() 667
- Stack 58
 - Klasse 246
- <stack> 745, 764
- stack 776
- stack unwinding 304
- Standard Ein-/Ausgabe 94
- Standardbibliothek
 - C 142, 873
 - C++ 742
- Standardheader 143, 745
- Standardklassen 745
- Standard-Typumwandlung 57
 - Zeiger 229
- start() 864, 867
- static 124, 125
 - Attribute und Methoden 242
 - in Funktion 105
 - Makefile 612
- static_assert 134
- static_cast<>() 53, 292
- statisches Binden 270, 956
- Statusabfrage einer Datei 389
- std 32, 60, 142
- <stdexcept> 308
- stdio 379
- Stelligkeit (Template) 253
- STL 397
- stod() 626
- stoi() 625, 626
- stoi() und verwandte numerische Konversionsfunktionen 847
- strcat(), strchr(), strcmp() 879

- `strncat()`, `strncmp()` 880
- `strpbrk()`, `strchr()`, `strstr()` 880
- `strcpy()` 199, 879
- `strcspn()` 879
- `stream` 375
- Stream-Iterator 813
- Stream-Öffnungsarten 390
- `strerror()` 728, 875, 879
- Streuspeicherung 791
- `strftime()` 882
- `stride()` 864, 867
- String 86
- `string` 86, 841
 - `append()` 843
 - `assign()` 844
 - `at()` 86, 843
 - `back()` 843
 - `begin()` 842
 - `capacity()` 843
 - `clear()` 843
 - `compare()` 846
 - `copy()` 843
 - `c_str()` 843
 - `data()` 843
 - `empty()` 843
 - `end()` 842
 - `erase()` 844
 - `find()` 845
 - `find_...-Methoden` 846
 - `front()` 843
 - `insert()` 844
 - `length()` 86, 842
 - `operator+=()` 843
 - `rbegin()`, `rend()` 842
 - `replace()` 845
 - `reserve()` 842
 - `resize()` 842
 - `rfind()` 845
 - `shrink_to_fit()` 842
 - `size()` 86, 842
 - `substr()` 846
 - `swap()` 843
 - `verketten` 86
- `<string>` 745, 841
- String in Zahl umwandeln 625
- string.h* 194
- String-Klasse `MeinString` 233
- Stringlänge 196, 197
- Stringliteral 826
- `strlen()` 194, 197, 879
- `strncpy()` 575, 880
- `strtod()`, `strtol()`, `strtoul()` 627, 878
- `strtok()` 643, 880
- `struct` 88, 265
- Strukturanalyse 545
- Stunde 881
- Subobjekt 260, 266, 287
 - in virtuellen Basisklassen 289
 - verschiedene 288
- Substitutionsprinzip 282
- `substr()` 846
- Subtyp 260, 266, 282, 956
- Subversion 546
- Suffix 47
- `sum()` 860
- `swap()` 329, 766, 843
 - Algorithmus 719
- `swap-Trick` 219, 573
- `swap_ranges()` 720
- `switch` 67
- symmetrische Differenz
 - sortierter Strukturen 687
- Synchronisation 426
- Syntaxdiagramm 117, 119
 - `?:` Bedingungsoperator 67
 - do while-Schleife 72
 - enum-Deklaration 79
 - for-Schleife 73
 - Funktionsaufruf 104
 - Funktionsdefinition 103
 - Funktionsprototyp 103
 - Funktions-Template 135
 - if-Anweisung 63
 - mathematischer Ausdruck 117
 - operator-Deklaration 318
 - struct-Definition 88
 - switch-Anweisung 68
 - Typumwandlungsoperator 338
 - while-Schleife 70
- `system()` 878
- `system_error` 308, 389

T

- Tabellensuche 84
- Tabulator 52
- Tag 881
- tan(), tanh() 696, 863, 876
- target (make) 519
- Taschenrechnersimulation 116
- Tastaturabfrage 94
- TCP 474
- »Teil-Ganzes«-Beziehung 585
- tellg(), tellp() 391
- Template
 - für Funktionen 134
 - #include 138
 - Instanziierung von T. 248
 - explizite 621, 622
 - ökonomische (bei vielen Dateien) 619
 - int-Parameter 249
 - für Klassen 246
 - Spezialisierung 137
 - partielle 806
 - static Attribut 357
 - variable Parameterzahl 253, 702
- Template-Metaprogrammierung 251
- temporäres Objekt 161
 - Vermeidung 159
- terminate() 309
- terminate_handler 309
- Test Driven Development 527
- test() 803
- Test-Suite 529
- Textersetzung 130
- this 209
- this->, *this bei Zugriff auf Oberklassenelement 333
- this_thread 422
- thousands_sep() 833, 834
- Thread 419
 - genau einer pro Objekt 434
- thread (API) 423
- Thread-Sicherheit 448
- thread_group 426, 428
- throw 304
- tie() 395
- tiefe Kopie 236
- __TIME__ 134
- time 830, 836
- time() 335, 882
- time_get 836
- time_put 838
- time_t 335, 881
- tm 335, 881
- tolower() 632, 829, 831, 874
- top() 776, 779
- to_string() 803
- to_ulong() 803
- toupper() 632, 827, 829, 831, 874
- to_string() 847
- traits 805
- transform() 720, 830
- tree (Programm) 604, 732
- Trennung von Schnittstellen und Implementation 124, 884
- true 54
- trunename() 833
- trunc 390
- try 304
- Tupel, <tuple> 752
- tuple_cat() 703
- Typ 956
 - polymorpher bzw. dynamischer Typ 295
- type cast *siehe* Typumwandlung
- <type_traits> 253
- typedef 227
- typeid() 295, 371
- type_info 295
- <typeinfo> 308
- typename (bei Template-Parametern) 135
- Typinformation 280
 - zur Laufzeit 295
- Typumwandlung
 - cast 53, 188, 224, 292
 - cast-Schreibweise 53
 - durch Compiler 164
 - const_cast<>() 294
 - dynamic_cast<>() 293
 - mit explicit 161
 - implizit 67, 161
 - mit Informationsverlust 115
 - reinterpret_cast<>() 295

- Standard- 57
 - Zeiger 229
- static_cast<>() 292
- Typumwandlungskonstruktor 160, 164, 319
- Typumwandlungsoperator 337
 - ios 390
- U**
- UDP 474, 483
- Überladen
 - von Funktionen 114
 - von Operatoren *siehe* operator
- Überlauf 44, 49
- Überschreiben
 - von Funktionen in
 - abgeleiteten Klassen 268
 - virtueller Funktionen 273, 886
- Übersetzung 122, 124
- Übersetzungseinheit 126
- Umgebungsvariable 208
- UML 257, 579
- Umleitung der Ausgabe auf Strings 393
- unärer Operator 319
- unäres Prädikat 661
- unary_negate 753
- UND
 - bitweises 45, 46
 - logisches 55
- #undef 130
- undefined behaviour 205
- underflow 49
- underflow_error 308
- Unicode 375, 825
- uninitialized_copy() 856
- uninitialized_fill(), _n() 856
- union 91
- unique() 658, 774
- unique_ptr 849
 - für Arrays 568
- unique_copy() 658
- unique_lock 428
- Unit-Test 525
- unitbuf 379, 380, 384
- unordered_map 793
- unordered_multimap 798

- unordered_multiset 801
- unordered_set 798
- unsigned 42
- unsigned char 51
- Unterklasse 257, 957
- upper_bound() 682, 785, 789
- uppercase 379, 384
- URI, URL 474, 638
- URL-Codierung 488
- use_facet() 632–634, 823
- using
 - Deklaration 298
 - Klassen 296
 - Namespace
 - Deklaration 142
 - Direktive 141
- UTC 882
- UTF-8 825
- <utility> 598, 748, 751

V

- Valarray
 - arithmetische Operatoren 861
 - Bit-Operatoren 861
 - logische Operatoren 862
 - mathematische Funktionen 863
 - relationale Operatoren 862
- <valarray> 857
- value_comp() 785, 789
- value_compare 783, 788
- value_type 765, 783, 788, 794
- Variable 34
 - automatische 124
 - globale 125, 128
 - make 521
- Variablenname 36
- variadic templates 253, 702
- vector
 - at() 82
 - push_back() 85
 - size() 81
- <vector> 745, 764
- vector 770
- vector<bool> 771
- Vektor 81
 - Klasse 323

- Länge (geom.) 652
- Verbundanweisung 62
- Vereinigung (Menge) 685
- Vererbung 957
 - der abstrakt-Eigenschaft einer Klasse 275
 - von constraints 283
 - der Implementierung 298
 - Mehrfach- 285
 - private 297
 - protected 299
 - von Zugriffsrechten 264
 - und Zuweisungsoperator 365
- Vergleich von double-Werten 561
- verschmelzen (*merge*) 671
- Versionskontrolle 546
- Vertrag 283, 957
- verwitwetes Objekt 204
- Verzeichnis
 - anlegen 730
 - anzeigen 731
 - löschen 728
 - umbenennen 729
- Verzeichnisbaum
 - anzeigen 732
 - make 605
- Verzweigung 63
- virtual 270, 272, 281
- virtuelle Basisklasse 289
- virtueller Destruktor 280
- virtuelle Funktionen 270, 273
 - rein- 275
- void 188
 - als Funktionstyp 103
- void* 224
 - Typumwandlung nach 188
 - Typumwandlungsoperator 390
- vollständiges Objekt 290, 291, 957
- Vorbedingung 121, 957
- vorgegebene Parameterwerte
 - in Funktionen 113
 - in Konstruktoren 155
- Vorkommastellen 47
- Vorrangregeln 56, 890
- Vorwärtsdeklaration 180

W

- Wächter (Tabellenende) 84, 192
- Wahrheitswert 54
- wait() 432, 440
- Warteschlange 777
- wchar_t 51, 826, 877
- weak_ptr 853
- Webserver 494
- Weite der Ausgabe 378
- Wert
 - eines Attributs 950
 - Parameterübergabe per 107
- Wertebereich 43
- Wertsemantik 237, 399, 587
 - Performanceproblem 589
- what() 308
- while 69, 196, 198
- whitespace *siehe* Zwischenraumzeichen
- wide character 51
- widen() 832
- Widget 457
- width() 378
- Wiederverwendung
 - durch Delegation 299
 - durch Vererbung 267
- wildcard 523
- Winterzeit 881
- Wochentag 881
- wofstream 828
- Worttrennung 32
- Wrapperklasse für Iterator 811
- write() 220, 377
- ws 384
- wstring 828, 841

X

- XOR, bitweises 45

Y

- yield() 424

Z

- Zahl in String umwandeln 629
- Zahlenbereich 42, 48
- Zeichen 51

- Zeichenkette 34, *siehe auch* string
 - C-String 193
 - Kopieren einer 197
- Zeichenklasse 412
- Zeichenkonstante 51
- Zeichenliteral 826
- Zeichensatz 825
- Zeiger 185, 200
 - vs. Array 190
 - auf Basisklasse 273, 280
 - auf Elementdaten 231
 - auf Elementfunktionen 230
 - auf Funktionen 223
 - hängender 203
 - intelligente *siehe* Smart Pointer
 - in Klassen 884
 - Null-Zeiger 188
 - auf Oberklasse 266, 272
 - auf ein Objekt (Mehrfachvererbung) 288
 - auf lokale Objekte 189
 - Parameterübergabe per 205
 - oder Referenz? 885
- Zeigerarithmetik 192
- Zeigerdarstellung
 - von [] 191
 - von [] [] 212
- Zeile
 - einlesen *siehe* `getLine()`
 - neue 52
- Zeit-Server 487
- Zeitkomplexität 957
- Zerlegung 666
- Ziel (make) 519
- Ziffernzeichen 51
- Zufallszahlen 712
- Zugriffsspezifizierer und -rechte 264
- zusammengesetzte Datentypen 79
- Zusicherung 133, 309
- Zustand 958
 - eines Iterators 404
- Zuweisung 34, 62, 66, 958
 - in abgeleiteten Klassen 266
 - und Initialisierung 84, 158
- Zuweisungsoperator 158, 328, 884
 - implizite Deklaration 328, 365
 - und Vererbung 365
- zweidimensionale Matrix 697
- Zweierkomplement 42
- Zwischenraumzeichen 94, 195, 381

Alles über Perl – und noch viel mehr



Plate

Der Perl-Programmierer

Perl lernen - Professionell anwenden -
Lösungen nutzen

1.232 Seiten.

ISBN 978-3-446-41688-8

Dieses Programmierhandbuch begleitet Sie von den ersten Schritten mit Perl bis hin zu Spezialthemen und der professionellen Anwendung von Perl in der täglichen Arbeit. In Teil 1 geht's los mit einem fundierten Einstieg in die Grundlagen und Konzepte von Perl einschließlich regulärer Ausdrücke, Systemschnittstelle, Debugging und Dokumentation. Teil 2 beschäftigt sich mit der Strukturierung von komplexeren Programmieraufgaben mithilfe von Packages und Modulen, um darauf aufbauend die objektorientierte Programmierung mit Perl zu behandeln. Teil 3 verschafft Ihnen Einblick in Spezialthemen und praktische Methoden wie z.B. die Berechnung von Datum und Uhrzeit, Grafik und Bildbearbeitung, Benutzeroberflächen und Datenbankanbindung, Entwicklung von Web-Anwendungen und die Netzwerk-Programmierung, Codegenerierung, Anbindung von LaTeX, automatisches Erzeugen von PDF-Dokumenten und Excel-Dateien, Hardwareansteuerung und vieles mehr.

Mehr Informationen zu diesem Buch und zu unserem Programm
unter www.hanser.de/computer

So macht programmieren lernen Spaß!



Warren D. Sande, Carter Sande

Hello World!

Programmieren für Kids und
andere Anfänger

452 Seiten. Mit CD.

ISBN 978-3-446-42144-8

Dein Computer wird nicht antworten, wenn du ihn anschreist, warum also nicht in seiner Sprache mit ihm sprechen? Wenn du programmieren lernst, kannst du das tun. Dann kannst du wirklich coole Sachen machen und sogar selbst Spiele programmieren. Und: Programmieren macht Spaß!

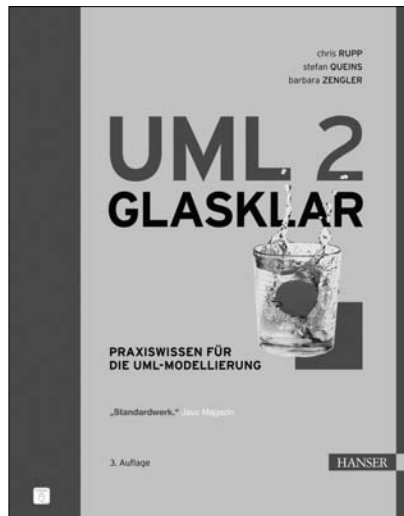
Hello World! ist eine wunderbar geschriebene Einführung in die Welt des Programmierens. Mit lustigen Beispielen macht es Programmier-Konzepte wie Speicher, Schleifen, Input und Output, Daten und Grafiken lebendig. Es ist so geschrieben, dass Kinder folgen können, aber jeder, der programmieren lernen will, kann es nutzen – auch Erwachsene! Du musst nichts über das Programmieren wissen, um das Buch zu nutzen.

Wenn du ein Programm starten und eine Datei speichern kannst, dann wirst du mit diesem Buch keine Probleme haben und in null Komma nichts programmieren können.

Mehr Informationen zu diesem Buch und zu unserem Programm
unter www.hanser.de/computer

Glasklar: Das „Standardwerk“!

Java SPEKTRUM



Rupp/Queins/Zengler

UML 2 glasklar

568 Seiten.

ISBN 978-3-446-41118-0

Die UML 2.0 ist erwachsen und in der Version 2.1 nun auch tageslichttauglich. Daher haben die Autoren diesen Bestseller in Sachen UML aktualisiert. Dieses topaktuelle und nützliche Nachschlagewerk enthält zahlreiche Tipps und Tricks zum Einsatz der UML in der Praxis. Die Autoren beschreiben alle Diagramme der UML und zeigen ihren Einsatz anhand eines durchgängigen Praxisbeispiels. Folgende Fragen werden u.a. beantwortet

- Welche Diagramme gibt es in der UML 2?
- Wofür werden diese Diagramme in Projekten verwendet?
- Wie kann ich die UML an meine Projektbedürfnisse anpassen?
- Was benötige ich wirklich von der UML?

Mehr Informationen zu diesem Buch und zu unserem Programm unter www.hanser.de/computer

ALLES ÜBER C++ – UND NOCH VIEL MEHR //

- Topaktuell: Entspricht dem neuen ISO-C++-Standard
- Ein Praxisbuch für alle Ansprüche - mehr brauchen Einsteiger und Profis nicht
- Stellt Grundlagen und fortgeschrittene Themen der C++-Programmierung vor und zeigt, welche Unterstützung professionelle Softwareentwickler in der Teamarbeit brauchen
- Enthält über 150 praktische Lösungen für typische Aufgabenstellungen und 85 Übungsaufgaben - natürlich mit Musterlösungen
- Auf DVD: Entwicklungsumgebung und GNU-Compiler für Windows und Linux, weitere Open Source-Software, u.a. Boost und Qt, alle Beispiele und Musterlösungen

DER C++-PROGRAMMIERER // Egal ob Sie C++ lernen wollen oder Ihre Kenntnisse in der Softwareentwicklung mit C++ vertiefen, in diesem Buch finden Sie, was Sie brauchen.

C++-Neulinge erhalten eine motivierende Einführung in die Sprache C++. Die vielen Beispiele sind leicht nachzuvollziehen. Klassen und Objekte, Templates, STL und Exceptions sind bald keine Fremdwörter mehr für Sie. Als Profi finden Sie in diesem Buch kurze Einführungen zu Themen wie Thread-Programmierung, Netzwerk-Programmierung mit Sockets und grafische Benutzungsoberflächen. Durch den Einsatz der Boost- und Qt-Libraries wird größtmögliche Portabilität erreicht. Weil Softwareentwicklung nicht nur Programmierung ist, finden Sie hier auch Themen für die professionelle Arbeit im Team, u.a. die Automatisierung der Dokumentation von Programmen, die Versionskontrolle und Werkzeuge zur Projektverwaltung und projektinternen Kommunikation.

Das integrierte »C++-Rezeptbuch« mit mehr als 150 praktischen Lösungen, das sehr umfangreiche Register und das detaillierte Inhaltsverzeichnis machen das Buch zu einem unverzichtbaren Nachschlagewerk für alle, die sich im Studium oder professionell mit der Softwareentwicklung in C++ beschäftigen.

// »Brillieren kann das Buch vor allem dadurch, dass es dem Leser alle im professionellen Umfeld notwendigen Hilfsmittel zurechtlegt. [...] Die Sprache ist verständlich und kurzweilig, die zahlreichen Codebeispiele sind gut gewählt und erklärt« //

Dr. Ulrich **BREYMANN** ist Professor für Informatik an der Hochschule Bremen. Er engagierte sich im DIN-Arbeitskreis zur Standardisierung von C++ und ist ein renommierter Autor zum Thema C++.

HANSE

www.hanser.de/computer

€ 49,90 [D] | € 51,30 [A]

ISBN 978-3-446-42691-7



9 783446 427549

C++-Programmierer, Einsteiger bis Profis