

Jürgen Wolf

C++

Das umfassende Handbuch

- ▶ Das Lehr- und Nachschlagewerk zu C++
- ▶ Inkl. der Neuerungen von C++11
- ▶ Sprachgrundlagen, OOP, Multithreading u. v. m.



Quellcode der Beispiele und
Entwicklungsumgebungen

Galileo Computing

Jürgen Wolf

C++

Das umfassende Handbuch

Liebe Leserin, lieber Leser,

Jürgen Wolf veröffentlichte bereits mehrere Bücher bei Galileo Computing, darunter die Standardwerke »C von A bis Z«, »Shell-Programmierung« und »Linux-UNIX-Programmierung«. All diese Werke sind von hoher fachlicher Qualität, vermitteln das jeweilige Themengebiet sehr gründlich und umfassend und sind zudem sehr unterhaltsam geschrieben. Diese Merkmale treffen auch auf sein Buch zur C++-Programmierung zu.

Die dritte Auflage dieses Buches, die Sie in Händen halten, ist mehr als eine aktualisierte Neuauflage. Tatsächlich wurde das Buch komplett neu geschrieben. Die vielen interessanten Neuerungen, die der Standard C++11 mit sich bringt, und die generelle Weiterentwicklung in der Programmierung machte eine Neukonzeption notwendig.

Sowohl ein Programmieranfänger (mit guten PC-Kenntnissen) als auch ein »ausgewachsener« C++-Entwickler wird dieses Werk mit Gewinn lesen. Die einen werden Schritt für Schritt vorgehen, die anderen gezielt nach Themen oder Lösungen suchen. Und beide werden nicht enttäuscht werden, denn sie finden hier eine praktische Einführung in die Sprache ebenso wie ausführliches Fachwissen zur Template-Entwicklung oder zur GUI-Programmierung.

Sollten Sie nach der Lektüre Fragen zu diesem Buch haben, Lob oder Kritik äußern wollen, dann wenden Sie sich an mich.

Und jetzt wünsche ich Ihnen viel Spaß beim Lesen!

Ihre Anne Scheibe

Lektorat Galileo Computing

anne.scheibe@galileo-press.de

www.galileocomputing.de

Galileo Press · Rheinwerkallee 4 · 53227 Bonn

Auf einen Blick

1	Wir richten uns ein und bringen es ans Laufen	25
2	Die Basisdatentypen in C++	51
3	Kontrollstrukturen	99
4	Jenseits der Basisdatentypen	129
5	Funktionen	187
6	Modularisierung	227
7	Grundlagen zu den Klassen	269
8	Operatoren überladen	359
9	Vererbung	407
10	Ausnahmebehandlung (Exceptions)	467
11	Template-Programmierung	507
12	Container, Iteratoren, Algorithmen und Hilfsmittel	545
13	Die Stream-Ein-/Ausgabeklassen von C++	647
14	Reguläre Ausdrücke (C++11)	695
15	Multithreading (C++11)	719
16	Weitere Neuerungen in C++11	773
17	GUI-Programmierung mit Qt	833

Wir hoffen sehr, dass Ihnen dieses Buch gefallen hat. Bitte teilen Sie uns doch Ihre Meinung mit. Eine E-Mail mit Ihrem Lob oder Tadel senden Sie direkt an die Lektorin des Buches: anne.scheibe@galileo-press.de. Im Falle einer Reklamation steht Ihnen gerne unser Leserservice zur Verfügung: service@galileo-press.de. Informationen über Rezensionen- und Schulungsexemplare erhalten Sie von: britta.behrens@galileo-press.de.

Informationen zum Verlag und weitere Kontaktmöglichkeiten finden Sie auf unserer Verlagswebsite www.galileo-press.de. Dort können Sie sich auch umfassend und aus erster Hand über unser aktuelles Verlagsprogramm informieren und alle unsere Bücher versandkostenfrei bestellen.

An diesem Buch haben viele mitgewirkt, insbesondere:

Lektorat Judith Stevens-Lemoine, Anne Scheibe

Fachgutachten Torsten T. Will, Bielefeld

Herstellung Norbert Englert

Korrektorat Annette Lennartz, Bonn

Layout Vera Brauner

Einbandgestaltung Janina Conrady

Satz Ill-satz, Husby

Druck Beck, Nördlingen

Dieses Buch wurde gesetzt aus der TheAntiquaB (9,35/13,7 pt) in FrameMaker.

Gedruckt wurde es auf chlorfrei gebleichtem Offsetpapier (80 g/m²).

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

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.

ISBN 978-3-8362-2021-7

© Galileo Press, Bonn 2014

3., aktualisierte Auflage 2014

Das vorliegende Werk ist in all seinen Teilen urheberrechtlich geschützt. Alle Rechte vorbehalten, insbesondere das Recht der Übersetzung, des Vortrags, der Reproduktion, der Vervielfältigung auf fotomechanischem oder anderen Wegen und der Speicherung in elektronischen Medien.

Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor, Herausgeber oder Übersetzer für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen.

Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

Inhalt

Geleitwort	19
Vorwort	21
1 Wir richten uns ein und bringen es ans Laufen	25
1.1 Von der Quelldatei zum ausführbaren Programm	25
1.2 Übersicht über die Entwicklungsumgebungen	26
1.2.1 Auf allen Systemen vorhandene Entwicklungsumgebungen	27
1.2.2 Entwicklungsumgebungen für Microsoft Windows	28
1.2.3 Entwicklungsumgebungen für Linux und Unix-like	30
1.2.4 Entwicklungsumgebungen für Mac OS	31
1.2.5 Reine Compiler ohne Entwicklungsumgebung	31
1.3 Quellcode übersetzen	32
1.3.1 Übersetzen mit Entwicklungsumgebung	32
1.3.2 Übersetzen mit der Kommandozeile	35
1.4 C++-Referenzen zum Nachschlagen (bzw. Laden)	37
1.5 Das Hauptprogramm – die main()-Funktion	38
1.5.1 Das Programm bei der Ausführung	40
1.5.2 Kommandozeilenargumente an main() übergeben	41
1.5.3 Quellcode kommentieren	43
1.5.4 Programmierstil	43
1.6 Die einfachen Streams für die Ein-/Ausgabe	43
1.6.1 Ausgabe auf dem Bildschirm	45
1.6.2 Einlesen von der Tastatur	48
1.6.3 Die Ein- und Ausgabeoperatoren >> und <<	48
1.7 Zusammenfassung	49
2 Die Basisdatentypen in C++	51
2.1 Grundlegendes zu Datentypen	51
2.1.1 Erlaubte Bezeichner für den Zugriff auf Variablen	51
2.1.2 Deklaration und Definition	53

2.1.3	Initialisierung von Variablen	54
2.1.4	Vereinheitlichte Initialisierung mit C++11	55
2.2	Ganzzahldatentypen (Integer-Datentypen)	55
2.2.1	Regeln für gültige Ganzzahlen	56
2.2.2	Ganzzahlen mit Werten initialisieren	57
2.2.3	Positive oder negative Ganzzahlen	58
2.2.4	Boolescher Datentyp für die Wahrheit	59
2.3	Typen für Gleitkommazahlen	60
2.4	Typ(en) für Zeichen	62
2.4.1	Der Datentyp char	62
2.4.2	Unicode-Unterstützung	65
2.5	Typ auto (C++11)	68
2.6	Übersicht und Größe der Basisdatentypen	70
2.6.1	Ganzzahltypen	70
2.6.2	Gleitkommazahlen	72
2.6.3	Byte-Größe ermitteln – sizeof-Operator	72
2.6.4	Sicherheit beim Kompilieren mit static_assert (C++11)	72
2.6.5	<limits> und std::numeric_limits	73
2.6.6	<climits> und <float>	75
2.6.7	Übersicht zu den fundamentalen Datentypen	76
2.7	Rechnen mit C++	77
2.7.1	Arithmetische Operatoren	78
2.7.2	Unäre Gegenstücke	80
2.7.3	Wenn Wertebereiche überschritten werden	81
2.7.4	Rundungsfehler bei Gleitkommazahlen	85
2.7.5	Komplexe Zahlen – <complex>	87
2.7.6	Nützliche mathematische Funktionen – <cmath>	89
2.8	Zufallszahlen (neu in C++11)	90
2.9	Konvertieren von Typen	92
2.9.1	Automatische (implizite) Typumwandlung	92
2.9.2	Automatische Typumwandlung beschränken (C++11)	95
2.9.3	Explizite Typumwandlung	96
3	Kontrollstrukturen	99
3.1	Bedingte Anweisung	99
3.1.1	Vergleichsoperatoren	102
3.1.2	Logische Operatoren	105

3.1.3	Verzweigung (if-else-Anweisung)	108
3.1.4	Bedingter Ausdruck (?:)	110
3.2	Fallunterscheidung (mehrfache Verzweigung)	111
3.2.1	Fallunterscheidung mit switch	111
3.2.2	Mehrfache Verzweigung mit else-if-Anweisung(en)	113
3.3	Schleifen (Wiederholungen)	115
3.3.1	Zähloperatoren (Inkrement und Dekrement)	116
3.3.2	Kopfgesteuerte Schleife – while()	117
3.3.3	Fußgesteuerte Schleife – do while()	119
3.3.4	Zählschleife – for()	121
3.3.5	Mengenschleife – Range-based for (C++11)	123
3.3.6	Endlosschleife	124
3.3.7	Kontrollierte Sprunganweisungen	125
4	Jenseits der Basisdatentypen	129
4.1	Arrays	129
4.1.1	Standardcontainer std::vector	130
4.1.2	Rohe Arrays (C-Style-Array)	135
4.1.3	Standardcontainer std::array (C++11)	139
4.1.4	Assoziatives Array	142
4.2	Strings (Zeichenketten)	143
4.2.1	Standardcontainer std::string	143
4.2.2	Rohe Strings (C-Style-Strings)	145
4.2.3	Unterstützung von Unicode (C++11)	150
4.2.4	Rohstringlitterale (Raw-String) (C++11)	152
4.3	Zeiger	153
4.4	Referenzen	164
4.5	Strukturen	166
4.6	Unions	177
4.7	Aufzählungstypen (C++11-Version)	180
4.7.1	Zugriff auf die Bezeichner	181
4.7.2	Typ für enum festlegen	182
4.8	Synonym-Technik	183
4.8.1	typedef	183
4.8.2	Alias-Templates (C++11)	185
4.9	Fazit	185

5	Funktionen	187
5.1	Funktionen definieren	187
5.2	Funktionen aufrufen	189
5.3	Funktionen deklarieren (Vorausdeklaration)	189
5.4	Exkurs: Gültigkeitsbereiche	191
5.4.1	Globaler Gültigkeitsbereich	192
5.4.2	Lokaler Gültigkeitsbereich	193
5.5	Funktionsparameter	195
5.5.1	Funktionsparameter als Kopie (Call by Value)	196
5.5.2	Funktionsparameter als Referenz (Call by Reference)	198
5.5.3	Funktionsparameter als rohe Zeiger	200
5.5.4	Strukturen und Klassen als Funktionsparameter	202
5.5.5	Schreibschutz für die Referenzparameter mit const	204
5.5.6	Default-Parameter (Standardparameter)	204
5.6	Rückgabewert aus einer Funktion	207
5.6.1	Referenz als Rückgabewert	210
5.6.2	Rohe Zeiger als Rückgabewert	211
5.6.3	Dinge, die man besser nicht zurückgibt	211
5.6.4	Mehrere Werte zurückgeben – std::pair<>	213
5.7	Funktionen überladen	215
5.8	Spezielle Funktionen	217
5.8.1	Inline-Funktionen	217
5.8.2	Lambda-Funktionen (C++11)	218
5.8.3	main()-Funktion	219
5.8.4	Funktions- und Programmende	221
5.9	Die neue Funktionssyntax (C++11)	221
5.9.1	Die neue Rückgabesyntax	222
5.9.2	decltype	223
5.10	Ausblick	226
6	Modularisierung	227
6.1	Namensräume	227
6.1.1	Einen neuen Namensraum erstellen	229
6.1.2	Namensraum verwenden	232

6.1.3	Aliases für Namensräume	236
6.1.4	Der Namensraum std	236
6.2	Speicherklassenattribute	238
6.2.1	Das Schlüsselwort extern	238
6.2.2	Das Schlüsselwort static	240
6.2.3	Aus Alt mach Neu und »deprecated«	243
6.3	Typqualifikatoren	244
6.3.1	Das Schlüsselwort const	244
6.3.2	Das Schlüsselwort volatile	246
6.4	Spezielle Schlüsselwörter	247
6.4.1	Das Schlüsselwort inline für Funktionen	247
6.4.2	Schlüsselwörter für Klassen (teilweise C++11)	247
6.4.3	Neue Schlüsselwörter mit C++11	248
6.5	Präprozessor-Direktiven	251
6.5.1	#include	252
6.5.2	#define und #undef	253
6.5.3	Bedingte Kompilierung	256
6.5.4	Weitere Direktiven	259
6.6	Modulare Programmierung – Code organisieren	260
6.6.1	Module	261
6.6.2	Sinnvolle Quellcodeaufteilung	261
6.6.3	Die öffentliche Schnittstelle (Headerdatei)	264
6.6.4	Die privaten Dateien	265
6.6.5	Die Client-Datei	266
7	Grundlagen zu den Klassen	269
7.1	Prinzip von Klassen	271
7.2	Klassen erstellen	272
7.3	Objekte einer Klasse erzeugen	274
7.4	Klassen(-Eigenschaften) initialisieren	276
7.4.1	Klassenelemente direkt initialisieren (C++11)	277
7.4.2	Konstruktoren	277
7.4.3	Initialisieren mit Methoden	293
7.5	Objekte zerstören – Destruktoren	295
7.5.1	Destruktor deklarieren	295

7.5.2	Destruktor definieren	296
7.5.3	Aufruf des Destruktors	296
7.6	Exkurs: Zugriffskontrolle auf die Klassenmitglieder	298
7.7	Methoden – die Funktionen der Klasse	303
7.7.1	Methoden deklarieren und definieren	303
7.7.2	Zugriffsmethoden – Setter und Getter	306
7.7.3	Zugriff auf die öffentlichen Mitglieder einer Klasse	311
7.7.4	const-Methoden (read-only)	314
7.7.5	Objekte als Methodenparameter	316
7.7.6	This-Zeiger	318
7.7.7	Objekte als Rückgabewert	321
7.7.8	Globale Hilfsfunktionen	322
7.7.9	Globale friend-Funktion	324
7.7.10	Methoden überladen	326
7.7.11	Die neue Funktionssyntax – decltype und auto (C++11)	327
7.8	Spezielle Eigenschaften einer Klasse	329
7.8.1	Konstante Elemente in einer Klasse	329
7.8.2	Roher Zeiger als Element in einer Klasse	332
7.8.3	Statische Eigenschaften in einer Klasse	338
7.8.4	Statische Methoden	341
7.8.5	Andere Klassen als Eigenschaft in einer Klasse	342
7.8.6	constexpr bei Klassen (Objekte zur Kompilierzeit) (C++11)	343
7.9	Erzeugen von Methoden steuern (C++11)	345
7.9.1	default	346
7.9.2	delete	348
7.10	Klassen für das Verschieben schreiben (C++11)	350
7.11	Friend-Klassen	357
8	Operatoren überladen	359
8.1	Grundlegendes zum Überladen von Operatoren	361
8.1.1	Binäre Operatoren	361
8.1.2	Binäre Operatoren als (friend-)Funktion überladen	364
8.1.3	Unäre Operatoren	366
8.1.4	Unäre Operatoren als (friend-)Funktion überladen	369
8.1.5	Unterschied zwischen Operatorüberladung und Methoden	371
8.1.6	Regeln für die Operatorüberladung	371
8.1.7	Überladbare Operatoren	372

8.2	Zuweisungsoperator – operator=	373
8.2.1	Zuweisungsoperator mehrfach überladen	376
8.2.2	Zuweisung verbieten (C++11)	378
8.3	Die Operatoren im Schnelldurchlauf	379
8.3.1	Der Zuweisungsoperator operator=	379
8.3.2	Binäre Operatoren – operator+, -, *, /, %	380
8.3.3	Erweiterte Schreibweise binärer Operatoren (+, -, *, /, %=)	380
8.3.4	Unäre Operatoren (+, -)	382
8.3.5	Bitweise Operatoren (& (binär), , ^)	383
8.3.6	Erweiterte Schreibweise bitweiser Operatoren (&=, =, ^=)	383
8.3.7	Logische Operatoren (==, !=)	383
8.3.8	Die logischen Verknüpfungen (&&, und !)	386
8.3.9	Vergleichsoperatoren (<, <=, >=, >)	386
8.3.10	Inkrement- und Dekrement-Operator (++ , --)	389
8.3.11	Die Operatoren operator*, operator-> und operator->*	389
8.3.12	Der Adressoperator &	389
8.3.13	Der Komplementoperator ~	389
8.3.14	Ein- und Ausgabeoperatoren – operator>> und operator<<	390
8.3.15	Der Konvertierungsoperator ()	392
8.3.16	Funktionsoperator () – Funktionsobjekte	395
8.3.17	Der Indexoperator []	397
8.3.18	Die Operatoren new, new[], delete und delete []	398
8.4	Übersicht der Operatoren in einer Tabelle	400
9	Vererbung	407
9.1	Grundlagen zur Vererbung	407
9.2	Abgeleitete Klassen implementieren	410
9.2.1	Basisklasse erstellen	411
9.2.2	Von der Basisklasse ableiten	412
9.2.3	Erweiterte Klasse verwenden	414
9.2.4	Direkte und indirekte Basisklasse	415
9.2.5	Abgeleitete Klassen verbieten (C++11)	416
9.3	Zugriffsschutz anpassen	417
9.4	Abgeleitete Klassen verwenden und erweitern	423
9.4.1	Zugriff auf die erweiterten Mitglieder der abgeleiteten Klasse	423
9.4.2	Zugriff auf die Mitglieder der Basisklasse	424
9.4.3	Suche nach dem passenden Namen	426
9.4.4	Redefinition von Methoden	426

9.5	Auf- und Abbau von abgeleiteten Klassen	428
9.5.1	Aufbau von Objekten	428
9.5.2	Basisklasse initialisieren	429
9.5.3	Abbauen von Objekten	431
9.5.4	Konstruktoren erben (C++11)	432
9.6	Implizite und explizite Typumwandlung in der Klassenhierarchie	434
9.6.1	Abgeleitete Objekte an Basisklassenobjekte	434
9.6.2	Basisklassenobjekt an abgeleitete Objekte	435
9.7	Polymorphie mithilfe von virtuellen Methoden	436
9.8	Virtueller Destruktor	442
9.9	Pure virtuelle Methoden und abstrakte Klassen	445
9.9.1	Pure virtuelle Methoden	445
9.9.2	Abstrakte Klassen	446
9.9.3	Reine Interface-Klassen	450
9.9.4	Typinformationen zur Laufzeit	451
9.9.5	Überschreiben erzwingen mit override (C++11)	453
9.9.6	Nicht mehr überschreiben mit final (C++11)	455
9.10	Mehrfachvererbung	457
9.10.1	Mehrdeutigkeiten	461
9.10.2	Virtuelle Basisklassen (virtuelle Vererbung)	463
10	Ausnahmebehandlung (Exceptions)	467
10.1	Prinzip der Ausnahmebehandlung	468
10.2	Ausnahmebehandlung implementieren	469
10.2.1	Ausnahme einleiten – try	470
10.2.2	Ausnahme werfen – throw	471
10.2.3	Ausnahme abfangen – catch	472
10.2.4	Mehrere Ausnahmen abfangen	474
10.2.5	Unbekannte oder alternative Ausnahmen abfangen	476
10.2.6	Aufräumarbeiten bei der Ausnahmebehandlung	477
10.2.7	Ausnahmen weiterleiten	478
10.2.8	try-catch verschachteln	479
10.2.9	Sinnvoller Einsatz von Ausnahmebehandlungen	480
10.3	Ausnahmeklasse implementieren	481
10.3.1	Ausnahmeklasse auslösen und abfangen	482
10.3.2	Ausnahmeklasse ableiten	485

10.4	Standardausnahmen von C++	485
10.4.1	Logische Fehler (logic_error)	487
10.4.2	Laufzeitfehler (runtime_error)	489
10.4.3	Weitere Standardfehlerklassen	493
10.4.4	Fehlermeldung mit what()	496
10.5	Spezielle Fehlerbehandlungsfunktionen	496
10.5.1	terminate() behandeln	497
10.5.2	std::uncaught_exception()	499
10.5.3	Ausnahme-Objekte zwischenspeichern (C++11)	499
10.5.4	noexcept (C++11)	501
10.5.5	Ausnahme-Spezifikation unexpected() behandeln (veraltet/deprecated)	503
10.6	Gefahren bei der Ausnahmebehandlung	504
10.6.1	Ausnahmen im Konstruktor	504
10.6.2	Ausnahmen im Destruktor	504
10.6.3	Aufräumprozess	504
10.6.4	Reservierter Speicher vom Heap	506
11	Template-Programmierung	507
11.1	Funktions-Templates	508
11.1.1	Funktions-Templates implementieren	509
11.1.2	Aufruf des Funktions-Templates	512
11.1.3	Funktions-Template spezialisieren	515
11.1.4	Mehrere Template-Parameter verwenden	518
11.1.5	Explizite Template-Argumente	520
11.1.6	Methoden-Templates	522
11.2	Klassen-Templates	523
11.2.1	Klassen-Templates implementieren	524
11.2.2	Methoden von Klassen-Templates implementieren	525
11.2.3	Objekte aus Klassen-Templates erzeugen	527
11.2.4	Klassen-Templates mit mehreren formalen Datentypen	528
11.2.5	Klassen-Templates mit Non-Type-Parameter	530
11.2.6	Klassen-Templates mit Default-Wert	532
11.2.7	Klassen-Templates spezialisieren	535
11.2.8	Klassen-Template als Parameter an Funktion/Methode	538
11.3	Templates mit variabler Argumentanzahl (C++11)	539

12	Container, Iteratoren, Algorithmen und Hilfsmittel	545
12.1	Grundlagen	546
12.2	Standardcontainer-Klassen	550
12.2.1	Sequenzielle Containerklassen	550
12.2.2	Assoziative Containerklassen	560
12.2.3	Container für Bit-Manipulationen – bitset	571
12.2.4	Neue Möglichkeiten mit C++11	572
12.3	Kleine Methodenübersicht aller Containerklassen	578
12.3.1	Methodenübersicht von sequenzielle Containern	579
12.3.2	Methodenübersicht von assoziativen Containern	584
12.4	Iteratoren	590
12.4.1	Kategorien von Iteratoren	595
12.4.2	Iterator-Funktionen	597
12.4.3	Iterator-Adapter	600
12.5	Algorithmen	603
12.5.1	Bereich	604
12.5.2	Mehrere Bereiche	605
12.5.3	Algorithmen mit Prädikat	606
12.5.4	Algorithmen mit einfachen unären Funktionen	608
12.5.5	Funktionsobjekte	608
12.5.6	Lambda-Funktionen (C++11)	613
12.5.7	Übersicht zu den Algorithmen	619
12.6	Hilfsmittel	627
12.6.1	Template pair	627
12.6.2	Tupel, das bessere pair (C++11)	632
12.6.3	Vergleichsoperatoren für eigene Typen	638
12.6.4	std::bind (C++11)	639
12.6.5	Verschieben mit move() (C++11)	642
12.6.6	Smart Pointer (C++11)	642
12.6.7	Zeitbibliothek – <chrono> (C++11)	643
13	Die Stream-Ein-/Ausgabeklassen von C++	647
13.1	Das Ein-/Ausgabe-Stream-Konzept von C++	647
13.2	Globale, vordefinierte Standard-Streams	648

13.3	Methoden für die Aus- und Eingabe von Streams	650
13.3.1	Methoden für die unformatierte Ausgabe	651
13.3.2	Methoden für die (unformatierte) Eingabe	652
13.4	Fehlerbehandlung bzw. Zustand von Streams	655
13.5	Streams manipulieren und formatieren	659
13.5.1	Manipulatoren	660
13.5.2	Eigene Manipulatoren ohne Argumente erstellen	666
13.5.3	Eigene Manipulatoren mit Argumenten erstellen	667
13.5.4	Format-Flags direkt ändern	669
13.6	Streams für die Datei-Ein-/Ausgabe	672
13.6.1	Streams für die Datei-Ein-/Ausgabe	673
13.6.2	Verbindung zu einer Datei herstellen	673
13.6.3	Lesen und Schreiben	678
13.6.4	Wahlfreier Zugriff	686
13.7	Streams für Strings	687
13.8	Stream-Puffer	692

14 Reguläre Ausdrücke (C++11) 695

14.1	Grammatik-Grundlagen zu den regulären Ausdrücken	696
14.2	Ein Objekt für reguläre Ausrücke erzeugen	699
14.3	Suchergebnis analysieren	702
14.4	Algorithmen für reguläre Ausdrücke	705
14.4.1	Genaue Treffer mit <code>regex_match</code>	706
14.4.2	Erweiterte Suche mit <code>regex_search</code>	709
14.4.3	Ersetzen mit <code>regex_replace</code>	711
14.4.4	Suchen mit <code>regex_iterator</code> und <code>regex_token_iterator</code>	713

15 Multithreading (C++11) 719

15.1	Die Grundlagen	719
15.2	Threads erzeugen	722
15.2.1	Argumente für den Thread	723
15.2.2	Methoden für Threads	727
15.2.3	Funktionen für Threads	732
15.2.4	Wie viele Threads sollen es sein?	733

15.3	Gemeinsame Daten synchronisieren	734
15.3.1	Schutz der Daten über einen Mutex	736
15.3.2	Schutz der Daten über Locks	741
15.3.3	Daten sicher initialisieren	747
15.3.4	Statische Variablen	749
15.4	Threadlokale Daten	750
15.5	Threads synchronisieren	751
15.6	Asynchrones Arbeiten (Future und Promise)	755
15.6.1	Futures	757
15.6.2	Promise	760
15.6.3	Methoden für future und promise	766
15.7	packaged_task	767
16	Weitere Neuerungen in C++11	773
16.1	Move-Semantik und Perfect Forwarding	773
16.1.1	Exkurs: LValue und RValue	774
16.1.2	RValue-Referenz und Move-Semantik	776
16.1.3	Perfect Forwarding	781
16.2	Benutzerdefinierte Literale (C++11)	784
16.3	Smart Pointer (C++11)	789
16.3.1	Shared Pointer	791
16.3.2	Weak Pointer	801
16.3.3	Unique Pointer	804
16.3.4	Weiteres zu den Smart Pointern	810
16.4	Zeitbibliothek	812
16.4.1	Zeitgeber (Clock)	812
16.4.2	Zeitpunkt (time_point) und Systemzeit (system_clock)	813
16.4.3	Zeitdauer (duration)	815
16.5	Type-Traits	820
16.6	POD (Plain Old Data) (C++11)	826
16.7	std::initializer_list	828

17	GUI-Programmierung mit Qt	833
17.1	Ein erstes Programmbeispiel schreiben	836
17.1.1	Kurze Übersicht zur Oberfläche von Qt Creator	837
17.1.2	Ein einfaches Projekt erstellen	838
17.2	Signale und Slots	846
17.2.1	Verbindung zwischen Signal und Slot herstellen	847
17.2.2	Signal und Slot mithilfe der Qt-Referenz ermitteln	848
17.3	Klassenhierarchie von Qt	866
17.3.1	Basisklasse QObject	866
17.3.2	Klassenhierarchie	866
17.3.3	Speicherverwaltung	869
17.4	Eigene Widgets erstellen	870
17.5	Eigene Widgets mit dem Qt Designer erstellen	873
17.6	Widgets anordnen	880
17.6.1	Grundlegende Widgets für das Layout	881
17.7	Dialoge erstellen mit QDialog	888
17.8	Vorgefertigte Dialoge von Qt	896
17.8.1	QMessageBox – der klassische Nachrichtendialog	896
17.8.2	QFileDialog – Dialog zur Dateiauswahl	902
17.8.3	QInputDialog – Dialog zur Eingabe von Daten	907
17.8.4	Weitere Dialoge	909
17.9	Eigenen Dialog mit dem Qt Designer erstellen	909
17.10	Grafische Bedienelemente von Qt (Qt-Widgets)	930
17.10.1	Schaltflächen (Basisklasse QAbstractButton)	930
17.10.2	Container-Widgets (Behälter-Widgets)	940
17.10.3	Widgets zur Zustandsanzeige	950
17.10.4	Widgets zur Eingabe	953
17.10.5	Online-Hilfen	966
17.11	Anwendungen in einem Hauptfenster	969
17.11.1	Die Klasse für das Hauptfenster QMainWindow	969
17.11.2	Eine Menüleiste für das Hauptfenster (QMenu und QMenuBar)	971
17.11.3	Eine Statusleiste mit QStatusBar	980
17.11.4	Eine Werkzeugleiste mit der Klasse QToolBar	985
17.11.5	An- und abdockbare Widgets im Hauptfenster mit QDockWidget	987
17.11.6	Einstellungen sichern mit QSettings	990
17.11.7	Kompletter Quellcode des Texteditors	996

17.12 Anwendung lokalisieren mit Qt Linguistic	998
17.13 Anwendungen in einem Hauptfenster mit dem Qt Designer	1006
17.14 Qt Designer vs. handgeschrieben	1018
17.15 Dinge, die man wissen sollte	1018
17.15.1 QApplication und QApplication	1019
17.15.2 Konsolenanwendungen mit Qt	1019
17.15.3 Wenn keine Icons angezeigt werden	1021
17.15.4 Das Ressourcen-System	1022
17.16 Klassen und Typen zum Speichern von Daten	1027
17.16.1 Qt-eigene Typdefinitionen	1027
17.16.2 QString	1028
17.16.3 QStringList	1030
17.16.4 QVariant	1031
17.16.5 Typen für Datum und Uhrzeit	1031
17.17 Fazit	1032

Anhang

A.1 Operatoren in C++ und deren Bedeutung (Übersicht)	1033
A.2 Vorrangtabelle der Operatoren	1035
A.3 Schlüsselwörter von C++	1037
A.4 Informationsspeicherung	1037
A.5 Zeichensätze	1044
 Index	 1051

Geleitwort

Mit der Einführung des neuen C++-Standards von 2011 ist der Bedarf nach einem Buch wie diesem groß geworden. Es haben sich viele Dinge geändert, die dem Neuling den Einstieg ermöglichen und dem erfahrenen Programmierer die Arbeit erleichtern. Den wenigsten kann man empfehlen, sich den Originaltext des Standards anzutun. Das haben die Compilerbauer getan: Für *Clang*, den C++-Compiler zur LLVM, wurde die komplette Unterstützung des C++11-Standards schon lange verkündet, der *g++* beherrscht ihn beinahe komplett und Microsoft und die anderen schreiben mit jedem Release weitere C++11-Unterstützung in die Prospekte. Nun müssen die Programmierer nachziehen: Gebraucht wird ein didaktisch gut aufgebautes Buch, das auch einen umfassenden Einblick in die komplette Sprache gibt.

Noch viel wichtiger finde ich, dass alte C-Zöpfe abgeschnitten werden. Wir werden C noch lange brauchen – gar niemals darauf verzichten können. Doch C++ kann mehr und mit der aktuellen Version wird die Sprache richtig erwachsen. Und so gibt es einige Dinge, die man heute in C++ anders machen sollte als noch vor zehn Jahren.

Beginnend bei »allerlei Entwicklungsumgebungen« bis zu »Zusätzliches in C++11« führt Jürgen Wolf Sie mit modernem Programmierstil und didaktisch gekonnt in alle Thematiken ein. Das abschließende Thema »Qt« ist insofern gut gewählt, weil es Sie in die Programmierung von GUIs plattformübergreifend einführt. Sie können sowohl unter Windows als auch unter Linux und auf dem Mac Fensterprogramme entwickeln. Mit Qt selbst können Sie sogar für Ihr Blackberry Smartphone-Programme schreiben, und viel von dem Gesagten ist auch für das iPhone, ohne Qt, nicht falsch.

Gerade diesen umfassenden Überblick schätze ich an diesem Buch. C++ ist – wie Bjarne Stroustrup gerne sagt – eine »Multiparadigmensprache«. Das heißt, Sie können beim Programmieren wählen, ob Sie sich der Vererbung der Objektorientierung bedienen (Kapitel 7 und 9), Verhalten über Verallgemeinerung auf Typen per Templates hinzufügen (Kapitel 11) oder generische Programmierung mit den vorhandenen Containern und Algorithmen verwenden (Kapitel 12).

Ein umfassendes Buch muss all diese Aspekte behandeln. In anderen Programmiersprachen heißt Wiederverwendung von Code automatisch Objektorientierung, Vererbung, überschreiben, virtuelles Verhalten, dynamische Typumwandlung usw. In C++ können Sie dies ebenfalls tun: Sie können sich auf genau jene Mittel beschränken, wie sie zum Beispiel auch Java bietet. Sie können aber auch darüber hinausgehen, wenn Sie wollen – Mehrfachvererbung, abstrakte Klassen mit Standardimplementierungen u. v. m. Es liegt an Ihnen, die richtigen Elemente für Ihr Problem auszuwählen. Sie sollten aber wissen, dass es in C++ noch andere Möglichkeiten für Wiederverwen-

dung, Modularisierung usw. gibt. Sind Type-Traits für Ihr Projekt möglicherweise die bessere Wahl, Daten und Operationen voneinander zu trennen? Vielleicht wollen Sie keine virtuellen Methoden und Ausnahmen mit dem Ballast, den diese jeweils mitbringen. Eine solche Flexibilität werden Sie in anderen populären Programmiersprachen kaum finden.

Beginnen Sie also über die Basistypen und Sprachkonstrukte von C++ zu lesen und lernen Sie die Standardbibliothek kennen. Erfahren Sie in diesem Buch, wie Sie die Dinge kombinieren – auch über die eingebauten Mittel hinaus.

Torsten T. Will

Vorwort

Es ist mir eine große Freude, Ihnen endlich die 3. und komplett neu geschriebene Auflage des Buches präsentieren zu dürfen. In dieser Auflage wurden alle Kritiken, die mich zur Vorgängerauflage erreicht haben, beherzigt und natürlich auch der neue C++11-Standard mit berücksichtigt. Daher war ein Neuschreiben des Titels unumgänglich.

Über dieses Buch

Auf dem Cover finden Sie den Untertitel »Das umfassende Handbuch«, und ich bin auch versucht, diesem Untertitel gerecht zu werden. Der Aufbau der einzelnen Kapitel ist recht einfach gehalten: Sie erfahren zunächst, worum es geht, gefolgt von mindestens einem Beispiel mit einer ausreichenden Erklärung dazu. Anders als in der Vorgängerauflage werden die Listings nicht mehr komplett abgedruckt, sondern nur die Stellen, die für das Kapitel bzw. Thema relevant sind. Dadurch steht dem Buch mehr Platz für weitere Themen zur Verfügung. Sie finden daher auch keine allzu komplizierten und überlangen Listings im Buch wieder. Die Listings sollten immer nur als Veranschaulichung des Themas verstanden werden und nicht als Teil eines umfangreichen Projekts. Selbstverständlich finden Sie die kompletten Listings auf der Buch-CD oder als Download auf der Bonus-Seite (<https://ssl.galileo-press.de/bonus-seite>).

Querverweise in den Kapiteln

Ein kurzer Hinweis auf die Querverweise im Buch auf andere Kapitel: Gerade als Einsteiger könnten diese Querverweise zunächst etwas verwirrend sein. Diese Querverweise dienen aber eher den Lesern, die etwas nachschlagen wollen. Es macht natürlich keinen Sinn, einen Einsteiger von Kapitel 3 ins Kapitel 10 springen zu lassen. Meistens handelt es sich lediglich um einen Hinweis, dass es zu diesem Thema noch etwas mehr zu sagen (oder eben zu wissen) gibt oder dass auf ein Thema vorgegriffen wurde (was sich leider nicht immer vermeiden lässt). Es ist nicht immer möglich, ein Thema aufbauend zu erklären, ohne (etwas) auf ein anderes Thema vorzugreifen. Trotzdem ist das Buch auch für Einsteiger so konzipiert, dass es von Kapitel 1 bis Kapitel 17 in sequenzieller Reihenfolge gelesen werden kann.

Zielgruppe

Das Buch fängt quasi bei null an und beginnt auch, wie von vielen Lesern immer wieder gewünscht, mit einigen Tipps zum Einrichten einer Entwicklungsumgebung.

Somit kann das Buch auch einem Anfänger ans Herz gelegt werden. Vorkenntnisse in einer anderen Programmiersprache sind nicht nötig. Gute Kenntnisse im Umgang mit dem Rechner sollten natürlich vorhanden sein. Da dieses Buch sehr viele Themen abdeckt, kann es auch als Nachschlagewerk und ständiger Begleiter verwendet werden. Aber auch leicht fortgeschrittenen Programmierern, welche ihre Kenntnisse um den neuen C++11-Standard erweitern wollen/müssen, dürften mit diesem Buch gut bedient sein.

C++11-Standard

Immer wenn ein neuer Standard verabschiedet wird, stellt sich die Frage, was davon kann mein Compiler und was nicht. So ist es auch diesmal mit dem C++11-Standard (genauer ISO/IEC 14882:2011). Die neueren Compiler können schon recht viel vom neuen C++11-Standard. Aber nicht jeder kann oder will den neuesten Compiler verwenden, und daher habe ich mich entschieden, in dieser Auflage noch nicht alles »aus einem Guss« zu erstellen. Daher finden Sie in den entsprechenden Kapiteln immer noch einen gesonderten Hinweis, wenn etwas nur im C++11-Standard funktioniert.

Betriebssystem

Gerade Einsteiger fragen gerne nach: Kann ich auf meinem System mit C++ anfangen? Hier kann ich zu 99% zustimmen. Im Buch wird vorwiegend das standardisierte C++ beschrieben, welches im Grunde unabhängig vom Betriebssystem ist und eher vom Compiler abhängt, den Sie verwenden. Und auch die Compilerhersteller wollen, dass Sie ihr Produkt verwenden, und beachten daher auch, dass Standard-C++ implementiert ist und verwendet werden kann. Aber da auch der Standard in unregelmäßigen Abständen erweitert wird (wie eben mit C++11 erwähnt), empfehle ich Ihnen, einen nicht allzu betagten Compiler zum Erlernen von C++ zu verwenden. Hierfür gibt es für jedes System mittlerweile mindestens eine (auch kostenlose) Variante, wie Sie es gleich im ersten Kapitel erfahren werden.

Danksagung und Schlussbemerkung

An dieser Stelle muss ich mich ganz besonders bei Torsten Will bedanken, der mein C++-Wissen auf den neuesten Stand gebracht und auch die Fachlesung für das Buch übernommen hat. Torsten hat hier schon mehr als nur eine gewöhnliche Fachlesung gemacht – es war schon eher ein Fernseminar. Torsten ist übrigens selbst Autor von C++-Büchern bei Galileo Press, und ich kann Ihnen seine Bücher wärmstens empfehlen. Mein Lieblingstitel von ihm, »C++11 programmieren – 60 Techniken für guten C++11-Code« würde sich bspw. ideal als Lektüre nach oder neben diesem Buch anbieten.

Weiterhin gilt mein Dank allen Mitarbeitern vom Galileo-Verlag, weil sie die Geduld mit mir und für dieses Buch nicht verloren haben, obwohl es länger gedauert hat, als geplant, allen voran Judith (Stevens-Lemoine) und Anne (Scheibe). Ebenfalls ein Riesendank gebührt natürlich meiner Familie, die mir wieder ein angenehmes Arbeiten zu Hause ermöglicht hat.

Obleich das Buch sehr umfangreich geworden ist, wurde es mit größter Sorgfalt geschrieben. Sollten Sie trotzdem einen Fehler finden oder etwas vermissen, scheuen Sie sich nicht, mich zu kontaktieren. Auch wenn es bei den ersten Schritten etwas hakt, dürfen Sie sich jederzeit bei mir melden. Ich freue mich auch über jedes Feedback.

Jetzt bleibt mir nur noch, Ihnen viel Spaß mit diesem Buch und der C++-Programmierung zu wünschen.

Jürgen Wolf (wolf@pronix.de)

Kapitel 1

Wir richten uns ein und bringen es ans Laufen

Bevor Sie C++ programmieren können, benötigen Sie erst das Werkzeug dafür. Um einen Quelltext zu schreiben, genügt im Grunde ein einfacher Texteditor und ein Compiler, um daraus einen Objektcode zu machen, und ein Linker, um aus dem Objektcode eine ausführbare Datei zu erstellen. Das Ganze gibt es allerdings auch als Rundum-sorglos-Paket in Form einer Entwicklungsumgebung. Wenn Sie sich für einen Compiler bzw. eine Entwicklungsumgebung entschieden haben, werden Sie hier auch gleich die ersten einfachen Schritte in C++ machen.

Der Einstieg in die C++-Welt beginnt oft mit der Frage nach der passenden Entwicklungsumgebung und danach, wie man einen Quellcode damit übersetzen und zur Ausführung bringen kann. Dieses Kapitel soll Ihnen daher einen kurzen Überblick über die gängigen Entwicklungswerkzeuge und das Grundgerüst eines C++-Programms verschaffen.

1.1 Von der Quelldatei zum ausführbaren Programm

Zum Schreiben von C++-Programmen wird mindestens ein reiner ASCII-Texteditor benötigt. Diese **Quelldatei** wird dann gewöhnlich mit der Endung `*.cpp` oder `*.cc` (oder falls es eine Headerdatei ist `*.h`) gespeichert. Wenn von *portabel* die Rede ist, dann ist hiermit noch der Quelltext gemeint (sofern dieser auch den C++-Standard einhält).

Grobe Beschreibung

Der Weg vom C++-Quelltext über den Compiler und Linker zur ausführbaren Datei wird hier nur grob beschrieben. Gerade die internen Arbeiten eines Compilers und Linkers sind erheblich umfangreicher und komplexer, als es nach dieser Beschreibung hier den Anschein hat.

Der **Compiler** ist ein spezielles Programm, welches den C++-Quellcode in eine Computer-Sprache übersetzt. Diese Computer-Sprache hat gewöhnlich eine binäre Form und wird auch als Objektcode bzw. Objektdatei (häufig anzutreffen mit der Endung:

`*.o` bzw. `*.obj`) bezeichnet. Beim Compilerlauf wird auch eine Syntaxüberprüfung und Fehleranalyse des Quellcodes vorgenommen. Stellt der Compiler hier schon C++-Syntaxfehler fest, werden die Fehlermeldungen protokolliert und ausgegeben. Abhängig vom Compiler und den Einstellungen, die verwendet werden, kann hierbei auch der Code optimiert werden.

Zum Schluss wird ein weiteres Programm, genannt **Linker** (engl. auch *linkage editor*), angewendet, um aus den einzelnen Programmmodulen ein ausführbares Programm zu erstellen (Abbildung 1.1). Sie ahnen es, auch unsere Objektdatei ist so ein Programmmodul. Hierbei ist es durchaus gängige Praxis, dass mehrere solcher Objektdateien zu einer ausführbaren Datei übersetzt werden. Neben Objektdateien können (und werden) auch Bibliotheken (oft vorzufinden mit der Endung `*.lib`) hinzugelinkt (zusammengefügt) werden, wie dies z.B. mit der Standardbibliothek von C++ der Fall ist.

Bibliotheken erstellen

Neben ausführbaren Dateien können mit dem Linker auch statische sowie dynamische Bibliotheken (mit den Endungen `*.lib` oder `*.dll`) erstellt werden. Die Dateieindungen sind in der Regel systemabhängig.

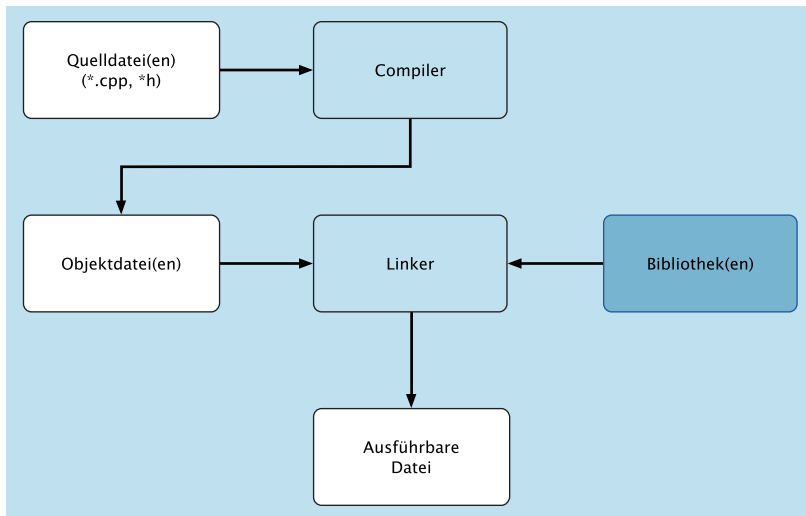


Abbildung 1.1 Vereinfachte Darstellung, wie aus einer Quelldatei eine ausführbare Datei gemacht wird

1.2 Übersicht über die Entwicklungsumgebungen

In der Praxis wird man wohl eher selten noch mit einem Texteditor, dem Compiler und Linker direkt hantieren. Trotzdem kann es nicht schaden, wenn Sie auch mit dem direkten Umgang (in der Kommandozeile) vertraut sind. Gerade auf Linux- bzw.

Unix-artigen Systemen ist es durchaus noch gängige Praxis. Größere Projekte werden allerdings mit einer **Entwicklungsumgebung** erstellt. Hier haben Sie den Vorteil, dass alles (Editor, Compiler, Linker usw.) gleich mit an Bord ist und Sie mit einer paar Klicks ein Programm übersetzen können. Aber der Nachteil bei solchen Entwicklungsumgebungen ist eben, dass man ein wenig Einarbeitungszeit benötigt.

Keine Werbung

An dieser Stelle möchte ich noch in eigener Sache erwähnen, dass ich nicht für einen bestimmten Compilerhersteller oder ein bestimmtes Betriebssystem werben will. Ich gebe Ihnen lediglich einen Überblick über die Marktsituation (ohne Garantie auf Vollständigkeit). Denn letztlich müssen Sie selbst entscheiden, welches Werkzeug Sie verwenden wollen, um aus Ihrem C++-Quellcode ein ausführbares Programm zu machen. Häufig hat man auch gar keine Wahl, den Compiler oder das Betriebssystem zu wählen, wenn man an einem großen Projekt in der Gruppe arbeiten muss.

1.2.1 Auf allen Systemen vorhandene Entwicklungsumgebungen

Es gibt ein paar spezielle Entwicklungsumgebungen, welche auf allen größeren Systemen wie Windows, Linux (und Unix-like) sowie Mac OS erhältlich sind und auch keinen Cent kosten. Zwei größere Projekte sind die Entwicklungsumgebungen **NetBeans** (Abbildung 1.2) und **Eclipse** (Abbildung 1.3). Beide wurden zwar in (und für) Java geschrieben, bieten aber trotzdem auch eine gute Unterstützung für andere Programmiersprachen (wie C++) an. Tatsächlich bedeutet dies allerdings auch, dass Java auf Ihrem System installiert sein muss.

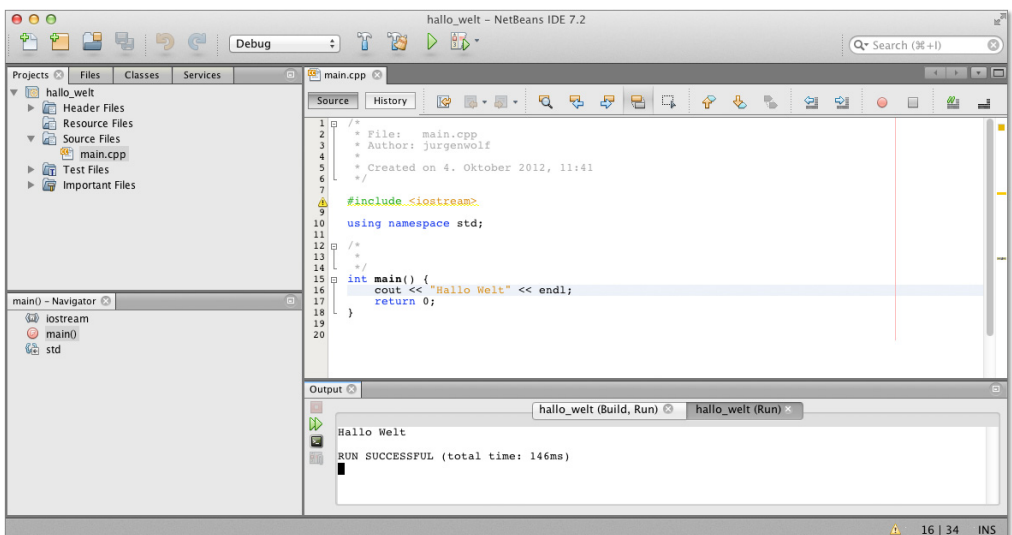


Abbildung 1.2 Die Entwicklungsumgebung NetBeans für C++ bei der Ausführung

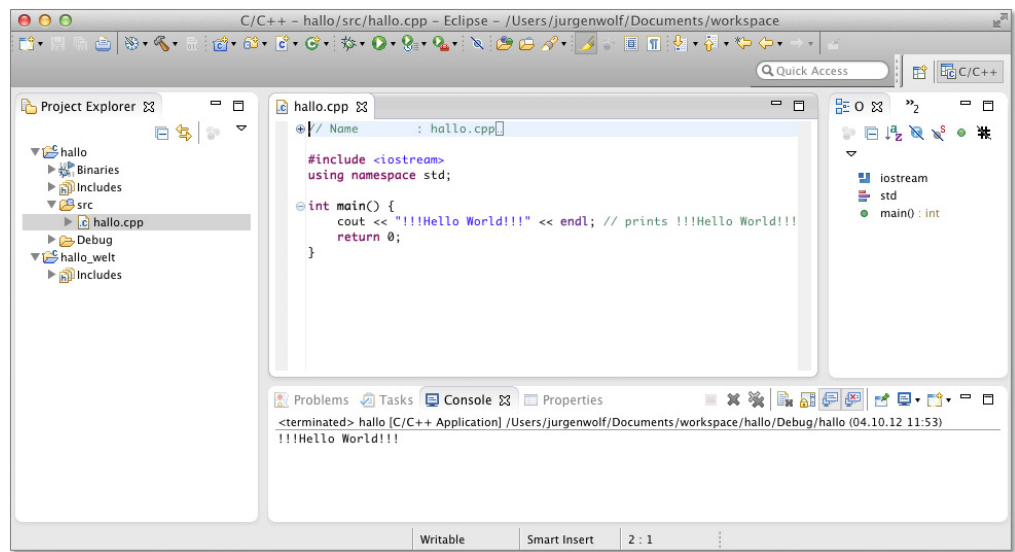


Abbildung 1.3 Die Entwicklungsumgebung Eclipse für C++ bei der Ausführung

Neben den eben vorgestellten Entwicklungsumgebungen sollte auf jeden Fall auch noch das **Qt SDK** erwähnt werden, womit sich, neben Anwendungen mit grafischer Oberfläche und dem Qt-Framework, auch gewöhnliche Standard-C++-Anwendungen erstellen lassen. Dann wäre da noch die Entwicklungsumgebung **Code::Blocks**, welche ebenfalls für alle gängigen Systeme erhältlich ist. Einen Überblick gibt Ihnen Tabelle 1.1.

IDE	Weblink	System
Eclipse	http://www.eclipse.org	plattformunabhängig
NetBeans	http://www.netbeans.org	plattformunabhängig
Qt SDK	http://qt-project.org/downloads	Win, Linux, Mac
Code::Blocks	http://www.codeblocks.org	Win, Linux, Mac

Tabelle 1.1 Entwicklungsumgebungen für mehrere Systeme

1.2.2 Entwicklungsumgebungen für Microsoft Windows

Platzhirsch auf Windows-Rechnern dürfte das hauseigene Produkt von Microsoft selbst sein. Da wäre zum einen die kommerzielle Version **Visual Studio** (mit unterschiedlichen Versionen) und dann die kostenlose Variante **Visual Studio Express-Edition** (für Windows Desktop). Für den Einstieg in C++ dürften die kostenlosen Express-Versionen von Visual Studio zunächst mehr als ausreichend sein. Microsoft bringt derzeit in einem Ein- bis Zwei-Jahres-Rhythmus eine neue Version von Visual

Studio auf dem Markt. Zur Drucklegung dieses Buches war die Version 2013 aktuell (Abbildung 1.4).

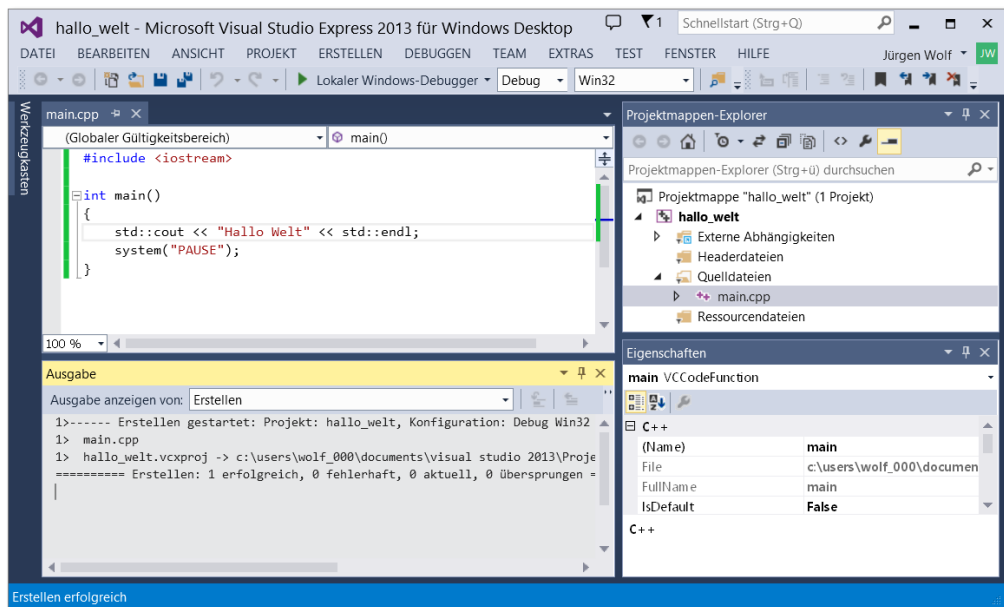


Abbildung 1.4 Visual Studio 2013 Express bei der Ausführung

Des Weiteren gibt es unter Windows viele andere Entwicklungsumgebungen, die den Compiler **MinGW** (eine Portierung des GCC auf Windows) verwenden. Einen Überblick über die einzelnen Windows-Entwicklungsumgebungen finden Sie in Tabelle 1.2.

IDE	Weblink	Bemerkung
Visual Studio	http://www.microsoft.com/visualstudio/deu/downloads	Die kommerzielle Version gibt es in Ultimate, Premium und Professionell.
Visual Studio Express-Edition	http://www.microsoft.com/visualstudio/deu/downloads	Bei der kostenlosen Version müssen Sie die Desktop-Version herunterladen. Neben C++ sind hierbei auch noch die Sprachen C# und Visual Basic enthalten.
Visual C++ 2010 Express	http://www.microsoft.com/visualstudio/deu/downloads	eine Vorgängerversion von Visual C++ Express für ältere Windowssysteme
Orwell Dev-C++	http://orwelldev-cpp.blogspot.de	Weiterentwicklung der Bloodshed-Dev-C++-Entwicklungsumgebung

Tabelle 1.2 Übersicht gängiger Entwicklungsumgebungen für Windows

1.2.3 Entwicklungsumgebungen für Linux und Unix-like

Bei Linux und anderen Unix-Systemen kommt als Compiler meistens das GCC-Paket zum Einsatz. Auf diesen Compilern aufbauend, gibt es viele interessante Entwicklungsumgebungen. Sehr beliebt sind hierbei **Anjuta** (Abbildung 1.5) und **KDevelop**. Persönlich gefällt mir die Entwicklungsumgebung **Geany** sehr gut, weil diese sehr schlicht und übersichtlich gestaltet ist.

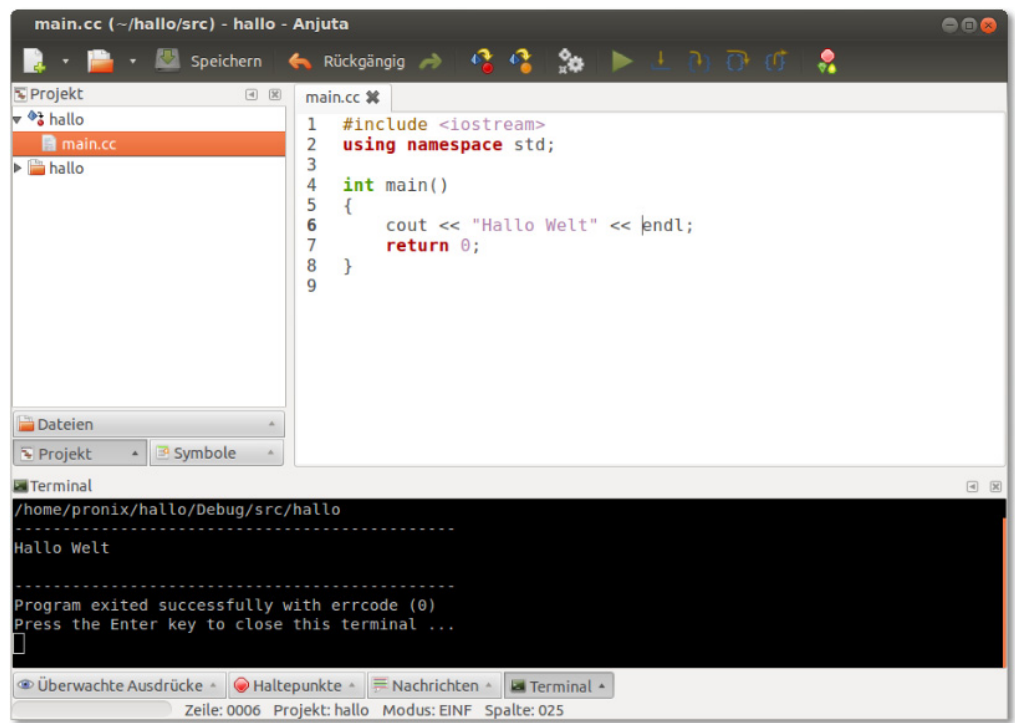


Abbildung 1.5 Die Entwicklungsumgebung Anjuta bei der Ausführung unter Ubuntu Linux

Bezugsadressen der Entwicklungsumgebungen für Linux finden Sie in Tabelle 1.3.

IDE	Weblink	Bemerkung
Anjuta	http://projects.gnome.org/anjuta	Entwicklungsumgebung für den GNOME-Desktop
KDevelop	http://www.kdevelop.org	Basiert auf der KDevPlatform und den KDE- und Qt-Bibliotheken.
Geany	http://www.geany.org	Basiert auf dem GTK+-Toolkit und kann auch unter Windows verwendet werden.

Tabelle 1.3 Übersicht gängiger Entwicklungsumgebungen für Linux

GCC, gcc, g++ und MinGW

GCC (großgeschrieben) ist die GNU Compiler Collection, also eine ganze Sammlung von Compilern. gcc (kleingeschrieben) ist der C-Compiler der Sammlung, und – Sie werden es vermuten – g++ ist der C++-Compiler der Sammlung GCC. MinGW (Minimalist GNU for Windows) ist eine Softwareportierung von GCC für Windows.

1.2.4 Entwicklungsumgebungen für Mac OS

Der Platzhirsch bei Apple dürfte ganz klar die hauseigene Entwicklungsumgebung Xcode sein, welche bequem und derzeit kostenlos über den App Store heruntergeladen werden kann (Abbildung 1.6).

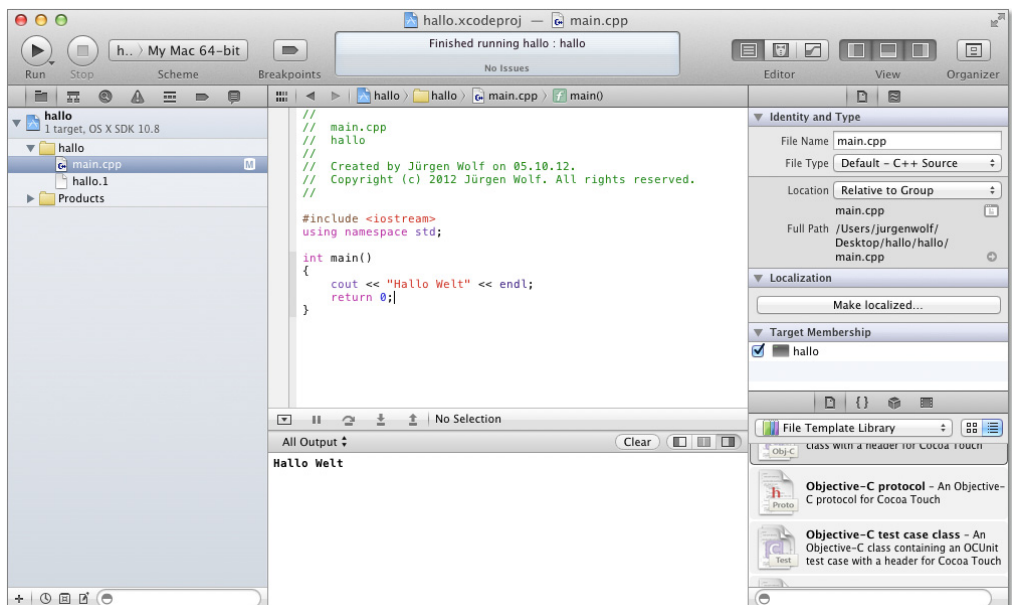


Abbildung 1.6 Der Platzhirsch auf dem Mac ist Xcode.

1.2.5 Reine Compiler ohne Entwicklungsumgebung

Es wurde bereits kurz erwähnt, dass viele Entwicklungsumgebungen im Grunde nur auf die für das System vorhandenen Compiler zurückgreifen. Genau genommen ist eine Entwicklungsumgebung nichts anderes als eine grafische Oberfläche, mit der Sie einen Compiler und andere Werkzeuge bedienen. So verwenden die meisten kostenlos erhältlichen Compiler unter Windows, wie z.B. NetBeans, Eclipse, Qt SDK, Code::Blocks oder Orwell Dev-C++, alle MinGW – die Softwareportierung für GCC. Entwicklungsumgebungen auf Linux, wie etwa Anjuta und KDevelop, aber auch Net-

Beans und Eclipse, verwenden wiederum auch nur den GCC. Für den Mac OS ist zwar auch noch GCC vorhanden und wird gerne verwendet, aber Xcode verwendet mittlerweile bevorzugt Clang, ein Backend des LLVM-Compilers. Microsoft verwendet für sein Visual Studio einen eigenen Compiler. Eine Übersicht reiner Compiler bietet Tabelle 1.4.

Compiler, Linker und noch vieles mehr

Zwar ist hier des Öfteren die Rede von einem *Compiler*, aber im Grunde sind hierbei auch der Linker und anderer Werkzeuge enthalten. Es handelt sich meistens um eine ganze Sammlung von Entwicklungswerkzeugen.

Compiler	Link	System
GCC	http://www.gnu.org	Win, Linux, Mac, Unix-like
Clang	http://www.clang.llvm.org	Unix-like, Mac, Windows, Linux
MinGW	http://www.mingw.org	Windows
Intel-C++	http://software.intel.com/en-us/intel-compilers	Linux, Win, Mac

Tabelle 1.4 Übersicht zu reinen Compilern (bzw. Sammlungen von Entwicklungswerkzeugen) für die Kommandozeile

1.3 Quellcode übersetzen

Nachdem Sie einige Entwicklungsumgebungen und Compiler kennen und sich vielleicht auch schon für ein Werkzeug entschieden haben, will ich Ihnen auf den nächsten Seiten zeigen, wie Sie mit einer Entwicklungsumgebung und in der Kommandozeile aus einem Quellcode ein ausführbares Programm machen können.

1.3.1 Übersetzen mit Entwicklungsumgebung

Als Entwicklungsumgebung wurde für diese Anleitung zwar **Visual Studio 2013 Express** verwendet, aber der Vorgang ist bei fast allen Entwicklungsumgebungen sehr ähnlich, nur dass hierbei der eine oder andere Befehl anders bezeichnet ist oder sich woanders befindet.

1. Der erste Schritt ist meistens, ein Projekt (bzw. eine Projektmappe) für C++ zu erstellen (Abbildung 1.7). Beim MS Visual Studio 2013 Express erledigen Sie dies über DATEI • NEUES PROJEKT oder die Tastenkombination **[Strg] + [N]**.

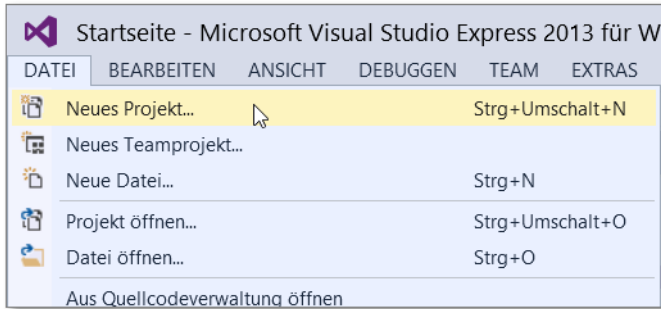


Abbildung 1.7 Als Erstes sollten Sie immer ein neues Projekt starten.

- Im nächsten Schritt erscheint gewöhnlich ein Dialog (häufig auch als Wizard oder Assistent bezeichnet), der Ihnen bei der Projektauswahl behilflich ist. In diesem Fall wählen Sie entweder ein LEERES PROJEKT (hier der Fall) oder, falls vorhanden, eine C++-Konsolenanwendung (Abbildung 1.8). Auch den Namen für das Projekt dürften Sie hier meistens schon vergeben (hier *hallo_welt*).

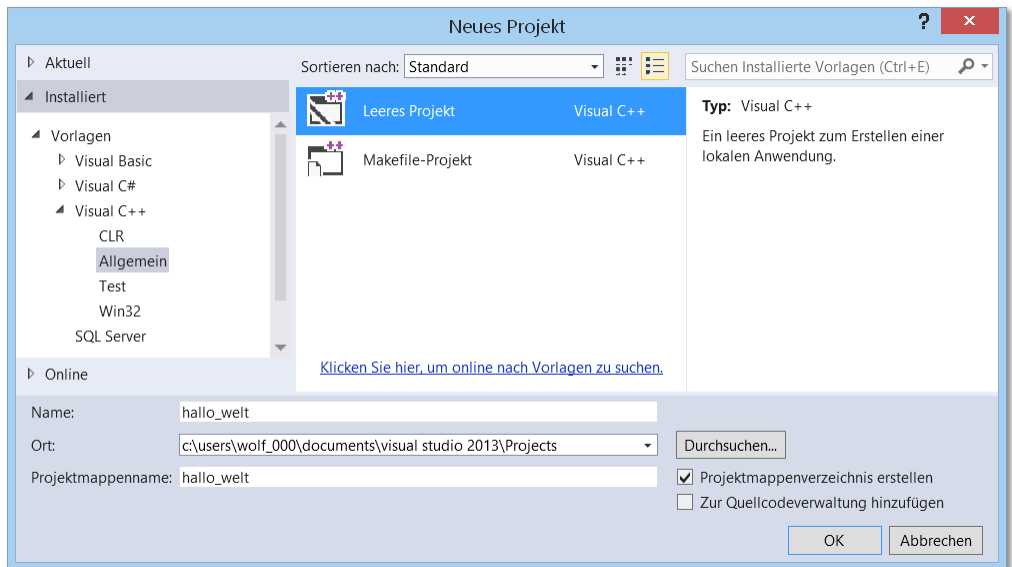


Abbildung 1.8 Ein Dialog hilft Ihnen beim Erstellen eines Projekts.

- Das neue Projekt wird jetzt gewöhnlich in der Entwicklungsumgebung geöffnet. Einige Entwicklungsumgebungen bieten hierbei häufig schon ein vordefiniertes Grundgerüst eines C++-Programms mit an. In unserem Fall ist dies nicht der Fall, weshalb wir erst noch eine neue Quelldatei hinzufügen müssen. Neue Dateien (bzw. Elemente) müssen Sie ohnehin irgendwann hinzufügen, wenn der Quellcode umfangreicher wird und Sie diesen in mehrere Quell- und Headerdateien

(auch Module genannt) aufteilen. Besonders wichtig ist hierbei, dass Sie die neue Datei auch dem Projekt hinzufügen und nicht nur eine neue leere Datei anlegen (Abbildung 1.9)!

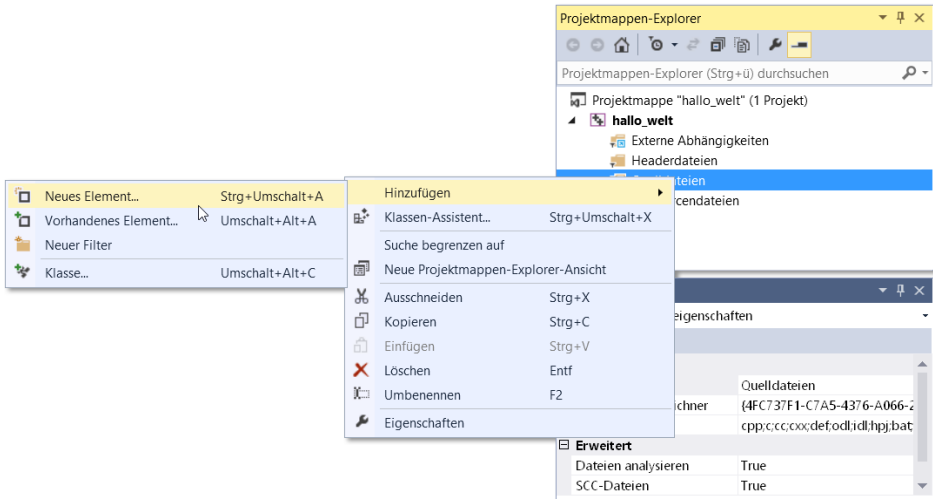


Abbildung 1.9 Neues Element (Datei) dem Projekt hinzufügen

4. Häufig wird auch hierfür wieder ein Dialog geöffnet, welcher Ihnen bei der Auswahl des neuen Elements für das Projekt behilflich ist (Abbildung 1.10). In unserem Fall wollen wir eine einfache C++-Datei mit der Endung `*.cpp` hinzufügen. In späteren Projekten, wenn Sie den Quellcode in mehrere Dateien aufteilen, werden Sie hier auch Headerdateien mit der Endung `*.h` hinzufügen.

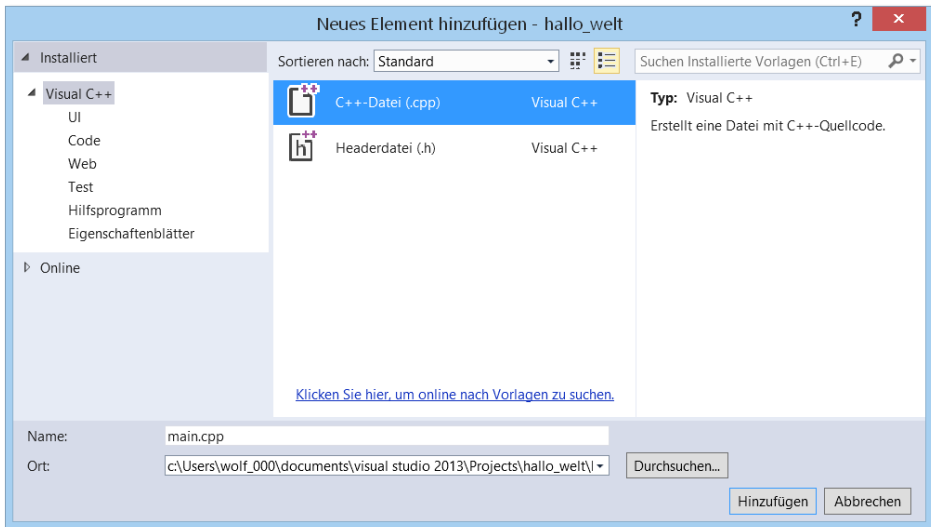


Abbildung 1.10 Ein weiterer Dialog hilft gewöhnlich bei der Auswahl der passenden Datei.

5. Wenn Sie eine neue Datei hinzugefügt haben, brauchen Sie diese nur noch in der Projektmappe zu aktivieren und dann den C++-Quelltext einzutippen. Nach dem Abspeichern können Sie den Quellcode übersetzen (kompilieren) und anschließend ausführen (Abbildung 1.11). In diesem Beispiel wird der Quellcode bspw. mit ERSTELLEN • KOMPILIEREN übersetzt und mit DEBUGGEN • STARTEN OHNE DEBUGGING ausgeführt.

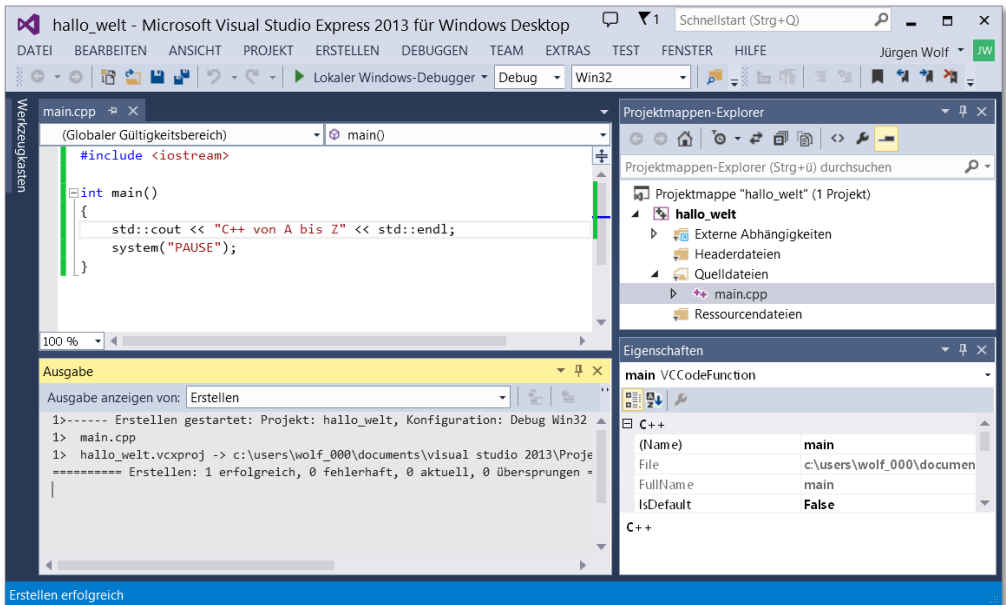


Abbildung 1.11 Nach dem Eintippen des C++-Quelltextes und dem Übersetzen

1.3.2 Übersetzen mit der Kommandozeile

Die Verwendung von Kommandozeilen-Compilern wie **g++** oder **clang++** mag sich zunächst vielleicht etwas nostalgisch anhören, aber hierbei haben Sie den Vorteil, dass Sie sich nicht in monströse Entwicklungsumgebungen einarbeiten müssen. Gerade Linux-, Mac- und Unix-like-Anwender greifen immer noch gerne zur Kommandozeile, um mal schnell einen Quelltext zu übersetzen. Selbstredend müssen Sie hierbei den Quelltext zunächst in einem separaten ASCII-Text-Editor eintippen. Ein weiterer enormer Vorteil ist es, dass bei den meisten Linux- und Unix-Systemen von Haus aus bereits der GCC (welcher auch **g++** enthält) installiert ist oder schnell mal eben nachinstalliert werden kann. Aber auch LLVM/CLANG mit **clang++** wird immer beliebter und ist auf dem Vormarsch (auch unter Linux).

Systeme ohne grafische Oberfläche

Außerdem gibt es auch Systeme ohne einen Fenstermanager (wie bspw. ein Webserver). Hier kommen Sie gar nicht um eine Übersetzung in der Kommandozeile herum. Als Editor werden hierbei dann häufig Klassiker wie **vi** oder **Emacs** verwendet.

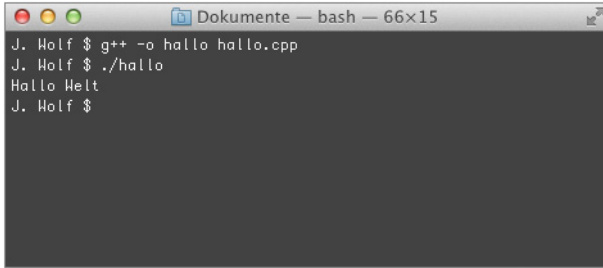


Abbildung 1.12 Hier wird der C++-Quellcode »hallo.cpp« übersetzt, den wir vorher geschrieben und gespeichert haben. Faktisch müssen Sie sich bei der Übersetzung in der Kommandozeile auch in dem Verzeichnis befinden, in dem Sie den C++-Quellcode gespeichert haben.

Beim Übersetzen in der Kommandozeile müssen Sie allerdings dem Compiler sagen, was Sie von ihm wollen. Dies können Sie über spezielle Optionen machen, wie hier mit `-o` geschehen (Abbildung 1.12). Mit dieser Option geben Sie an, dass die ausführbare Datei *hallo* heißen soll. Es wird also der Name verwendet, der hinter dem Schalter `-o` steht. Wollen Sie einen Quellcode nur kompilieren und eine Objektdatei erstellen, brauchen Sie nur die Option `-c` (`g++ -c quelltext.cpp`) zu verwenden. Ein wenig mitteilungsfreudiger können Sie den Compiler mit `-Wall` (sinnvolle Warnungen ausgeben) und `-pedantic` (Warnungen ausgeben die vom Standard gefordert werden) machen.

Um neue C++11-Sachen verwenden zu können, müssen Sie häufig noch zusätzlich den Standard mit dem Flag `-std=c++11` (bzw. auch `-std=gnu++11`) bei der Übersetzung mit angeben. Vorausgesetzt natürlich, Ihr Compiler unterstützt den neuen C++11-Standard.

Eine klassische Übersetzungszeile von einem Quellcode, wie ich sie bspw. in der Kommandozeile mit dem `g++` gerne verwende, lautet:

```
g++ -o main main.cpp -Wall -pedantic -std=c++11
```

Selbiges funktioniert auch mit `clang++`:

```
clang++ -o main main.cpp -Wall -pedantic -std=c++11
```

Mehr Informationen

Es gibt noch unglaublich viele weitere Optionen, aber hier muss ich Sie leider auf die Dokumentation des Compilers verweisen, um nicht zu sehr vom eigentlichen Thema des Buches – der C++-Programmierung – abzuschweifen. Eine Eingabe von `g++ --help`, `man g++` oder `info g++` liefert Ihnen viele weitere Informationen über `g++` und seinen Optionen.

1.4 C++-Referenzen zum Nachschlagen (bzw. Laden)

Der Umfang der C++-Standardbibliothek ist mittlerweile mehr als nur beachtlich. Würden wir in diesem Buch allen Headerdateien, Klassen und Methoden usw. eine Referenz hinzufügen, dann würden hierfür wohl noch weitere 1.000 Seiten benötigt. Alleine für die Standardcontainer gibt es mittlerweile schon ein eigenes (englischsprachiges) Buch auf dem Markt. Die Empfehlung, ein weiteres Buch mit einer C++-Referenz anzuschaffen, ginge dann allerdings doch zu weit – wenn Sie überhaupt eines finden würden, das wirklich »komplett« ist.

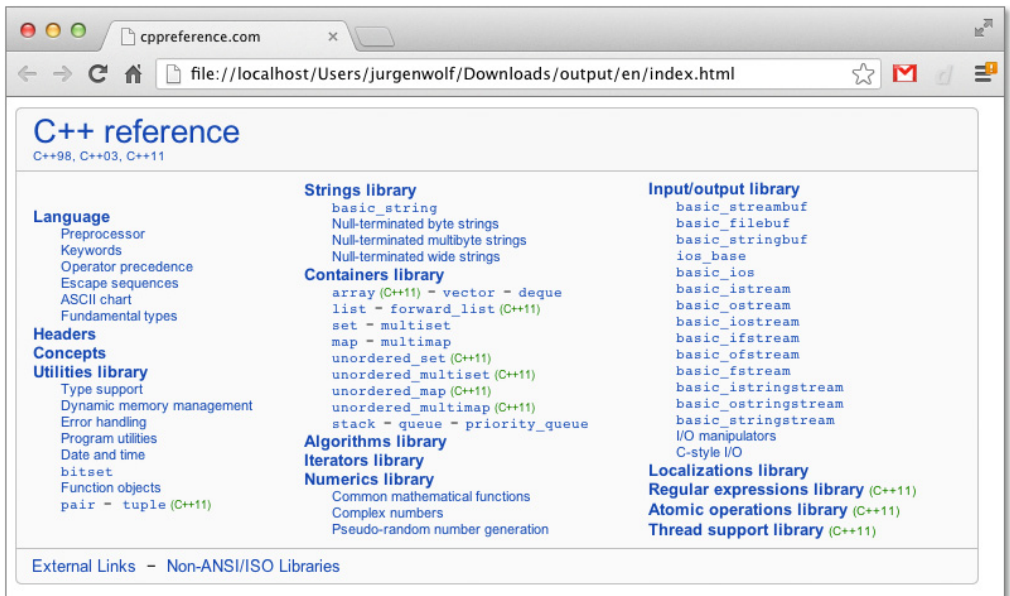


Abbildung 1.13 Die Offline-Version von <http://en.cppreference.com>

Ich empfehle Ihnen an dieser Stelle zwei interessante Webseiten, welche zum Teil eine komplette Referenz zum C++-Standard (inklusive C++11 und auch der Bibliothek zu ANSI C) enthalten: <http://www.cplusplus.com/reference> und [37](http://en.cppref-</p>
</div>
<div data-bbox=)

erence.com/w. Mein Geheimtipp ist hierbei <http://en.cppreference.com/w>, weil bei dieser Referenz teilweise schon mit der Implementierung des C++14-Standards angefangen wurde und Sie sich die komplette Referenz (C++98, C++03, C++11 und C89, C99 und C11) zum Offline-Lesen herunterladen können (Abbildung 1.13). Es gibt sogar eine herunterladbare Version für den Assistenten von Qt, um die Referenz in die Hilfe der Entwicklungsumgebung von Qt einzubinden.

1.5 Das Hauptprogramm – die main()-Funktion

Bevor Sie richtig in C++ einsteigen können, soll hier noch das Grundgerüst eines einfachen C++-Programms vorgestellt werden. Und zwar geht es um die `main()`-Funktion, welche der Einstiegspunkt ist, wenn das ausführbare Programm gestartet wurde. Ohne eine solche `main()`-Funktion könnte der Linker gar kein ausführbares Programm erstellen.

Groß- und Kleinschreibung wird unterschieden

Die erste Anmerkung gleich zu Beginn: Sie sollten auf jeden Fall wissen, dass C++ strengstens zwischen Groß- und Kleinschreibung unterscheidet (auch *Case Sensitivity* genannt).

Hierzu ein Minimalbeispiel, welches die Textfolge »C++ von A bis Z« auf dem Bildschirm ausgibt:

```
00 // kapitel01/hallo.cpp
01 #include <iostream>

02 int main()
03 {
04     std::cout << "C++ von A bis Z" << std::endl;
05     return 0;
06 }
```

In **01** finden Sie einen Präprozessor-Befehl der noch **vor** der Ausführung des Compilers ausgeführt wird. `std::cout` oder `std::endl` sind in der Headerdatei `<iostream>` enthalten und stehen dank `include` für unser Programm zur Verfügung.

In der Zeile **02** finden Sie mit `main()` den klassischen **Einstiegspunkt**, den jedes C++-Programm enthalten muss, wenn Sie ein ausführbares Programm daraus machen wollen. Die einzelnen Befehle der `main`-Funktion werden in einer sich öffnenden (**03**) und schließenden (**06**) geschweiften Klammer zusammengefasst. Dieser Bereich wird auch als **Anweisungsblock** bezeichnet.

Anweisungen mit Semikolon abschließen

Eine einzelne Anweisung innerhalb eines Anweisungsblocks wird mit einem Semikolon (;) abgeschlossen. Der Prozessor arbeitet quasi Anweisung für Anweisung ab. Mit dem Semikolon erkennt dieser dann das Ende der Anweisung und kann mit der nächsten Anweisung fortfahren (sehr vereinfacht beschrieben).

Alles innerhalb des Anweisungsblocks sind die Anweisungen. In der Zeile **04** finden Sie eine Anweisung, in der über `std::cout` und `<<` der Text zwischen den Gänsefüßchen (hier "C++ von A bis Z") auf die Standardausgabe (hier den Bildschirm) ausgegeben wird. Damit noch ein Zeilenumbruch gemacht wird, wurde hier nochmals über den `<<`-Operator ein `std::endl` nachgeschoben.

Anweisung (engl. »statement«)

Vereinfacht ausgedrückt, ist eine Anweisung eine Arbeitsvorschrift im Programm, welche in der entsprechenden Programmiersprache verfasst wird und auch in mehreren Teilanweisungen vorliegen kann. Im Rahmen dieser Arbeitsvorschriften, wird die Abarbeitung des Programms festgelegt.

In der Zeile **05** geben Sie der Funktion `main()` mit `return` den Wert `0` zurück, weil `main()` einen Rückgabewert vom Typ `int` erwartet. Im Fall der `main`-Funktion können Sie aber auf die Zeile **05** komplett verzichten, weil dies bei der `main`-Funktion implizit gemacht würde. Beachten Sie allerdings, dass dies nur bei der `main`-Funktion möglich ist – eine absolute Ausnahmeregelung also!

`std::` oder `using namespace std`

Sofern Sie vielleicht schon erste Erfahrungen mit C++ gemacht haben, könnte es sein, dass Sie hier oft eine globales `using namespace std;` hingeschrieben finden, womit der Zugriff auf bspw. `cout` oder `endl` über den Namensraum `std` und den Bereichsoperator `::` entfallen kann (also dann einfach: `cout << "C++ von A bis Z" << endl;`). Diese Anwendung ohne `std::` ist keineswegs falsch und wird durchaus gerne verwendet. Aber Sie sollten bedenken, dass hier von den Standard-C++-Entwicklern etwas in einem bestimmten Namensbereich (hier `std`) zusammengefasst wurde (um Namenskonflikte zu vermeiden) und dass es dann nicht dem Sinn und Zweck entsprechen kann, alles wieder global einzublenden.

Sicherlich gibt es auch andere Möglichkeiten, nur bestimmte Mitglieder des Namensraumes in bestimmten Bereichen einzublenden, aber das geht hier zunächst noch zu weit und wird umfassend in Abschnitt 6.1.4, »Der Namensraum `std`«, erläutert. Aus diesem Grund verwenden wir bis auf Weiteres `std::`.

Wenn Sie absoluter Einsteiger in eine Programmiersprache sind, dann sind hier schon Begriffe wie Funktion, Rückgabewert, Operator usw. gefallen. Alles gleich hier an Ort und Stelle zu beschreiben, macht jetzt noch keinen Sinn. All diese Begriffe werden im Verlaufe des Buches noch hinreichend behandelt.

1.5.1 Das Programm bei der Ausführung

Wenn Sie ein Programm übersetzt und eine ausführbare Datei daraus gemacht haben, liegt diese Datei im Maschinencode vor. Starten Sie die ausführbare Datei, wird sie zunächst in den Arbeitsspeicher des Rechners geladen, ehe der Prozessor das Programm Anweisung für Anweisung abarbeitet. In Abbildung 1.14 bis Abbildung 1.16 sehen Sie das eben geschriebene Programm bei der Ausführung unter verschiedenen Betriebssystemen.

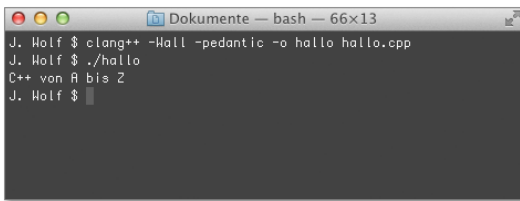


Abbildung 1.14 Das Programm bei der Ausführung auf einem Mac OS X in der Kommandozeile, übersetzt mit clang++

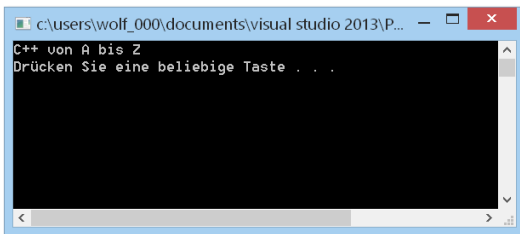


Abbildung 1.15 Das Programm bei der Ausführung unter MS Windows, übersetzt mit Visual Studio 2013 Express Edition

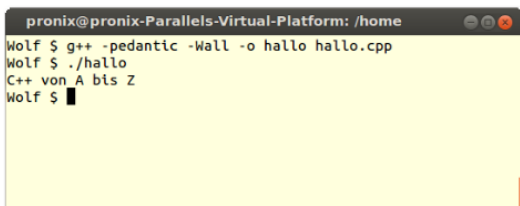


Abbildung 1.16 Das Programm bei der Ausführung auf einem Ubuntu-Linux-System, übersetzt wurde es mit dem g++-Compiler.

Bei einigen Entwicklungsumgebungen wird bei der Ausführung von Konsolenprogrammen nicht immer ein neues Konsolenfenster geöffnet, sondern eine in die Entwicklungsumgebung integrierte Ausgabe verwendet.

Schnell aufpoppende Fenster mit Windows

Wenn Sie das Programm bei einigen Entwicklungsumgebungen ausführen, gerade unter Windows, kann es sein, dass Sie nur ein kurz aufpoppendes Fenster sehen, welches gleich wieder geschlossen wird. Im Grunde ist allerdings nichts Falsches daran. Sie starten das Programm, es öffnet sich eine Konsole, das Programm wird ausgeführt, und die Konsole schließt sich wieder. In diesem Fall müssen Sie lediglich einen Befehl kurz vor der Beendigung des Programms einbauen, welcher das Programm kurz anhält, damit sich die Konsole nicht gleich wieder schließt.

Sehr häufig anzutreffen ist hierbei der MS-DOS-Befehl `PAUSE`, der ein Programm so lange aufhält, bis eine Taste betätigt wird. Sie können diesen Befehl mit der C-Funktion `system()` absetzen, welche in der Headerdatei `<cstdlib>` enthalten ist. Wohlgemerkt, hierbei handelt es sich nur um eine **Improvisation** und keine spezielle C++-Technik. Hierzu eine mögliche Lösung, damit das Fenster nicht gleich wieder geschlossen wird:

```
00 // kapitel01/hallo_pause.cpp
01 #include <iostream>
02 #include <cstdlib>

03 int main()
04 {
05     std::cout << "C++ von A bis Z" << std::endl;
06     system("PAUSE"); // nur für Windows; nicht portabel
07     return 0;
08 }
```

`cin.get()` oder `std::cin >> ch`

Gelegentlich findet man hier auch `cin.get()` oder `std::cin >> ch` als Lösungsvorschlag, aber auch dieser Weg ist nicht immer befriedigend. In der späteren Praxis werden Sie ja ohnehin kaum Programme schreiben, welche eben mal schnell etwas auf dem Bildschirm ausgeben und sich wieder beenden.

1.5.2 Kommandozeilenargumente an main() übergeben

Es gibt noch eine zweite mögliche Version einer `main`-Funktion die Sie verwenden können, wenn Sie Argumente von der Kommandozeile gleich beim Start an das Pro-

programm übergeben wollen. In der Regel macht eine solche `main`-Funktion nur dann Sinn, wenn das Programm auch von einer Kommandozeile gestartet wird. Ich will Ihnen ein Beispiel hierzu nicht vorenthalten:

```
00 // kapitel01/argumente.cpp
01 #include <iostream>

02 int main(int argc, char **argv)
03 {
04     for(int i=0; i<argc; ++i)
05     {
06         std::cout << argv[i] << std::endl;
07     }
08     return 0;
09 }
```

Das Beispiel dient nur der Demonstration, wie Sie Argumente aus der Kommandozeile verwenden können, welche Sie beim Programmstart übergeben haben. Hier wurde auf einige Sachen vorgegriffen, die erst etwas später im Buch behandelt werden. Trotzdem soll der Code hier ein wenig kommentiert werden.

In Zeile 02 finden Sie bei der `main`-Funktion zwischen den runden Klammern jetzt zwei Funktionsparameter. Der erste Parameter ist vom Typ `int` und enthält die Anzahl der Argumente (`argc = argument count`; Argumentzähler), die Sie, durch ein Leerzeichen getrennt, beim Aufruf an das Programm übergeben haben. Der Wert ist immer mindestens 1, weil der Name des ausführbaren Programms mitgezählt wird. Die eigentlichen Argumente sind dann im zweiten Parameter (`argv = argument values`; Argumentwerte), einem zweifachen indirekten Zeiger auf ein Array von Zeichenketten, enthalten. Das erste Argument (`argv[0]`) enthält immer den Namen des auszuführenden Programms. Achtung, hier wird mit 0 beim Zählen angefangen! In Zeile 04 wird dann in einer Schleife Argument für Argument durchlaufen, dessen Inhalt mit jeder Schleifenerhöhung in Zeile 06 ausgegeben wird. Das Programm bei der Ausführung sehen Sie in Abbildung 1.17.

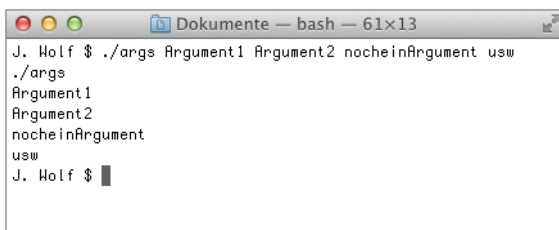


Abbildung 1.17 Dem Programm wurden hier mit »Argument1«, »Argument2«, »nochkeinArgument« und »usw« vier Argumente übergeben, welche alle (inklusive des Namens des auszuführenden Programms) Zeile für Zeile ausgegeben werden.

1.5.3 Quellcode kommentieren

Wenn der Quellcode umfangreicher oder komplizierter wird, macht es Sinn, diesen zu kommentieren. In C++ können Sie den Code auf zwei verschiedene Arten kommentieren. Wenn sich der Quelltext über eine Zeile erstreckt, können Sie die Zeichenfolge `//` verwenden. Alles was sich hinter der Zeichenfolge `//` befindet, wird vom Compiler ignoriert. Ein Beispiel zu `//`:

```
// Ich bin ein Kommentar
cin.get(); // Wartet auf die Eingabe von Enter
```

Kommentare, die sich über mehrere Zeilen erstrecken, leiten Sie mit der Zeichenfolge `/*` ein und beenden diese mit der Zeichenfolge `*/`. Alles zwischen der Zeichenfolge `/*` und `*/` wird ebenfalls vom Compiler ignoriert. Auch hierzu ein Beispiel:

```
/* Dies ist ein Kommentar,
   der sich über mehrere
   Zeilen erstreckt */
```

1.5.4 Programmierstil

Ein paar Worte sollen noch zum Programmierstil verloren werden. Die Rede ist von der Formatierung des Quellcodes. Denn im Grunde können Sie den Code eintippen, wie Sie wollen, solange Sie die Syntax und Regeln von C++ einhalten (bspw. eine Anweisung mit einem Semikolon abschließen). Sie können quasi mehrere Anweisungen in eine Zeile oder zwischen die geschweiften Klammern setzen. Also Folgendes ist ohne Weiteres möglich:

```
#include <iostream>
int main(int argc, char **argv){
    std::cout << "C++ von A bis Z" << std::endl; return 0;}
```

Aber solch ein Code ist recht mühsam, zu lesen, und im Fall eines Fehlers ist es auch schwieriger, diesen zu lokalisieren. Aus diesem Grund empfehle ich Ihnen, den Code sauber zu formatieren. Hierzu gehört ein gleichmäßiges Einrücken innerhalb eines Anweisungsblocks. Für das schnellere Auffinden von Fehlern empfehle ich Ihnen außerdem, nur eine Anweisung pro Zeile zu verwenden. Wenn der Fehler nämlich nur ein einfacher Syntaxfehler ist, gibt der Compiler auch die Zeilennummer mit dem Fehler aus.

1.6 Die einfachen Streams für die Ein-/Ausgabe

Bevor Sie richtig in C++ einsteigen können, sollen hier noch kurz die grundlegenden Streams für die Ein-/Ausgabe von Tastatur und Bildschirm erläutert werden. Um

überhaupt eine Form der Ein-/Ausgabe in C++ verwenden zu können, benötigen Sie die `iostream`-Bibliothek. Das ist auch der Grund, warum Sie in fast jedem Programm die Headerdatei `<iostream>` mit angeben müssen.

iostream-Bibliothek

Das `iostream`-Konzept von C++ ist weitaus umfangreicher, als dies hier zunächst den Anschein hat. In diesem Abschnitt geht es primär nur darum, Ihnen kurz zu beschreiben, womit Sie eine einfache Eingabe von der Tastatur und einfache Ausgaben auf dem Bildschirm realisieren können. Tiefgründiger behandelt wird das Thema in Kapitel 13, »Die Stream-Ein-/Ausgabeklassen von C++«, und auch weitere Streams werden hier vorgestellt.

Für die einfache Ein- und Ausgabe von Tastatur und Bildschirm von der `iostream`-Bibliothek stehen Ihnen die in Tabelle 1.5 aufgelisteten Stream-Objekte zur Verfügung (Abbildung 1.18). Alle Stream-Objekte befinden sich im Namensraum `std`, weshalb Sie diesen Namensraum auch mit angeben müssen.

Objekt	Bedeutung	Ziel
<code>cout</code>	Standardausgabe	Bildschirm
<code>cerr</code>	Standardfehlerausgabe	Bildschirm
<code>clog</code>	Standardfehlerausgabe (gepuffert)	Bildschirm
<code>cin</code>	Standardeingabe	Tastatur

Tabelle 1.5 Stream-Objekte für die einfache Ein- und Ausgabe

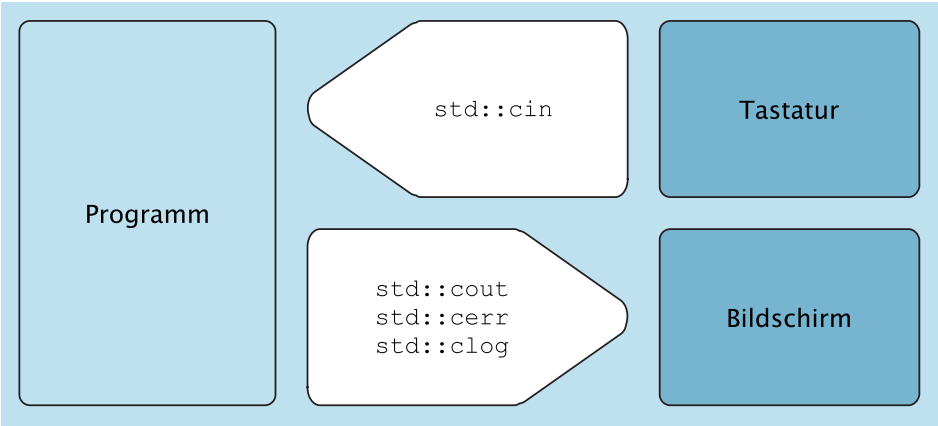


Abbildung 1.18 Grafische Übersicht der einfachen Ein- und Ausgabe-Streams in `iostream`

Stream

Der Begriff *Stream* lässt sich ungefähr als Datenstrom eindeutschen (was wir hier aber nicht machen wollen). Sie können sich einen solchen Stream wie einen Strom von Daten vorstellen, der durch einen Tunnel fließt und durch den die einzelnen Bytes fließen. Neben den einfachen Streams von der Tastatur oder zum Bildschirm gibt es noch weitere Streams, wie bspw. für Dateien oder Strings.

1.6.1 Ausgabe auf dem Bildschirm

Der folgende Code soll Ihnen die drei Stream-Objekte `cout`, `cerr` und `clog` für die einfache Ausgabe auf dem Bildschirm zeigen:

```
00 // kapitel01/ostreams.cpp
01 #include <iostream>
...
02 std::cout << "Standardausgabe" << std::endl;
03 std::cerr << "Standardfehlerausgabe - 1" << std::endl;
04 std::clog << "Standardfehlerausgabe - 2" << std::endl;
```

Ohne die Headerdatei aus Zeile **01** geht hier gar nichts. Mit Zeile **02** wird über das Objekt `cout` eine Ausgabe auf dem Bildschirm erzeugt. Mit `cerr` in Zeile **03** wird die Ausgabe auf die Standardfehlerausgabe bewirkt. Dies ist sinnvoll, wenn die Standardausgabe (über `cout`) nicht mehr zur Verfügung steht oder ein Fehler aufgetreten ist. Die letzte Ausgabe in Zeile **04** über `clog` ist weniger geläufig und wird gewöhnlich zur Ausgabe von einfachen Kontrollmeldungen verwendet. Im Grunde ist `clog` wie `cerr`, nur wird mit `clog` die Ausgabe gepuffert. Bei `cerr` ist die Ausgabe nicht gepuffert.

Pufferung (extrem vereinfachte Beschreibung)

Aus Effizienzgründen ist der Datenverkehr gepuffert. Das bedeutet, dass so lange Daten in einen Puffer gelegt werden, bis dieser voll ist. Erst wenn der Puffer voll ist, geht der Datenfluss weiter an das andere Ende des Streams. Natürlich gibt es auch Möglichkeiten, den Puffer manuell zu entleeren und die Daten gleich weiterzugeben, aber darauf wird hier noch nicht eingegangen. Würde überhaupt keine Pufferung verwendet, dann würde, vereinfacht ausgedrückt, Byte für Byte über einen Stream auf dem Bildschirm ausgegeben oder in eine Datei geschrieben. Wenn alle Prozesse auf Ihrem System ungepuffert arbeiten würden, dann wäre der Rechner ständig mit der Ein-/und Ausgabe von Daten beschäftigt.

Im Falle eines Programmfehlers kann allerdings nicht mehr gewartet werden, bis der Puffer voll ist und dann entleert wird (häufig kann der Puffer gar nicht mehr geleert werden). Dafür werden dann nicht gepufferte Streams wie `cerr` benötigt, um sofort, ohne irgendwelchen Pufferentleerungen, etwas auf dem Bildschirm auszugeben.

Zeilenende mit `endl` oder `\n`

Da viele Leser gerne mehrere Bücher oder Online-Tutorials parallel mit diesem Buch verwenden, kann es sein, dass Sie oft für die Ausgabe einer neuen Zeile entweder `std::endl` oder das Steuerzeichen `\n` vorfinden. Ich will Ihnen hier die vier Möglichkeiten zeigen, die Sie damit haben. Welche davon Sie dann verwenden, überlasse ich Ihnen. Hier die vier Möglichkeiten:

```
00 // kapitel01/neuzeile.cpp
...
01 std::cout << "Mit Manipulator" << std::endl;
02 std::cout << "Als Zeichenliteral" << '\n';
03 std::cout << "Als Stringliteral" << "\n";
04 std::cout << "Eingebettet\n";
...
```

In Zeile **01** wurde der Manipulator `std::endl` verwendet. Intern stellt dies allerdings die *langsamste* Version dar, weil hierbei auch noch eine extra Synchronisation des Puffers ausgeführt wird. Die Version als Zeichenliteral (zwischen einzelnen Anführungszeichen) in Zeile **02** ist die *günstigste* Version, weil lediglich ein Zeichen verwendet wird. In Zeile **03** wurde das Zeichen als Stringliteral (zwischen doppelten Anführungszeichen) verwendet. Bei einem Stringliteral wird noch ein zusätzliches String-Endezeichen verwendet (also insgesamt zwei Zeichen). In der letzten Zeile (**04**) sehen Sie die *noch günstigere* Lösung (gegenüber dem Zeichenliteral), in der das Steuerzeichen `\n` im String am Ende eingebettet wurde. Bei der letzten Möglichkeit kann somit auch auf einen zweiten Aufruf des Ausgabeoperators `<<` verzichtet werden.

Die Crux mit den Umlauten unter Windows

Spätestens wenn Sie Programme mit Microsoft Visual Studio oder Ähnlichem übersetzen und Umlaute in Ihrem Listing zur Ausgabe verwendet haben, werden Sie sich sicherlich ärgern, dass in der sich öffnenden Windows-Eingabeaufforderung die Umlaute nicht richtig dargestellt werden (Abbildung 1.19). Das Problem liegt an unterschiedlichen Codepages, welche in der klassischen MS-DOS-Box und in der Windows-Eingabeaufforderung verwendet werden.

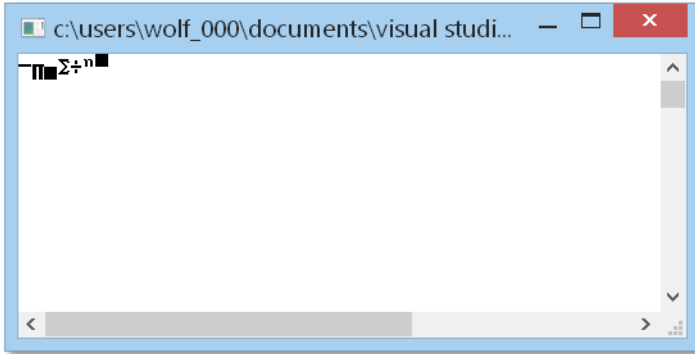


Abbildung 1.19 Der Versuch, die Umlaute »ÄÖÜäöüß« auszugeben, verursacht hier Zeichensalat.

Daher will ich Ihnen zumindest eine klassische Möglichkeit demonstrieren, wie Sie die Umlaute für Ihre Kultur (hier speziell unter Windows) ausgeben können. Die einfachste Möglichkeit dürfte die Klasse `locale` sein, welche sich ebenfalls im Namensraum `std` befindet und in der Headerdatei `<locale>` enthalten ist. Das folgende sehr einfache Beispiel zeigt, wie Sie das Umlaute-Problem beheben können:

```
00 //kapitel01/umlaute_windows.cpp
01 #include <locale>
...
02 std::locale::global(std::locale("German"));
03 std::cout << "ÄÖÜäöüß" << std::endl;
...
```

Die Headerdatei in Zeile **01** wird für `locale` benötigt. In Zeile **02** stellen wir die Umgebung für die laufende Anwendung auf "German". Jetzt sollte es unter Windows auch mit der Ausgabe von Umlauten in Zeile **03** klappen (Abbildung 1.20).

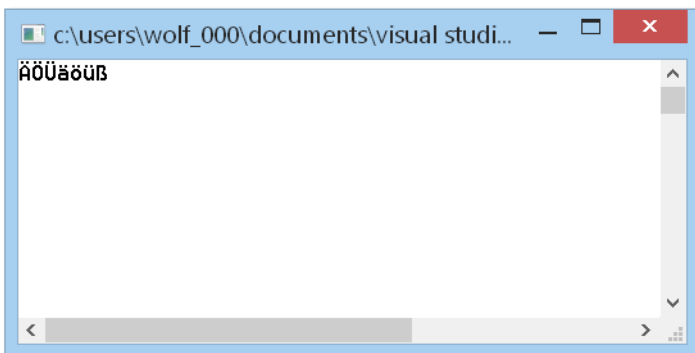


Abbildung 1.20 Nach der entsprechenden Einstellung mit »locale« klappt es jetzt auch unter Windows mit den Umlauten.

Bitte beachten Sie, dass das Setzen von `locale` systemabhängig ist. So wird das Beispiel auf einem Unix-System wohl eine Ausnahme (*Exception*) werfen. Die aktuell gesetzte Klasse `locale` müssen Sie hier aus der Umgebungsvariablen `LANG` ermitteln. Unter Unix-artigen Systemen wird hierfür häufig "`de_DE.UTF-8`" verwendet. Aber in der Regel hat eine Linux- bzw. Unix-Konsole eher selten Probleme mit den Umlauten, wenn die Umgebung entsprechend konfiguriert ist.

1.6.2 Einlesen von der Tastatur

Zum Einlesen von der Tastatur können Sie das Stream-Objekt `cin` verwenden. Die Verwendung ist hierbei denkbar einfach, wie folgender Codeausschnitt demonstrieren soll:

```
00 //kapitel01/WertEinlesen.cpp
...
01 int eineGanzzahl = 0;
02 std::cout << "Eine Ganzzahl bitte: ";
03 std::cin >> eineGanzzahl;
04 std::cout << "Ihre Eingabe: " << eineGanzzahl
    << std::endl;
```

In Zeile **01** finden Sie eine Variable `eineGanzzahl` vom Datentyp `int` vor, welche eine Ganzzahl speichern kann. Nach einer Eingabeaufforderung in Zeile **02** können Sie in Zeile **03** über die Tastatur eine Ganzzahl eingeben. Mit  bestätigen Sie die Eingabe. Die Eingabe wird vom Stream-Objekt `cin` über den Eingabeoperator `>>` in der Variablen `eineGanzzahl` gespeichert. In Zeile **04** wird der zuvor eingelesene Wert wieder auf dem Bildschirm ausgegeben (Abbildung 1.21).

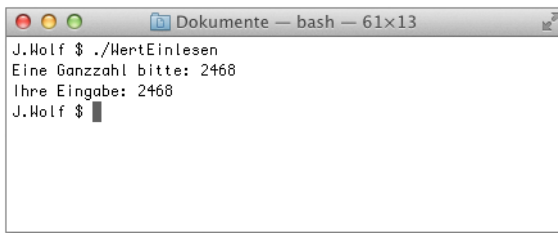


Abbildung 1.21 Einlesen von der Tastatur

1.6.3 Die Ein- und Ausgabeoperatoren `>>` und `<<`

Bei der Verwendung der Ausgabe-Streams `cout`, `cerr` und `clog` bzw. des Eingabe-Streams `cin` müssen Sie den Ausgabeoperator `<<` bzw. den Eingabeoperator `>>` verwenden. Beide Operatoren sind so implementiert (genauer überladen), dass Sie hiermit die grundlegenden Basisdatentypen ausgeben bzw. einlesen können, ohne

besondere Vorkehrungen treffen zu müssen. Unnötig zu erwähnen, dass der Ausgabeoperator << nur für Ausgabe-Streams und der Eingabeoperator >> nur für den Eingabe-Stream verwendet werden kann.

1.7 Zusammenfassung

In diesem Kapitel wurden jetzt viele (vor allem kleinere) Themen behandelt, die Ihnen dabei helfen sollen, in den folgenden Kapiteln die abgedruckten Listings problemlos oder möglichst reibungslos übersetzen und ausführen zu können. Um mit dem Buch fortfahren zu können, sollten Sie jetzt in der Lage sein, ein Listing mit dem Compiler bzw. der Entwicklungsumgebung Ihrer Wahl zu übersetzen und auszuführen.

Sie kennen mit der `main()`-Funktion das Hauptprogramm und wie das Grundgerüst auszusehen hat. Des Weiteren wissen Sie jetzt auch, womit Sie etwas auf dem Bildschirm ausgeben (`cout`, `cerr` und `clog`) und von der Tastatur einlesen (`cin`) können.

Kapitel 2

Die Basisdatentypen in C++

Bevor Sie sich höheren und komplexeren Themen der C++-Programmierung nähern können, müssen Sie zunächst die eingebauten Basisdatentypen von C++ und den Umgang damit kennenlernen.

Datentypen sind grundlegend für die Verwendung von Variablen und Konstanten. Hierbei wird zwischen Typen für Ganzzahlen, Fließkommazahlen, Zeichen und Wahrheitswerten unterschieden. Neben den grundlegenden Datentypen lernen Sie in diesem Kapitel auch den Umgang mit denselben, z.B. für arithmetische Berechnungen, kennen. Ebenfalls erklärt wird, was passiert, wenn diese Typen miteinander vermischt werden: hierbei wird eine Konvertierung durchgeführt. Die verschiedenen Verfahren von Typenumwandlungen werden am Ende des Kapitels erläutert.

2.1 Grundlegendes zu Datentypen

Zum Speichern von Werten benötigen Sie einen **Datentyp** in C++. Es stehen Ihnen mehrere eingebaute Typen (auch primitive Datentypen genannt) zur Verfügung. Hierbei unterscheidet man zwischen arithmetischen Ganzzahltypen (`short int`, `int`, `long int` und `long long int`), Fließkommatypen (oder auch Gleitpunkttypen, `float`, `double` und `long double`) und Typen für Zeichen (`char`, `wchar_t`, `char16_t` und `char32_t`). Mit der Nennung des Datentyps legen Sie auch gleich fest, was Sie damit machen können.

Selbst wenn Sie bspw. nur eine einfache Addition durchführen wollen, muss der Datentyp eine Addition auch unterstützen. Bei den eingebauten Typen mag sich dies etwas trivial und selbstverständlich anhören, aber wenn Sie später eigene Typen (bspw. mit Klassen) erstellen, sind Sie dafür verantwortlich, was bei der Verwendung des `+`-Operators passieren soll, wenn dieser zwischen zwei selbst erstellten Typen steht (Stichwort Operator-Überladung). Aber das geht an dieser Stelle etwas zu weit.

2.1.1 Erlaubte Bezeichner für den Zugriff auf Variablen

Zum Datentyp benötigen Sie einen gültigen Bezeichner (einen Namen für die Variable), um überhaupt auf die Daten im Speicher zugreifen zu können. Hierfür können

Sie beliebige Buchstaben (aber keine landesspezifischen Zeichen wie Umlaute), Ziffern oder das Unterstrichzeichen (bspw. `_bezeichner`) verwenden (Tabelle 2.1). Allerdings ist es nicht erlaubt, dass das erste Zeichen mit einer Ziffer beginnt (somit falsch: `4me`). Auf manchen Compilern kann auch das Dollarzeichen `$` im Bezeichner verwendet werden. Bei den Bezeichnern wird zwischen Groß- und Kleinschreibung unterschieden.

Gültige Bezeichner	Ungültige Bezeichner
<code>_8tung</code>	<code>8tung</code>
<code>_789_</code>	<code>wörter</code>
<code>val_4_you</code>	<code>paragraph\$</code>
<code>NeBeL</code>	<code>ung%ltig</code>

Tabelle 2.1 Beispiele von gültigen und ungültigen Bezeichnern

Reservierte Bezeichner

Auf Bezeichner, die mit zwei sequenziellen Unterstrichen oder einem Unterstrich, gefolgt von einem Großbuchstaben, beginnen, sollten Sie verzichten, weil diese für die C++-Implementierung reserviert sind. Bezeichner wie `value` sind gewöhnlich für Compilerzwecke und `Value` für Betriebssystem- und Bibliothekszwecke gedacht.

Umlaute als Bezeichner?

Zwar können nicht direkt Umlaute wie `üöäÛÖÄ` in einem Bezeichner verwendet werden, aber der Standard erlaubt Universalcharakternamen wie `\UXXXXXXX` oder `\uXXXX`. Folgender Code wäre somit standardkonform:

```
int sch\u00f6n = 5; // nach C++-Standard erlaubt
std::cout << sch\u00f6n << std::endl;
```

Nicht standardkonform hingegen ist die Verwendung von Umlauten direkt im Bezeichner, wie:

```
int schön = 5; // Fehler! Nicht erlaubt nach C++-Standard
std::cout << schön << std::endl;
```

Allerdings hat dieses Features mit den Universalcharakternamen den Nachteil, dass es nur sehr wenige Compilerhersteller implementiert haben. `g++` hat etwa das Beispiel mit dem Universalcharakternamen ohne Beanstandungen übersetzt. Die meisten

anderen Compiler haben den Universalcharakternamen aber nur »kopfschüttelnd« abgelehnt.

Guter Bezeichner vs. schlechter Bezeichner

Sie wissen zwar nun, wie ein gültiger Bezeichner aussehen darf, und können unter Einhaltung dieser Regeln den Bezeichner nennen, wie Sie wollen. Trotzdem möchte ich Ihnen hier einige Ratschläge ans Herz legen:

- ▶ Verwenden Sie **konsistente Namen**. Verwenden Sie bspw. für eine Zählvariable entweder `cn`, `counter`, `cntr`, `nr` oder `n`, aber nicht mehrere nebeneinander. Ebenso sollten Sie die Bezeichner entweder auf Englisch (geläufiger) oder Deutsch schreiben. Ein Mischen der Sprachen wirkt sehr inkonsistent. Achten Sie außerdem möglichst auf Rechtschreibfehler, und vermeiden Sie Bezeichner, die ähnlich lauten (bspw. `color` und `colour`).
- ▶ Verzichten Sie auf unnötig lange Bezeichner, weil sich diese schwer merken lassen und zu Fehlern einladen. `einlangerundgueltigerBezeichner` ist für meinen Geschmack schon etwas zu lang.
- ▶ Bezeichner wie `0` (Buchstabe O) und `1` (Buchstabe L) sollten Sie vermeiden, weil diese leicht mit `0` (Null) und `1` (Eins) zu verwechseln sind.

2.1.2 Deklaration und Definition

Die Begriffe *Deklaration* und *Definition* werden in der Informatik im Allgemeinen gerne in einem Atemzug genannt und irrtümlicherweise gleichgesetzt. In diesem Abschnitt sollen diese beiden Begriffe etwas voneinander differenziert werden. Dieses Missverständnis rührt leider oft daher, weil man beim »Erstellen« einer Variablen wie folgt eben von einer Deklaration und Definition spricht (was wohlgemerkt nicht falsch ist):

```
// eine Variable erstellen
Datentyp Bezeichner1;
// mehrere Variablen erstellen
Datentyp Bezeichner2, Bezeichner3, Bezeichner4;
```

Von einer **Deklaration** ist gewöhnlich die Rede, wenn Größe, Bezeichner, Datentyp und weitere Attribute einer Variablen (oder – später im Buch – einer Funktion) festgelegt werden. Mit einer Deklaration machen Sie quasi zunächst nur die Variable (oder – später im Buch – auch die Funktion) mit dem Compiler bzw. Linker bekannt. Damit ist es möglich, dass Sie diese Variable oder Funktion an einer anderen Stelle im Quelltext verwenden können.

Die **Definition** ist eigentlich nur ein spezieller Fall einer Deklaration. Bei einer Variablen ist die Rede von einer Definition, wenn der Compiler bzw. Linker einen Code erzeugen und für die Variable(n) einen Speicherplatz reservieren soll, womit die Variable (bzw. Funktion) eine eindeutige Speicheradresse besitzt. Ob der Speicherplatz hier vom Datensegment (statisch) oder zur Laufzeit (dynamisch) reserviert wird, ist dabei egal. Bei Funktionen ist erst die Rede von einer Definition, wenn der Quelltext der Funktion (zwischen den geschweiften Klammern) geschrieben wird.

Eine Variable wirklich nur deklarieren

Bei einer Funktion haben Sie eine Nur-Deklaration, wenn Sie eben nur den Prototyp (Funktionskopf) ohne den eigentlichen Code verwenden (dazu mehr in Abschnitt 5.3, »Funktionen deklarieren (Vorausdeklaration)«). Aber auch bei einer Variablen können Sie eine Nur-Deklaration erstellen, indem Sie das Schlüsselwort **extern** davor setzen:

```
// Reine Deklaration  
extern Datentyp Bezeichner;
```

In der Regel bedeutet dies dann auch, dass diese Variable irgendwo definiert werden muss, was man gewöhnlich in einer anderen Übersetzungseinheit macht. Diese Nur-Deklaration von Variablen sei aber hier jetzt nur am Rande erwähnt.

2.1.3 Initialisierung von Variablen

An dieser Stelle geht es jetzt noch nicht um die Initialisierung von Variablen mit dem `=`-Operator (Zuweisungsoperator) oder dem Stream-Objekt `cin`, sondern vielmehr darum, was nach der Deklaration und Definition für ein Wert im Speicherbereich steht. Gewöhnlich steht nämlich ohne weiteres Hinzutun ein beliebiger **undefinierter Zufallswert** in der Variablen. Hier sollten Sie es sich also gleich zur Regel machen, eine Variable **sofort** mit einem Wert zu **initialisieren**. Nur so können Sie vermeiden, dass mit einem undefinierten Zufallswert weitergearbeitet wird, was auch zu undefinierten Ergebnissen führen kann. Gehen Sie daher immer auf Nummer sicher, und initialisieren Sie die Variable gleich bei der Erstellung:

```
Datentyp Bezeichner = 0;  
Datentyp Bezeichner{0}; // Neu in C++11  
Datentyp Bezeichner = {0};
```

Empfehlung für besseren Code: Variablen sofort initialisieren

Um sich vor nicht vorhersehbaren Ergebnissen zu schützen, sollten Sie Variablen immer schon beim Erstellen mit einem gültigen Wert initialisieren.

2.1.4 Vereinheitlichte Initialisierung mit C++11

Mit dem neuen C++11-Standard wurde jetzt (endlich) auch eine vereinheitlichte Initialisierung mit geschweiften Klammern `{...}` eingeführt. Diese Initialisierung kann neben grundlegenden Datentypen später auch bei komplexeren Typen wie Arrays, Strukturen, Klassen oder Behälter-Klassen (Container-Klassen) verwendet werden. Vereinheitlichte Initialisierungen sind frei von Mehrdeutigkeiten und bedeuten bei gleichem Aussehen auch immer dasselbe.

Der Grund für eine vereinheitlichte Initialisierung in C++11 war, dass es zu viele verschiedene Formen von Initialisierungen gegeben hat (und immer noch gibt). Und zu viele unterschiedliche Dinge für denselben Zweck macht eine Programmiersprache eher komplexer. Für Einsteiger in C++ bedeutet diese einheitliche Verwendung jetzt auch eine wesentlich einfachere Verwendung der Sprache, weil kein Gedanke mehr daran verschwendet werden muss, wie eine Initialisierung für bspw. eingebaute Datentypen, Strukturen, Arrays oder Klassen mit Konstruktoren usw. auszusehen hat. Jetzt kann man überall einfach und einheitlich die `{...}`-Syntax verwenden.

Vereinheitlichte Syntax im Buch?

Als Autor steht man vor dem Dilemma, was man für das Buch verwenden soll. Würde ich durchgehend die neue vereinheitlichte Syntax verwenden, könnte es sein, dass die Leser außen vor bleiben, die keinen Compiler mit dem neuen C++11-Standard verwenden. Außerdem gibt es ja auch noch viel mehr Code ohne vereinheitlichte Syntax auf der Welt. Auf der anderen Seite könnte der Aufschrei laut sein, weil ich nicht den neuen C++11-Standard durchgehend behandle. So habe ich mich dafür entschieden, die entsprechenden Codezeilen im Buch mit einem Kommentar zu versehen

Hierzu nochmals ein theoretischer Code zu vereinheitlichten Initialisierung mit C++11:

```
Datentyp Bezeichner1{Wert};           // Vereinheitlichte Initialisierung
Datentyp Bezeichner2{Bezeichner1};   // auch erlaubt
Datentyp Bezeichner3 = {Wert};        // das = ist hier optional
```

Empfehlung für besseren Code: Vereinheitlichte Initialisierung verwenden

Wenn es möglich ist und Sie sich sicher sein können, dass Sie den C++11-Standard verwenden und verwenden können, sollten Sie die vereinheitlichte Syntax bevorzugen.

2.2 Ganzzahldatentypen (Integer-Datentypen)

C++ bietet eingebaute Ganzzahldatentypen (auch Integer-Datentypen oder integrale Typen genannt) in verschiedenen Größen an. In aufsteigender Größe sind diese

short int, int, long int und long long int. Die Versionen short, long und long long können Sie auch ohne den Namen int verwenden.

Gültige (Ganzzahl-)Literele für Integer-Typen, können wie folgt aussehen:

-1234 -99 0 456 39543 +2333 +1001

Zu diesen Zahlen muss noch Folgendes angemerkt werden:

- Positive Integer-Typen benötigen kein +-Zeichen davor. Das heißt, diese sind auch ohne das +-Zeichen positiv.

long long int (C++11)
Der extra lange Ganzzahltyp long long int wurde offiziell erst mit dem C++11-Standard eingeführt.

2.2.1 Regeln für gültige Ganzzahlen

Bevor Sie Ganzzahlen an Variablen übergeben können, müssen hier noch kurz die Regeln beschrieben werden, welche Zeichenfolgen Sie verwenden müssen, damit die Ganzzahl auch für den Compiler gültig ist. Neben der gängigeren dezimalen Schreibweise (Basis 10) können Sie auch eine oktale (Basis 8) und hexadezimale (Basis 16) Schreibweise verwenden. In Tabelle 2.2 finden Sie einige Beispiele gültiger Schreibweisen (auch Literale genannt) aufgelistet.

Dezimal	Oktal	Hexadezimal
0	00	0x0
1	01	0x1
10	12	0xa bzw. 0xA
98	0142	0x62

Tabelle 2.2 Einige dezimale Beispiele mit oktalen und hexadezimalen Gegenstücken

Oktal und hexadezimal
Oktale Zahlen beginnen mit **0** und die hexadezimale Schreibweise mit **0x** am Anfang. Bei der hexadezimalen Schreibweise werden die Buchstaben a, b, c d, e, f (bzw. die gleichen Großbuchstaben) verwendet, um 10, 11, 12 13, 14 und 15 darzustellen.

2.2.2 Ganzzahlen mit Werten initialisieren

Ganzzahlen lassen sich ganz einfach mit der einheitlichen {...}-Syntax oder dem Zuweisungsoperator = initialisieren. Ebenso können Sie die Werte mithilfe des Stream-Objekts cin und den Eingabeoperatoren in die Variable schieben. Hierzu einige Beispiele:

```
00 // kapitel02/ganzzahlen.cpp
01 short sval1{0};      // C++11
02 short sval2;
03 int ival1 = {0144};  // 0144 = 100
04 int ival2{ival1};    // C++11
05 long lval1 = 0xff;   // 0xff = 255
06 long lval2 = lval1;
07 long long llval{0};  // C++11
08 std::cout << "Einen long long-Wert bitte: ";
09 std::cin >> llval;
```

Fehler beim Übersetzen (C++11): Zuweisung via {} oder = ?

Sollten Sie Probleme bei der Übersetzung von Listing *ganzzahlen.cpp* haben, kann es sein, dass Ihr Compiler den C++11-Standard leider noch nicht unterstützt (oder er nicht aktiviert ist). In dem Fall müssen Sie die Zuweisungen mit der neuen vereinheitlichten {...}-Syntax zurück auf den klassischen Zuweisungsoperator = stellen. Für Zeile **01** würde dies daher bspw. bedeuten, dass Sie diese wie folgt ändern müssten:

```
short sval1 = 0;
```

Gleiches müssten Sie dann hier auch mit den Zeilen **03**, **04** und **07** machen. Die Entwicklungsumgebung von Microsoft (ab Visual Studio 2013) unterstützt die vereinheitlichte Initialisierung, ebenso Entwicklungsumgebungen wie NetBeans, Eclipse, Qt Creator oder selbst Dev-C++ 5.3.x (von Orwell), mit dem MinGW ab 4.6 können Sie dort die neueren C++11-Features nutzen, indem Sie gegebenenfalls bei den Compiler-Flags die Option -std=c++11 setzen. Mit dem GCC (getestet wurde hier mit den Versionen 4.7 und 4.8) und Clang klappte es ebenfalls ohne Probleme.

In Zeile **01** wird die Variable sval1 vom Typ short mit dem Wert 0 initialisiert. Die Zeile **02** zeigt, dass Variablen nicht unbedingt sofort initialisiert werden müssen, aber ich empfehle Ihnen, dies gewöhnlich immer zu tun (siehe Abschnitt 2.1.3, »Initialisierung von Variablen«). In Zeile **03** sehen Sie, wie Sie einen oktalen Wert an die Variable ival1 vom Typ int zuweisen können. Hier verwenden wir außerdem zusätzlich noch das = vor {...}, was der einheitlichen Initialisierungssyntax entspricht und so ebenfalls möglich ist. Gleiches machen wir dann auch in Zeile **04**, wo wir die Vari-

able `ival2` mit dem Wert von `ival1` initialisieren. Selbiges geschieht auch in den Zeilen **05** und **06**, nur mit einem hexadezimalen Wert und hier mit dem klassischen Zuweisungsoperator. In Zeile **09** sehen Sie außerdem, wie Sie über das Stream-Objekt `cin` selbst einen Wert an die Variable `lval` von Zeile **07** über die Tastatur interaktiv eingeben können.

2.2.3 Positive oder negative Ganzzahlen

Ohne weitere Angabe können die Ganzzahldatentypen sowohl positive als auch negative Werte bis zu einer für den Typ bestimmten Größe speichern. Benötigen Sie allerdings einen vorzeichenlosen Ganzzahltyp, müssen Sie jeweils das Schlüsselwort `unsigned` vor den Typ stellen.

```
00 //kapitel02/vorzeichen.cpp
01 unsigned int ival1={0};
02 int ival2={0};
03 signed int ival3={0};
```

Mit dem Voransetzen von `unsigned` in Zeile **01** kann die Variable `ival1` vom Typ `int` keine negativen Werte mehr speichern. Standardmäßig ohne besondere Angaben wie in Zeile **02** können positive und negative Werte gespeichert werden. Es ist zwar auch möglich, mit dem Schlüsselwort `signed` den Typ explizit als vorzeichenbehafteten Typ zu kennzeichnen, wie dies in Zeile **03** gemacht wurde, aber darauf können Sie auch meistens **verzichten** (abgesehen von `char`, siehe Abschnitt »Vorzeichen vom Datentyp `char`« in Abschnitt 2.4.1), weil ganzzahlige Typen ohne Verwendung von `unsigned` immer vorzeichenbehaftet wie in Zeile **02** sind.

Konstanten (Literale) mit den Suffixen U und L kennzeichnen

Eine ganzzahlige Konstante wie `123456` ohne weitere Angaben ist stets vom Typ `int`. Wenn Sie am Ende der Ganzzahl ein `u` bzw. `U` anhängen, gilt dieses Literal explizit als `unsigned int`. Hängen Sie hingegen ein `l` bzw. `L` ans Ende, wird diese Konstante explizit als `long int` gekennzeichnet. Kombinieren Sie beide miteinander (`ul` bzw. `UL`), wird diese Zahl als `unsigned int long` betrachtet:

```
unsigned int uval = {123456U};    // unsigned int Wert
long lval = {0x7FFFFFFL};        // long int Wert
unsigned long ulval = {0777ul};  // unsigned long int Wert
```

Auch für den neu in C++11 eingeführten Datentyp `long long` gibt es mit `ll` bzw. `LL` ein entsprechendes Suffix. Für die `unsigned`-Versionen von `unsigned long long` lautet das Suffix `ull` bzw. `ULL`.

2.2.4 Boolescher Datentyp für die Wahrheit

Eigentlich kein Ganzzahldatentyp im eigentlichen Sinne, aber ich habe diesen hier trotzdem aufgenommen. Der Datentyp `bool` ist darauf spezialisiert, zu prüfen, ob ein Ausdruck wahr (`= true`) oder unwahr (`= false`) ist. Damit werden in der Praxis verschiedene Werte bzw. (bedingte) Ausdrücke auf Wahrheit oder Rückgabewerte getestet.

```
00 //kapitel02/wahrheitswert.cpp
01 int ival1 = 1234;
02 int ival2 = 0;
03 int ival3 = -1;
04 bool bval1 = ival1; // = true
05 bool bval2 = ival2; // = false
06 bool bval3 = ival1 == ival2; // = false
07 bool bval4 = ival1 != ival2; // = true
08 bool bval5 = ival3; // = true
```

In Zeile **04** wird der Ganzzahlwert von `ival1` dem booleschen Typ `bval1` zugewiesen, wodurch der Wert nach `true` konvertiert wurde, weil ein Wert ungleich 0 immer `true` ist. Das Gegenteil wird mit Zeile **05** bewiesen, wo Sie den Typ `ival2` mit dem Wert 0 `bval2` zuweisen. Hierbei wird nach `false` konvertiert, weil 0 immer `false` ist. In Zeile **06** wird der Ausdruck `ival1==ival2` an `bval3` zurückgegeben. Hierbei wird quasi behauptet, dass `ival1` gleich `ival2` ist, was ja nicht stimmt, und somit bekommt `bval3` den Wert `false` zugewiesen. Ähnliches geschieht in Zeile **07**, nur bekommt `bval4` hier tatsächlich ein `true` zugewiesen, weil `ival1!=ival2` (`ival1` nicht gleich `ival2`) stimmt. In Zeile **08** bekommt `bval5` den negativen Wert von `ival3` (`= -1`) zugewiesen. `bval5` erhält trotzdem `true` zugewiesen. Es ist nämlich ein geläufiger Irrtum, dass negative Werte `false` sind. Dem ist nicht so, alle Werte ungleich 0 sind `true`, und nur 0 wird in ein `false` konvertiert.

Wozu bool?

Die Interaktion von `bool` und `int` mag dem Einsteiger an dieser Stelle noch etwas unlogisch erscheinen, aber spätestens wenn Sie mit Logikanweisungen oder Funktionen arbeiten, bekommen diese ihre Daseinsberechtigung.

++boolval bzw. boolval++ – bool inkrementieren (deprecated)

In der Vergangenheit wurde der Operator `++` (Präfix- und Postfixschreibweise) gerne mal zum Inkrementieren auf dem Typ `bool` angewendet, um den Wert so auf `true` zu setzen. Andersherum mit dem `--` (Präfix- und Postfixschreibweise) war dies nicht möglich. Der `--` zum Dekrementieren eines `bool` war unzulässig. Die Version mit Operator `++` wurde mit dem C++11-Standard als *deprecated* markiert und sollte in neuen Projekten nicht mehr verwendet werden.

2.3 Typen für Gleitkommazahlen

Auch bei den Gleitkommatypen (bzw. Gleitpunkttypen oder Fließkommazahlen, engl. *floating point numbers*) stehen Ihnen verschiedene Größen zur Verfügung. Für die einfache Genauigkeit wird `float`, für die doppelte Genauigkeit `double` und für eine erweiterte Genauigkeit `long double` verwendet. Die exakte Genauigkeit hängt vom Compilerhersteller ab.

Komma oder Punkt?

In C++ wird die US-Schreibweise für die Gleitkommazahlen verwendet, und daher wird ein Punkt statt eines Kommas gesetzt.

Hierzu einige Beispiele mit gültigen Initialisierungen von Gleitkommazahlen in C++:

```
00 //kapitel02/gleitkommazahlen.cpp
01 float fval1{123.456}; // C++11
02 float fval2{fval1}; // C++11
03 float fval3 = .336; // 0.336
04 double dval1 = 2.22e-15;
05 long double ldval1 = 1.9e-3f;
06 long double ldval2{0.0}; // C++11
07 std::cout << "Bitte eine Gleitkommazahl: ";
08 std::cin >> ldval2;
09 std::cout << "Ihre Eingabe lautete " << ldval2 << std::endl;
```

In den Zeilen **01** und **02** sehen Sie zwei Möglichkeiten der vereinheitlichten C++11-Initialisierung. Wird der Wert vor dem Komma (Vorkommateil) leergelassen, wird automatisch 0 dafür verwendet, wie Zeile **03** beweisen soll. Sie könnten auch auf den Nachkommateil verzichten, wenn Sie einen Vorkommateil verwenden.

Zuweisung ohne einen Punkt

Würden Sie auf das Komma (bzw. genauer den Punkt) verzichten, würde intern dem Gleitpunkttyp ein `int`-Literal zugewiesen. Dabei würde eine implizite Umwandlung von `int` zum entsprechenden Gleitkommatyp stattfinden.

In den Zeilen **04** und **05** wird eine Exponential-Zuweisung (bzw. Schreibweise) demonstriert, welche Sie in der Regel bei besonders kleinen (0.0000000123) oder großen (1234500000000.0) Zahlen verwenden sollten. Die wissenschaftliche Exponential-Schreibweise besteht aus der Mantisse, einer Folge von Ziffern und dem Zehner-Exponent. Zwischen der Mantisse und Exponent muss ein `E` oder `e` stehen. In Zeile **08** wird dann noch eine Gleitkommazahl über das Stream-Objekt `std::cin` interaktiv über die Tastatur eingelesen und der Variablen `ldval2` zugewiesen.

Ausgabe mit und ohne Exponential-Schreibweise

Bei sehr kleinen und großen Zahlen wird häufig empfohlen, die wissenschaftliche Exponential-Schreibweise zu verwenden, weil es häufig einfacher ist, diese (fehlerfrei) einzutippen. Wer allerdings bei der Ausgabe nicht die Exponential-Schreibweise haben will, für den gibt es einen einfachen Manipulator. Auf die Manipulatoren wird im Detail noch gesondert in Abschnitt 13.5.1, »Manipulatoren«, eingegangen. Für die Ausgabe über das Stream-Objekt `cout` können Sie für diese Zwecke die Manipulatoren `fixed` und `scientific` verwenden. Diese Manipulatoren befinden sich ebenfalls im Namensraum `std`. Mit `scientific` erzwingen Sie die Ausgabe der Gleitpunktzahlen in der Exponential-Schreibweise. Das Gegenteil erreichen Sie hingegen mit dem Manipulator `fixed`. Hier ein Beispiel dazu:

```
00 //kapitel02/exponent.cpp
01 double dval1 = {3.14159265358979323};
02 double dval2 = {6.0234567e17};

03 std::cout << std::scientific;
04 std::cout << dval1 << std::endl; //=3.141593e+00
05 std::cout << std::fixed;
06 std::cout << dval2 << std::endl; //=602345670000000000.00...
```

In Zeile **01** wurde die Variable `dval1` mit einer sehr langen Gleitkommazahl initialisiert. Wollen Sie diesen Wert jetzt in der kürzeren Exponential-Schreibweise betrachten, müssen Sie nur den Manipulator `scientific` über das Stream-Objekt schicken, wie dies in Zeile **03** geschieht. Die nächste Ausgabe in Zeile **04** wird dann in der Exponential-Schreibweise durchgeführt.

Umgekehrt ist dies auch möglich. In Zeile **02** wurde bspw. eine Gleitkommazahl in der Exponential-Schreibweise angegeben. Um diesen Wert in einer Gleitkommazahl ohne Exponential-Schreibweise auszugeben, brauchen Sie nur den Manipulator `fixed`, wie in Zeile **05** zu sehen, an das Stream-Objekt `cout` zu schicken, und in der nächsten Zeile (**06**) wird die lange Schreibweise ohne die Exponential-Darstellung verwendet.

Konstanten (Literele) mit den Suffixen F und L kennzeichnen

Der bevorzugte Typ vom Compiler bei Gleitkommaliteralen ist immer `double`. Aber auch hier können Sie durch Anhängen eines Buchstabens explizit den Typ festlegen. Fügen Sie `f` bzw. `F` ans Ende der Konstante, wird das Literal explizit als `float` verwendet. Wollen Sie hingegen eine Konstante als `long double` kennzeichnen, müssen Sie `l` bzw. `L` an das Ende hängen:

```
float fval1 = {3.141592653589F}; // float Wert
long double ldval2 = {6.0234567e17L}; // long double Wert
```

2.4 Typ(en) für Zeichen

Gleich vorneweg: Die Sache mit der Darstellung von Zeichen auf einem Rechner ist gar nicht so trivial, wie man dies vielleicht auf den ersten Blick meinen würde. Also seien Sie gewarnt. Auch will ich hier noch erwähnen, dass in diesem Abschnitt vorerst nur die Darstellung einzelner Zeichen behandelt wird. Sind Sie auf der Suche nach Strings, dann sollten Sie zu Abschnitt 4.2, »Strings (Zeichenketten)«, blättern.

2.4.1 Der Datentyp `char`

Der Typ `char` (abgeleitet von engl. *character*, »Zeichen«) ist ein fundamentaler Datentyp, der Zeichen repräsentieren kann. Auch wenn der Typ fähig ist, Ganzzahlen (allerdings in einem kleinen Bereich) zu speichern, wird dieser eigentlich vorwiegend nur zur Zeichendarstellung verwendet. In den meisten Fällen hat ein `char` 8 Bits, womit sich rein theoretisch 256 verschiedene Zeichen ($2^8 = 256$) darstellen lassen.

Welcher Wert dann welchem Zeichen entspricht, hängt vom verwendeten Zeichensatz (Character Sets; Charsets) ab. Wenn Sie bspw. eine `char`-Variable mit dem Wert 65 initialisiert haben, wird dieser Wert anhand des verwendeten Zeichensatzes auf dem System codiert. Wird hierbei der ASCII-Zeichensatz verwendet (meistens der Fall), wäre dieser Wert anhand dieses Zeichensatzes das Zeichen A. Glücklicherweise können Sie das Zeichen auch als Literal (genauer Zeichenliteral) verwenden. Ein solches Zeichenliteral wird zwischen einzelnen Anführungszeichen gesetzt (bspw. `'A'`, `'C'`, `'X'`).

Zeichensatz

In den meisten Fällen können Sie sich zumindest darauf verlassen, dass in den ersten 128 Zeichen (0 bis 127) im (US-)ASCII-Code die Dezimalziffern, die 26 Buchstaben des englischen Alphabets (Groß- und Kleinschreibung) und andere grundlegenden (auch nicht darstellbaren) Zeichen enthalten sind. Leider wird es dann häufig etwas schwieriger, wenn Sie die landestypischen Zeichen darstellen wollen, wie bspw. hierzulande die Umlaute. Hier müssen Sie sich dann mit dem verwendeten Zeichensatz auf Ihrem System auseinandersetzen, oder aber Sie verwenden die universelle Lösung mit UTF-8-Literalen, indem Sie bspw. `locale` verwenden (siehe den Abschnitt »Die Crux mit den Umlauten unter Windows« in Abschnitt 1.6.1).

Hier ein erster Codeausschnitt dazu:

```
00 //kapitel02/zeichen.cpp
01 char cval1 = {'A'};
02 char cval2 = {65};
03 char cval3 = {10};
```

```

04  std::cout << cval1 << cval3;    // = A
05  std::cout << cval2 << cval3;    // = A
06  std::cout << "Bitte ein Zeichen: ";
07  std::cin >> cval3;
08  std::cout << cval3 << " = "
      << static_cast<int>(cval3) << std::endl;

```

In Zeile **01** wird die Variable `cval1` mit dem Zeichenliteral `'A'` initialisiert. Dass dies auch mit Ganzzahlen erlaubt ist, zeigt Zeile **02** mit dem Wert 65, welcher der Variablen `cval2` zugewiesen wird. Gemäß der ASCII-Tabelle entspricht dieser Wert ebenfalls dem Zeichen `'A'`. In Zeile **03** wird der Wert 10 der Variablen `cval3` zugewiesen. Mehr dazu gleich.

Mit Zeile **04** wird das Zeichen `'A'` auf dem Bildschirm ausgegeben. Gleiches wird in den meisten Fällen auch bei der Ausgabe in Zeile **05** passieren. Hier sehen Sie, dass auch im Falle einer Initialisierung mit einer Ganzzahl bei der Ausgabe über `cout` das Zeichen ausgegeben wird, welches dem Wert im Zeichensatz entspricht.

Dass außerdem in den Zeilen **04** und **05** mit der Variablen `cval3` eine neue Zeile ausgelöst wird, liegt daran, dass im ASCII-Zeichensatz der Wert 10 für das Newline-Zeichen `'\n'` steht. In Zeile **07** können Sie zur Demonstration der Variablen `cval3` interaktiv ein Zeichen mit der Tastatur übergeben.

Wollen Sie den tatsächlichen Ganzzahlwert eines `char` ausgeben, sollten Sie eine Typumwandlung mit `static_cast<>` durchführen, wie dies in Zeile **08** demonstriert wurde. Mehr zur Konvertierung von Datentypen erfahren Sie in Abschnitt 2.9, »Konvertieren von Typen«. Statt des `static_cast<>` in diesem Beispiel würde sich auch ein implizite Typumwandlung mit `int ival3=cval;` anbieten.

Zeichenliteral oder Ganzzahlliteral?

Wenn Sie sich jetzt fragen sollten, ob Sie als Literal eine Ganzzahl (bspw. 65) oder eine Zeichen (bspw. `'A'`) verwenden sollen, so würde ich Ihnen die Zeichenliterale empfehlen, weil diese einen Tick portabler und somit zuverlässiger sind. Wenn Sie erwarten, dass sich hinter der Ganzzahl 65 das Zeichen `'A'` verbirgt, dann gehen Sie davon aus, dass auf dem Rechner der ASCII-Zeichensatz läuft, was vielleicht auch in 99,99 % der Fälle richtig sein dürfte.

Aber für den Fall, dass eben das 0,01 % zutrifft und eben nicht der ASCII-Zeichensatz verwendet wird, sind Sie mit dem Zeichenliteral auf der sichereren Seite. Würde nämlich bspw. der EBCDIC-Zeichensatz verwendet, wäre dort der Wert 65 gar nicht (!) belegt.

Vorzeichen vom Datentyp char

Hier muss noch eine wichtige Anmerkung gemacht werden. Da `char` im Grunde auch ein Ganzzahltyp ist, hängt es bei diesem konkreten Typ von den Compilerherstellern ab, ob `char` als `signed` (-128 bis 127) oder als `unsigned` (0 bis 255) implementiert wurde. Somit ein weiterer Grund, `char` nicht für Dezimalzahlen zu verwenden.

Nicht darstellbare Steuerzeichen

An dieser Stelle sollten auch gleich noch die nicht darstellbaren Steuerzeichen beschrieben werden, wovon Sie bspw. das Zeichen `'\n'` zum Ausgeben einer neuen Zeile bereits mehrmals in diesem Buch verwendet haben. Solche Steuerzeichen werden mit einem umgekehrten Schrägstrich eingeleitet (auch *Backslash* genannt), gefolgt von einem Zeichen mit einer festen Bedeutung. Die Steuerzeichen sind auch in der Zeichencode-Tabelle des entsprechenden Zeichensatzes enthalten. Im weitverbreiteten ASCII-Code sind diese Steuerzeichen die Zeichen mit dem Dezimalwert 0 bis 31 und das Zeichen 127. Davon haben aber in der heutigen Zeit nur noch wenige Steuerzeichen eine wirkliche Bedeutung. In Tabelle 2.3 finden Sie eine Übersicht zu verschiedenen nicht darstellbaren Steuerzeichen, welche Sie in C++ verwenden können.

Dezimal	C++	Bedeutung
0	<code>\0</code>	Nullzeichen. Zeichen hat keinen Inhalt. Wird bspw. verwendet, um das Ende einer Zeichenkette zu markieren.
7	<code>\a</code>	Erzeugt ein Tonsignal (Beep).
8	<code>\b</code>	Bewegt den Cursor eine Position zurück.
9	<code>\t</code>	Horizontaler Tabulator, bewegt den Cursor zur nächsten Position (Tabstopp).
10	<code>\n</code>	Zeilenvorschub, bewegt den Cursor zur nächsten Zeile.
11	<code>\v</code>	Vertikaler Tabulator, bewegt den Cursor zu nächsten vordefinierten Zeile.
12	<code>\f</code>	Seitenvorschub, bewegt den Cursor zur nächsten Seite.
13	<code>\r</code>	Wagenrücklauf, bewegt den Cursor zum Anfang der aktuellen Zeile.

Tabelle 2.3 Nicht darstellbare Steuerzeichen im Überblick. Der Dezimalwert entspricht dem Wert in der ASCII-Code-Tabelle.

2.4.2 Unicode-Unterstützung

Jetzt soll auch das Unicode-Thema gesondert behandelt werden. Das Thema Unicode ist sehr umfangreich und kann hier daher nur angerissen werden, weil hierzu auch Kenntnisse jenseits der Programmierung nötig sind. Ich will Ihnen hier nur die C++-Seite und somit die Verwendung von Unicode mit C++ näherbringen.

Wie Sie jetzt wissen, gibt es mit den ersten 128 Zeichen dank des weitverbreiteten ASCII-Zeichensatzes selten Probleme mit den Zeichen. Sobald aber landestypische Zeichen wie Umlaute ins Spiel kommen, gibt es immer wieder Dinge, die fortan nicht klar sind. Zwar wurde hierfür der ASCII-Zeichensatz von 7 auf 8 Bits erweitert, und dieser Zeichensatz wurde dann unter den Bezeichnungen ISO-8859-1, ISO-8859-2 etc. bis ISO-8859-15 zusammengefasst. Die Umlaute finden Sie hierzulande in ISO-Latin-1 zusammengefasst. Allerdings hilft es meistens gar nichts, einfach die Eingabeaufforderung auf ISO-Latin-1 zu stellen, weil es so zu Problemen mit den mittlerweile viel weiter verbreiteten UTF-8-Codierungen für Unicode kommt, welche viele moderne Systeme verwenden. Gerade Einsteiger unter Windows verzweifeln dann bei den ersten Schritten in C++, wenn Sie noch Konsolenprogramme schreiben, dass dort noch aus Kompatibilitätsgründen am alten IBM-PC-Zeichensatz festgehalten wird, in dem die Dezimalwerte der Zeichen wieder einen andere Wert haben als dies mit ISO-Latin-1 der Fall ist.

Erst mit dem neuen C++11-Standard bietet C++ eigentlich eine richtig gute Unicode-Unterstützung an. Zwar war bis dato schon der Datentyp `wchar_t` vorhanden, aber dieser Typ hatte auf manchen Plattformen eine unterschiedliche Breite (mal 16 Bits und mal wieder 32 Bits).

Anwendung von `wchar_t`, bzw. `wchar_t` überhaupt noch verwenden?

Für die Verwendung von Zeichenliteralen für `wchar_t` müssen Sie ein `L` vor das Zeichen stellen. Da `cout` nur `char` als Zeichen ausgeben kann, müssen Sie außerdem noch das Stream-Objekt `wcout` für die Ausgabe bzw. `wcin` für das Einlesen von breiten Zeichen verwenden, bspw.:

```
wchar_t wcval={L'A'};
std::wcout << wcval << std::endl;
std::cout << "Bitte ein Zeichen: ";
std::wcin >> wcval;
std::wcout << wcval << std::endl;
```

Da es aber nun mal leider so ist, dass die Breite von `wchar_t` compilerspezifisch ist, wäre meine Empfehlung, diesen Typ gar nicht erst zu verwenden, um Unicode-Zeichen zu speichern. Ganz besonders dann nicht, wenn Ihr Programm zwischen verschiedenen Compilern portabel bleiben soll. `wchar_t` ist dafür geeignet, compilerdefinierte breite Zeichen zu speichern, welche auf manchen Compilern dann auch tatsächlich Unicode-Zeichen sein **könnten**. Leider hat man sich in der Praxis gerne darauf verlassen, dass schon alles in ein `wchar_t` reinpassen wird, ohne sich wirklich sicher zu sein, dass das Programm auch tatsächlich Unicode-kompatibel ist. Letztendlich führte dies dann nur noch dazu, dass **Encodings** weniger geläufig waren und dann einfach gar nicht mehr verwendet wurden.

Exkurs: Encoding/Codierung

Der Begriff *Encoding* (oder auch *Codierung*) in Verbindung mit Zeichen wird häufig missverstanden. Hierbei geht es ganz einfach darum, dass der Rechner Buchstaben, Zahlen und Sonderzeichen irgendwie abspeichern muss. Der Computer speichert Informationen wie Zeichen, Zahlen oder Sonderzeichen letztlich auch nur mit 0en und 1en. Wenn Sie das Zeichen »a« als solches sehen, dann nur deshalb, weil eine Zeichentabelle bzw. ein Zeichensatz (auch *Character Sets/Charsets*) auf Ihrem System aus einer Bitfolge dieses Zeichen darstellt. Als Encoding wird somit die Regel bezeichnet, nach der die Bits einem Zeichen (mithilfe des Zeichensatzes) zugeordnet werden können.

Einer der ersten und der wohl bekannteste Zeichensatz dürfte ASCII sein. Da ASCII nicht für landestypische Zeichen ausreichte, wurden 15 weitere Zeichensätze unter der Norm ISO 8859 standardisiert und erweitert (bspw. ISO 8859-1, ISO 8859-2 usw.). Die ersten 128 hat man identisch aus ASCII herauskopiert, um kompatibel zu bleiben. Die deutschen Umlaute und die meisten anderen mittel- und westeuropäischen Sonderzeichen waren in den ersten 10 der 15 ISO 8859 vorhanden. Für fernöstliche Sprachen war aber kein Platz mehr in ISO 8859, was maximal mit 256 Zeichen belegt werden konnte. Dafür gibt es jetzt Unicode, womit es möglich ist, alle Zeichen der Welt in eine Zeichentabelle zu stecken. Natürlich bleibt Unicode kompatibel zu ASCII mit den ersten 128 Zeichen. Allerdings gibt es auch bei Unicode wieder verschiedene Encodings, wovon allerdings UTF-8 und UTF-16 die wichtigsten sind.

Die neuen Unicode-Typen `char16_t` und `char32_t` (C++11)

Da man ja nicht 100 %ig vorhersagen konnte, in welchem Format auf welchem System denn nun genau ein Zeichen in `wchar_t` gespeichert wurde (hängt von der eingestellten locale ab), unterstützt C++ mit dem neuen C++11-Standard jetzt Unicode mit den neuen Zeichentypen `char16_t` und `char32_t` mit einer fixen Breite. Der Vorteil

gegenüber `wchar_t` liegt ganz klar auf der Hand, indem Sie genau wissen, dass mit `char16_t` das Zeichen (mindestens) 16 Bits breit und im UTF-16-Format codiert ist. Ebenso ist es mit `char32_t`, wo das Zeichen (mindestens) 32 Bits breit ist und als UTF-32-Format vorliegt (Tabelle 2.4).

Unicode-Strings

Aus den neuen Zeichentypen `char16_t` und `char32_t` werden auch die neuen Stringtypen `u16string` und `u32string` (siehe Abschnitt 4.2.3, »Unterstützung von Unicode (C++11)«) gebildet.

Codierung	Datentyp	Stringliteral
UTF-8	<code>char</code>	<code>u8"String ist UTF-8 codiert"</code>
UTF-16	<code>char16_t</code>	<code>u"String ist UTF-8 codiert"</code>
UTF-32	<code>char32_t</code>	<code>U"String ist UTF-8 codiert"</code>

Tabelle 2.4 Codierung der Datentypen

Die Verwendung der Unicode-Zeichen lässt sich ebenfalls mit einer Art Steuerzeichen realisieren. Für ein Unicodezeichen in einem String können Sie entweder `\unnnn` (für UTF-16) oder `\Unnnnnnnn` (für UTF-32) verwenden. Für `nnnn` bzw. `nnnnnnnn` müssen Sie die entsprechende 16- bzw. 32-Bit-Hexadezimalzahl für das gewünschte Unicode-Zeichen angeben. Hierzu ein einfacher Codeausschnitt:

```
00 //kapitel02/unicode.cpp
01 char16_t c16val{u'A'}; // (mind.) 16 Bits breites Zeichen
02 char32_t c32val{U'Z'}; // (mind.) 32 Bits breites Zeichen
03 std::cout << sizeof(c16val) << std::endl;
04 std::cout << sizeof(c32val) << std::endl;
05 std::cout << u8"\u00a9 J\u00fcrgen Wolf" << std::endl;
```

In Zeile **01** können Sie sicher sein, dass hier das Zeichen (mindestens) als 16 Bits breites UTF-Zeichen `c16val` zugewiesen wird. Die Ausgabe in Zeile **03** sollte daher mindestens 2 (Bytes) zurückgeben. Selbiges finden Sie in Zeile **02**, nur handelt es sich hier um (mindestens) ein 32 Bits breites UTF-32-Zeichen. Die Ausgabe von Zeile **04** sollte daher mindestens 4 (Bytes) sein. Mit dem `sizeof`-Operator, der in den Zeilen **03** und **04** verwendet wird, wird die Byte-Größe für den Typ zurückgegeben. In Zeile **05** wird zur Demonstration ein UTF-8-codierter String mit dem Inhalt "© Jürgen Wolf" ausgegeben.

Fortsetzung folgt ...

Das Thema Unicode mag Ihnen vielleicht hier noch etwas unvollständig erscheinen, und damit haben Sie auch Recht, aber spätestens wenn Sie später in Abschnitt 4.2, »Strings (Zeichenketten)«, die Strings kennenlernen, werden Sie die neue Unterstützung von Unicode als Ganzes besser verstehen. Dennoch ist es unerlässlich, sich selbst zunächst mit dem Thema Unicode im Allgemeinen auseinanderzusetzen. Leider funktioniert hierbei noch nicht alles auf allen Compilern so, wie man es sich wünscht.

UTF-8-Tabellen

Tabellen mit Unicode-Zeichen finden Sie sehr viele im Internet. Eine hilfreiche Tabelle, welche ich gerne verwende, finden Sie auf <http://www.utf8-zeichentabelle.de>. Wenn Sie aber weit mehr als die landestypischen Unicode-Zeichen suchen, dann empfehle ich Ihnen folgende Tabelle: <http://www.isthisthingon.org/unicode/index.php>.

2.5 Typ `auto` (C++11)

An dieser Stelle will ich Sie schon mal mit dem neuen Schlüsselwort `auto` vertraut machen, welchem Sie noch des Öfteren in diesem Buch begegnen werden.

Altes Schlüsselwort mit neuer Verwendung

`auto` ist nicht wirklich eine neues Schlüsselwort von C++11. In der Vergangenheit (C++03-Standard) konnten Sie das Schlüsselwort noch verwenden, um eine Speicherklasse zu beschreiben. Allerdings wurde es in der Vergangenheit fast nie verwendet, weil Variablen innerhalb von Funktionen ohnehin per Standard `auto` und außerhalb von Funktionen gar nicht erlaubt waren. Wozu also das Schlüsselwort `auto` verwenden, wenn eine Variable bereits `auto` war? So hat auch das Komitee entschieden, dieses Schlüsselwort neu zu definieren. Natürlich bedeutet dies auch, dass `auto` **nicht abwärtskompatibel** zu C++03 ist. Daher müssen Sie, falls Sie in einem älteren Code tatsächlich `auto` verwendet haben oder vorfinden, um die Speicherklasse einer Variablen zu beschreiben, dieses `auto` aus dem Quellcode entfernen.

Ab C++11 wird das Schlüsselwort verwendet, damit bei der Initialisierung automatisch der passende Datentyp vom Compiler ausgewählt wird. Wohlgermerkt ist die Rede hier nicht von einer dynamischen Typisierung, sondern von einer Typinferenz. Bei einer Typinferenz muss der Typ des Objekts nicht mehr explizit festgelegt werden, weil dieser ohnehin implizit schon festgelegt ist. In der Praxis kann eine solche Typinferenz nur dann funktionieren bzw. verwendet werden, wenn Sie die Variable

gleich bei der Erzeugung mit einem Wert initialisieren. Der Compiler identifiziert den Typ von `auto` also anhand des Initialisierers. Hierzu sei folgender Codeausschnitt gegeben:

```
00 //kapitel02/typinferenz.cpp
01 auto aval1 = 1234;    // = int
02 auto aval2 = 1234L;   // = long int
03 auto aval3 = 12.34;   // = double
04 auto aval4 = 12.34F;  // = float
05 auto aval5 = 'A';     // = char
```

Anhand des Initialisierers erkennt der Compiler, welchen Typ er für das Schlüsselwort `auto` einsetzen muss. In Zeile **01** wird `aval1` ein `int`, weil das der bevorzugte Typ vom Compiler ist, wenn ein Ganzzahlliteral ohne weitere Angaben (wie hier mit `1234`) verwendet wird. In Zeile **02** haben wir den Zusatz `L` an das Ende des Literals gehängt, wodurch der Compiler daraus jetzt ein `long int` macht. In Zeile **03** wird der Compiler für `aval3` ein `double` verwenden, weil dies bei Gleitkommalliteralen der bevorzugte Typ des Compilers ist. In Zeile **04** wurde das Suffix `F` hinzugefügt, wodurch der Compiler daraus jetzt ein `float` macht. In Zeile **05** wird `char` als Typ verwendet.

»auto« macht das Leben einfacher

Bei den fundamentalen Datentypen mag Ihnen diese Typinferenz mit `auto` vielleicht ein wenig banal vorkommen. Aber spätestens wenn Sie Dinge wie Templates oder die Standardcontainer mit den Doppelpunkten und spitzen Klammern verwenden, werden Sie das kleine, aber feine Schlüsselwort `auto` nicht mehr missen wollen. Mit `auto` wird das Programm zwar nicht besser, aber der Code ist wesentlich einfacher zu lesen und auch zu schreiben.

Folgendes ist allerdings bei einer Typinferenz mit `auto` nicht erlaubt:

```
01 auto aval1 = 1234;    // = int
02 auto aval2 = aval1;   // Ok! = int
03 aval2 = 123.123;      // Auch Ok
04 aval1 = "string";     // Fehler!!!
```

Die Zuweisung von Zeile **02** geht noch in Ordnung, weil ja der Compiler den Typ in Zeile **01** mit `int` schon festgelegt hat und somit aus Zeile **02** auch ein `int` macht. Auch die Zuweisung in Zeile **03** wird anstandslos (wenn auch nicht schön) übernommen. Nur wird dabei der `double`-Wert `123.123` implizit vom Compiler in einen `int`-Wert konvertiert, so dass `aval2` nur noch den Wert `123` enthält. Diese Konvertierung, welche unabhängig von `auto` ist, wird auch *Narrowing* genannt. Die Zuweisung in Zeile **04** geht aber gar nicht mehr, weil dies hier eine Umwandlung von `const char*` zu `int` wäre und so etwas gar nicht mehr möglich ist.

2.6 Übersicht und Größe der Basisdatentypen

Nachdem Sie jetzt mit den fundamentalen Datentypen in C++ bekannt gemacht wurden, wird es Zeit, ein wenig auf die Details dieser Typen einzugehen – genauer auf deren Größe. Jeder Datentyp ist an eine bestimmte Grenze und Größe gebunden.

Der kleinste Datentyp `char` kann 256 verschiedene Werte speichern und darstellen (unabhängig vom Vorzeichen). Die Anzahl unterschiedlicher Werte ergibt sich aus den acht zusammenhängenden einzelnen Bits ($2^8 = 256$). Jede einzelne dieser acht Bitstellen kann den Wert 1 oder 0 darstellen. Und so ergeben sich 256 verschiedene Bitstellungen. Bei den anderen Datentypen ist dies genauso, nur ist hierbei die Anzahl der Bits häufig länger, wodurch sich viel mehr verschiedene Werte darstellen lassen. Wie viele Bits das sind und wie dieser Wert dann interpretiert wird, geben Sie mit dem Datentyp an.

Ein char mit einem Byte besteht nicht zwangsläufig aus 8 Bits

Hier muss noch ein weitläufiger Irrtum beseitigt werden: Und zwar muss ein `char` mit 1 Byte nicht zwangsläufig aus 8 Bits bestehen. Es gibt bspw. Rechner, auf denen ein `char` mit 32 Bits implementiert ist. Der Standard schreibt nur vor, dass 1 Byte eine zusammenhängende Folge von **mindestens** 8 Bits sein muss.

Wie groß letztendlich die einzelnen fundamentalen Datentypen sind, kann gar nicht zu 100 % genau gesagt werden und hängt davon ab, wie diese implementiert sind. Aber es gibt doch gewisse Spielregeln, an die sich die Compilerhersteller halten müssen. Diese sollen hier jetzt erwähnt werden.

2.6.1 Ganzzahltypen

Der natürlichste Typ bei den Ganzzahlen ist `int`, weil dieser der Größe der Ausführungsumgebung angepasst ist. Bei (älteren) 16-Bit-Systemen waren dies 2 Bytes und bei 32-Bit-Systemen wiederum 4 Bytes. Leider lässt sich dies nicht bei einem 64-Bit-System weiterführen, wo das `int` ja 8 Bytes haben müsste. Auf 64-Bit-Systemen hat man das `int` auf einer Breite von 32 Bits belassen. Stattdessen hat dort jetzt der Datentyp `long int` eine Breite von 64 Bits. Leider trifft dies auch nicht immer zu, so bleibt bspw. auf einem 64-Bit-Windows-Rechner auch ein `long int` bei 32 Bits Breite, für 64 Bits müssen Sie dort auf `long long` zurückgreifen.

Wie breit also die Typen sind, hängt vom zugrunde liegenden Datenmodell ab, das verwendet wird. Sobald Sie anfangen, portablen Code zu schreiben, werden Sie sich wohl auch mit diesem Thema befassen müssen. In Tabelle 2.5 finden Sie einen Überblick über einige gängige Datenmodelle und deren Breite.

Type	LP64	ILP64	LLP64	ILP32	LP32
char	8	8	8	8	8
short	16	16	16	16	16
int	32	64	32	32	16
long	64	64	32	32	32
long long			64		
Zeiger	64	64	64	32	32

Tabelle 2.5 Zum besseren Verständnis: Das L steht für »long«, das P für Pointer (Zeiger), und I steht für »int«. Die Zahlen 32 und 64 sind selbsterklärend.

Nachdem nicht 100 %ig gesagt werden kann, wie breit ein bestimmter Typ implementiert wurde, da das ja vom zugrunde liegenden Datenmodell abhängt, können Sie sich aber auf folgende Reihenfolge verlassen:

`char <= short <= int <= long <= long long`

(<= bedeutet hier: ist kleiner oder gleich)

Datentypen für Zeichen

`char` ist in der Regel immer 1 Byte groß und kann daher 256 verschiedene Zeichen darstellen, was für den ASCII-Code und auch die Umlaute ausreicht. Für breitere Typen steht Ihnen auch der Typ `wchar_t` zur Verfügung, der abhängig von der Implementierung mal 2 und mal 4 Bytes groß ist. Beachten Sie, dass es sich bei diesen Typen im Prinzip trotzdem um Ganzzahltypen handelt. Besser als `wchar_t` sind die neuen in C++11 eingeführten Typen `char16_t` und `char32_t` für die Unicode-Zeichendarstellung von UTF-16 und UTF-32, weil hierbei eine Breite von mindestens 16 bzw. 32 Bits garantiert ist.

Wahrheitswert `bool`

Da `bool` nur zwei Zustände (`true` oder `false`) speichern kann, würde hierfür lediglich 1 Bit ausreichen. Allerdings ist die kleinste adressierbare Einheit 1 Byte, weshalb `bool` auch mit dieser Größe implementiert sein könnte. Auf manchen Systemen hat ein `bool` aber auch dieselbe Größe wie die Prozessorarchitektur (32 oder 64 Bits), was eine bessere Zugriffsgeschwindigkeit ermöglichen kann.

2.6.2 Gleitkommazahlen

Auch bei den Gleitkommazahlen lässt sich nicht mit 100 %iger Sicherheit vorhersagen, wie breit diese Typen auf den unterschiedlichen Systemen sind. `float` ist zwar häufig mit 4, `double` mit 8 und `long double` mit 10 bzw. 16 Bytes vorzufinden, aber eine Garantie gibt es hierfür auch nicht. Aber wie auch bei den Ganzzahlen können Sie sich auf folgende Reihenfolge verlassen:

```
float <= double <= long double
```

(<= bedeutet hier: ist kleiner oder gleich)

2.6.3 Byte-Größe ermitteln – sizeof-Operator

Die Größe der fundamentalen Datentypen auf Ihrem System können Sie mit dem `sizeof`-Operator ermitteln. Hierbei müssen Sie lediglich den gewünschten Datentyp zwischen den Klammern des `sizeof`-Operators (bspw. `sizeof(Datentyp)`) stellen, und es wird die Byte-Größe für den Datentyp zurückgegeben. Hier einige Beispiele dazu:

```
00 //kapitel02/sizeof.cpp
01 std::cout << sizeof(char) << std::endl; // = immer 1 Byte
02 std::cout << sizeof(wchar_t) << std::endl;
03 std::cout << sizeof(int) << std::endl;
04 std::cout << sizeof(double) << std::endl;
05 std::cout << sizeof(long double) << std::endl;
```

Ausgeführt wurden diese Beispiele auf unterschiedlichen Systemen und besonders bei `wchar_t` in Zeile 02 und `long double` in Zeile 05 gab es häufig unterschiedliche Größen. `wchar_t` war auf dem einen System 2 und auf anderen wiederum 4 Bytes breit. `long double` hatte hingegen 8, 10 und 16 Bytes auf Testsystemen.

2.6.4 Sicherheit beim Kompilieren mit `static_assert` (C++11)

Erfreulicherweise ist dieses neue C++11-Feature bei fast alle gängigen neueren Compilern implementiert. Mit `static_assert()` überprüfen Sie einen konstanten Ausdruck zwischen den Klammern zur Übersetzungszeit. Gibt die Auswertung dieses Ausdrucks nicht `true` zurück, bricht der Compiler die Übersetzung mit der Fehlermeldung ab, welche Sie ebenfalls angeben können. Daher passt dieses `static_assert()` recht gut in dieses Kapitel hier rein. Ein einfaches Beispiel hierzu:

```
00 //kapitel02/static_assert.cpp
01 static_assert( sizeof(long double) == 16,
                "Need 16 byte long double" );
```

Hier fordern wir den Compiler auf, den Ausdruck `sizeof(long double)==16` zu überprüfen. Unsere Anwendung erfordert 16 Bytes für ein `long double` auf dem System, auf dem der Quellcode übersetzt wird. Ist der Ausdruck `true`, wird der Quellcode weiter übersetzt. Gibt der Ausdruck `false` zurück, bricht der Compiler die Übersetzung ab und gibt die dahinter geschriebene Fehlermeldung (hier »Need 16 byte long double«) aus (Abbildung 2.1).



Abbildung 2.1 Beim Microsoft Visual Studio ist auf diesem Testsystem ein »long double« nicht 16 Bytes breit, und daher bricht der Compiler die Übersetzung dank unseres `static_assert()` ab.

Machen Sie regen Gebrauch von `static_assert()`

Die Verwendung von `static_assert()` kann nur empfohlen werden. Sie kostet auch überhaupt keine Laufzeit der Anwendung, weil diese Sicherheitschecks nur vom Compiler benutzt werden. Lediglich die Übersetzungszeit nimmt logischerweise zu, aber ich denke, damit kann man leben, weil man hiermit auf so manche Sicherheitsüberprüfung während der Laufzeit des Programms verzichten kann. Und dadurch muss man ja eigentlich auch weniger Code schreiben, wodurch wiederum die Laufzeit des Programms verbessert wird. Die Überprüfungen mit `static_assert()` sind nicht nur auf fundamentale Datentypen beschränkt, sondern machen u.a. auch später Sinn, etwa um auf falsche Template-Parameter zu testen. Auch können Sie diese Prüfungen an jeder Stelle (in einem Namensraum, global, lokal) im Code einbringen.

2.6.5 `<limits>` und `std::numeric_limits`

Wenn Sie auf der Suche nach mehr Informationen zu den fundamentalen Datentypen sind, dann werden Sie mit der Headerdatei `<limits>` und der darin enthaltenen Spezialisierung des Templates `numeric_limits` fündig. Die Spezialisierungen befinden sich im Namensraum `std`. Die Verwendung ist im Grunde denkbar einfach, auch wenn die Templates hier noch nicht behandelt wurden:

```
numeric_limits<DATENTYP>::member_constant_or_function;
```

Zwischen den spitzen Klammern geben Sie für DATENTYP den fundamentalen Datentyp an, über den Sie Informationen haben möchten. Hinter den beiden Doppelpunkten geben Sie die Member-Funktion oder Member-Konstante an, wovon in der Headerdatei `<limits>` eine Menge enthalten sind. So lautet bspw. die Member-Funktion für den maximalen Wert eines Datentyps `max()` oder eine Member-Konstante, um herauszufinden, ob der Typ ein Vorzeichen hat, `is_signed`. Hierzu ein einfaches Beispiel, wie Sie Member-Funktion und Member-Konstanten anwenden können:

```
00 //kapitel02/limits.cpp
01 #include <limits> // benötigte Headerdatei
...
02 std::cout << std::numeric_limits<int>::min()
    << std::endl;
03 std::cout << std::numeric_limits<int>::max()
    << std::endl;
04 std::cout << std::numeric_limits<int>::digits
    << std::endl;
05 std::cout << std::numeric_limits<int>::is_signed
    << std::endl;

06 static_assert(
    (std::numeric_limits<unsigned char>::digits == 8),
    "!!! unsigned char hat hier keine 8 Bits !!!"
    );

07 std::cout << std::numeric_limits<long>::min()
    << std::endl;
08 std::cout << std::numeric_limits<long>::max()
    << std::endl;
09 std::cout << std::numeric_limits<long>::digits
    << std::endl;
10 std::cout << std::numeric_limits<long>::is_signed
    << std::endl;
```

Der Header in Zeile **01** ist nötig, um die Spezialisierungen von `numeric_limits` verwenden zu können. In den Zeilen **02** bis **03** ermitteln Sie den kleinstmöglichen (`min()`) und größtmöglichen (`max()`) Wert, der in einem `int` gespeichert werden kann.

Die Anzahl der Bits eines Datentyps **ohne** das Vorzeichenbit (!) können Sie mit der Member-Konstante `digits` ermitteln. Das ist auch der Grund, warum in Zeile **04** bspw. 31 (Bits) statt der vielleicht eher erwarteten 32 zurückgegeben wird. Testen Sie es einfach aus, indem Sie ein `unsigned int` stattdessen verwenden, wobei dann auch

32 (Bits) zurückgegeben wird (ich gehe jetzt davon aus, dass Sie vor keinem ILP64-System sitzen; siehe Abschnitt 2.6.1, »Ganzzahltypen«).

Ob ein Typ mit einem Vorzeichen implementiert wurde, können Sie mit `is_signed` (siehe Zeile 05) ermitteln. Zurückgegeben wird entweder `true` (mit Vorzeichen) oder `false` ohne Vorzeichen. In dem Fall wird 1 für `true` und 0 für `false` ausgegeben.

Selbstverständlich können Sie die Spezialisierungen auch mit `static_assert()` verwenden (siehe Abschnitt 2.6.4, »Sicherheit beim Kompilieren mit `static_assert` (C++11)«). In Zeile 06 machen wir dies, indem unser Programm voraussetzt, dass `unsigned char` auf dem System 8 Bits breit ist, um übersetzt werden zu können. In den meisten Fällen wird das Beispiel wohl ohne Anstand übersetzt. Wenn nicht, dann haben Sie eine Maschine vor sich, auf der `char` größer als 8 Bits ist.

In den Zeilen 07 bis 10 wird nochmals mit dem Typ `long` alles wiederholt, was eben zuvor mit `int` bei den Zeilen 02 bis 05 gemacht wurde.

Die Member in `<limits>`

In der Headerdatei `<limits>` sind noch viele weitere Mitglieder vorhanden. Hierfür lohnt sich ein Blick in eine Referenz Ihrer Wahl (bspw. <http://en.cppreference.com>).

2.6.6 `<climits>` und `<cfloat>`

Die beiden alten Headerdateien `<climits>` (für Ganzzahltypen) und `<cfloat>` (für Gleitkommatypen) sind in der C++-Praxis nicht mehr nötig, seit die Spezialisierungen in Form von Templates mit `<limits>` eingeführt wurden. Beide Headerdateien sind Relikte aus der C-Programmierung und sollten, wenn möglich, in künftigen C++-Projekten nicht mehr verwendet werden.

Kleine Randnotiz: Anzahl der Bits eines `char`

Zwar wird empfohlen, nicht auf Konstanten von `<climits>` oder `<cfloat>` zurückzugreifen, aber früher oder später kommt ein Anwender einmal damit in Berührung, die Anzahl der Bits für ein `char` ermitteln zu müssen. Ein beliebter Fehler hierbei ist es, `std::numeric_limits<char>::digits` zu verwenden. Da nicht 100 %ig gesagt werden kann, ob `char` als `signed` oder `unsigned` implementiert ist, könnte der Rückgabewert 7 (wenn `signed`) oder eben 8 (wenn `unsigned`) zurückliefern. Es wird leider immer wieder vergessen, dass `digits` die Anzahl von Bits **ohne** Vorzeichenbit zurückgibt. Entweder greifen Sie hierbei auf die alte Konstante `CHAR_BIT` aus `<climits>` zurück, oder Sie verwenden hierfür die modernere (bessere) und gleichwertige Alternative mit `std::numeric_limits<unsigned char>::digits`.

2.6.7 Übersicht zu den fundamentalen Datentypen

An dieser Stelle finden Sie noch eine kurze Übersicht zu den fundamentalen Datentypen. Die Werte für die Größe oder den Wertebereich sind implementierungsabhängig und entsprechen in den folgenden Tabellen lediglich Werten, wie sie recht häufig anzutreffen sind. Wie groß ein Typ auf Ihrem System tatsächlich ist oder welchen Wertebereich er hat, können Sie mit `sizeof(T)` (für die Speichergröße in Bytes) oder `std::numeric_limits<T>::min()` bzw. `std::numeric_limits<T>::max()` (für den minimalen bzw. maximalen Wertebereich) ermitteln. Für T geben Sie den entsprechenden Datentyp an.

Integrale Typen (Ganzzahltypen)

Der grundlegende integrale Datentyp ist `int` mit allen dazugehörenden Verwandten. Auch der Wahrheitswert `bool` und der Zeichentyp `char` gehören zur Gruppe der integralen Typen. In Tabelle 2.6 finden Sie eine Übersicht zu diesen Typen.

Datentyp	Speicher	Gängiger Wertebereich
<code>bool</code>	1 Byte	<code>true</code> (1) oder <code>false</code> (0)
<code>char</code>	1 Byte	−128 bis +127 bzw. 0 bis 255
<code>signed char</code>	1 Byte	−128 bis +127
<code>unsigned char</code>	1 Byte	0 bis 255
<code>wchar_t</code>	2 oder 4 Bytes	implementierungsabhängig
<code>short</code>	2 Bytes	−32.768 bis +32.767
<code>unsigned short</code>	2 Bytes	0 bis 65.535
<code>int</code>	4 Bytes	−2.147.483.648 bis +2.147.483.647
<code>unsigned int</code>	4 Bytes	0 bis 4.294.967.295
<code>long</code>	4 oder 8 Bytes	−9.223.372.036.854.775.808 bis +9.223.372.036.854.775.807
<code>unsigned long</code>	4 oder 8 Bytes	wie <code>unsigned int</code> oder 0 bis 18.446.744.073.709.551.615

Tabelle 2.6 Ganzzahltypen

Neue Typen mit C++11

Ergänzend zu Tabelle 2.6 mit den Ganzzahltypen, wurden mit dem C++11-Standard noch folgende Datentypen offiziell hinzugefügt (Tabelle 2.7).

Datentyp	Speicher	Gängiger Wertebereich
char16_t	2 Bytes	implementierungsabhängig
char32_t	4 Bytes	implementierungsabhängig
long long	8 Bytes	−9.223.372.036.854.775.808 bis +9.223.372.036.854.775.807
unsigned long long	8 Bytes	0 bis 18.446.744.073.709.551.615

Tabelle 2.7 Neue C++11-Datentypen

Gleitkommatypen

Zu guter Letzt fehlt Ihnen nur noch der Überblick über die Gleitkommatypen von C++ (Tabelle 2.8).

Datentyp	Speicher	Gängiger Wertebereich
float	4 Bytes	1.2E-38 bis 3.4E+38
double	8 Bytes	2.3E-308 bis 1.7E+308
long double	10 (oder 16) Bytes	3.4E-4.932 bis 1.1E+4.932

Tabelle 2.8 Gleitkommatypen

2.7 Rechnen mit C++

Nachdem Sie nun die fundamentalen Datentypen kennen, ist es an der Zeit, Ihnen zu zeigen, wie Sie damit einfache arithmetische Operationen ausführen können. An dieser Stelle werden Sie auch bemerken, dass es hier doch einige Tücken gibt, die beachtet werden müssen. Wir beschränken uns bei diesem Buch auf grundlegende und einfache Rechenoperationen.

Unterschiedliche Typen

Falls Sie bei arithmetischen Berechnungen unbedingt Datentypen mischen müssen, sollten Sie 100 % darüber Bescheid wissen und nichts dem Zufall überlassen. Hierbei besteht die Möglichkeit, alles dem Compiler zu überlassen oder die Fäden selbst in die Hand zu nehmen.

Seit C++11 besteht endlich auch die Möglichkeit, die automatische Konvertierung einzuschränken. Auf das Thema wird in Abschnitt 2.9, »Konvertieren von Typen«, eingegangen.

2.7.1 Arithmetische Operatoren

An dieser Stelle kann ich davon ausgehen, dass Sie mit den grundlegenden Bedeutungen arithmetischer Berechnungen (aus Grundschulzeiten) vertraut sind. In Tabelle 2.9 finden Sie die Darstellung der verwendeten arithmetischen Operatoren in C++.

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo

Tabelle 2.9 Arithmetische Operatoren in C++

Alle Operatoren können Sie auf integralen Ganzzahltypen- und Gleitkommatypen anwenden, mit Ausnahme des %-Operators (=Modulo). Dieser darf nur für Ganzzahlen verwendet werden.

%-Operator (Modulo)

Mit dem Modulo-Operator wird eine Division von zwei Ganzzahlen durchgeführt. Als Ergebnis erhalten Sie dann den Rest der Division zurück. So würde eine Berechnung von `10 % 3` das Ergebnis 1 ergeben, weil der Rest von `10/3` gleich 1 ist.

Bei den arithmetischen Operatoren wird auch von *binären Operatoren* gesprochen, weil diese zwei Operanden benötigen (bi = zwei). Operanden können dabei Variablen oder Werte sein. Da eine arithmetische Berechnung, wie bspw. `val1+val2`, häufig keinen Sinn macht (außer bei Überprüfungen oder als Argument bei einem Funktionsaufruf bzw. Rückgabewert), werden Sie sicherlich den Wert dieser Berechnung irgendwo speichern wollen. Wenig überraschend dürfte es sein, dass Sie hierfür den Zuweisungsoperator (=) verwenden können.

Folgender Codeausschnitt soll die arithmetischen Operatoren in der Praxis demonstrieren:

```
00 //kapitel02/calc.cpp
01 int result={0};
02 int val1={100};
```

```

03 int val2={30};
04 int rest{val1 % val2}; // C++11 only !!!

05 val1 + val2; // Nutzloser Ausdruck
06 result = val1 * val2;
07 std::cout << result << std::endl; //3000
08 std::cout << rest << std::endl; //10
09 std::cout << val1 + val2 << std::endl; //130

10 result = val1 + val2 * 2;
11 std::cout << result << std::endl; //160
12 result = (val1 + val2) * 2;
13 std::cout << result << std::endl; //260
14 std::cout << 102.2 / 33.2 << std::endl; //3.07831

15 std::cout << "Eine einfache Addition!" << std::endl;
16 std::cout << "1. Summand: ";
17 std::cin >> val1;
18 std::cout << "2. Summand : ";
19 std::cin >> val2;
20 std::cout << "Summe = " << val1 + val2 << std::endl;

```

In Zeile **04** wurde eine arithmetische Berechnung direkt der Variablen `rest` zugewiesen. In diesem Fall enthält `rest` den Rest der Division von `val1/val2`. Diese Zuweisung zwischen geschweiften Klammern funktioniert allerdings so nur mit dem neuen C++11-Standard. Bei einem Nicht-C++11-Compiler verwenden Sie stattdessen bitte `int rest=val1%val2`; Das Ergebnis in `rest` wird in Zeile **08** ausgegeben.

Die Addition in Zeile **05** ist zwar kein Fehler, aber in dieser Zeile recht nutzlos, was der Compiler gewöhnlich mit einer Warnmeldung monieren sollte. In Zeile **06** wird der Wert von `val1` mit `val2` multipliziert und `result` zugewiesen und das Ergebnis in Zeile **07** ausgegeben. In Zeile **09** können Sie sehen, wie Sie eine Berechnung als Ausdruck direkt über den `<<`-Operator nach `cout` schicken und ausgeben können, ohne dass der Wert zwischengespeichert wird.

Natürlich können Sie Zahlen und Variablen bei den arithmetischen Berechnungen beliebig mischen, wie dies in den Zeilen **10** und **12** demonstriert wird. Dass hierbei bei gleichen Werten in den Zeilen **11** und **13** unterschiedliche Ergebnisse ausgegeben werden, liegt daran, dass sich C++ auch an die Punkt-vor-Strich-Regelung hält.

Punkt- vor Strichrechnung

Bei der Regel der Punkt- vor Strichrechnung werden Multiplikationen und Divisionen vor Additionen und Subtraktionen durchgeführt. Daher ergibt eine Berechnung von `10 + 10 * 2` den Wert 30, weil hier zuerst `10 * 2` durchgeführt und dann die 10 hinzuaddiert wird. Wollen Sie zuerst die Strichrechnung durchführen, müssen Sie Klammernungen verwenden (siehe Zeile 12), weil geklammerte Operatoren einen höheren Rang bei der Operatorrangfolge haben. Damit würde `(10 + 10) * 2` den Wert 40 ergeben, weil nach dieser Schreibweise zuerst die Berechnung in den Klammern `10 + 10` durchgeführt und dieser Wert erst anschließend mit 2 multipliziert wird.

Sinngemäß können Sie auch nur konstante Zahlen für eine Berechnung verwenden, wie Sie dies in Zeile 14 sehen. Da nicht anders angegeben, werden diese beiden Gleitkommazahlen als `double`-Werte behandelt. In den Zeilen 15 bis 20 wird noch eine interaktive Addition durchgeführt.

Erweiterter arithmetischer Zuweisungsoperator

Für alle eben gezeigten arithmetischen Rechenoperatoren `+`, `-`, `*`, `/` und `%` gibt es noch eine erweiterte Zuweisungsform mit `+=`, `-=`, `*=`, `/=` und `%=`. In Tabelle 2.10 finden Sie die Verwendung dieser Form.

Arithmetische Berechnung	Kürzere Schreibweise
<code>val1 = val1 * 100;</code>	<code>val1 *= 100;</code>
<code>val1 = val1 + val2;</code>	<code>val1 += val2;</code>
<code>val1 = val1 - val2;</code>	<code>val1 -= val2;</code>
<code>val1 = val1 / 2;</code>	<code>val1 /= 2;</code>
<code>val1 = val1 % 3;</code>	<code>val1 %= 3;</code>

Tabelle 2.10 Arithmetische Berechnungen und gleichwertige kürzere Gegenstücke

2.7.2 Unäre Gegenstücke

Neben den binären Operatoren haben die Operatoren `+`, `-` und `*` noch jeweils ein unäres Gegenstück. Im Gegensatz zum binären Operator haben die unären Operatoren nur eine einseitige Beziehung zu einem Operanden (Tabelle 2.11).

Operator	Binär	Unär
+	Addition (10 + 10)	Liefert das Positive einer Zahl zurück (+10).
-	Subtraktion (10 – 5)	Liefert das Negative einer Zahl zurück (–10)
*	Multiplikation (10 * 10)	Dereferenzierungsoperator (siehe Abschnitt 4.3, »Zeiger«)

Tabelle 2.11 Unäre Doppelgänger zum binären Gegenstück

Da die unären Operatoren einen höheren Rang als die binären haben, gibt es auch keine Probleme, wenn Sie diese bei arithmetischen Berechnungen verwenden. Hierzu ein kleines Beispiel:

```
00 //kapitel02/unaer.cpp
01 int negval={-123};
02 int posval={+123};

03 std::cout << negval + posval << std::endl; //0
04 std::cout << -50 + -100 << std::endl;      //=-150
05 std::cout << -150 + +200 << std::endl;      //50
```

In Zeile **01** erhält `negval` dank des unären Minus einen negativen Wert. Auf das unäre Plus in Zeile **02** hingegen könnten Sie verzichten, weil der Wert ohne weitere Angaben automatisch positiv ist. Die Addition von `negval` und `posval` ergibt gleich den Wert 0, wie Zeile **03** ausgibt. Die Berechnungen in den Zeilen **04** und **05** sollen demonstrieren, dass die unären Operatoren eine höhere Priorität in der Operatorrangfolge haben. Dabei wird zuerst der Wert mit dem Vorzeichen ausgewertet und dann die arithmetische Berechnung ausgeführt. Auch hier hätte man in Zeile **05** auf das Plus vor dem Wert 200 verzichten können.

Weitere unäre Operatoren

Die hier gezeigten unären Operatoren sind nicht die einzigen in C++. Im Verlauf des Buches werden Sie noch auf weitere unäre Operatoren stoßen.

2.7.3 Wenn Wertebereiche überschritten werden

Wenn Sie Datentypen für arithmetische Berechnungen verwenden, müssen Sie beachten, dass es hierbei auch einen Wertüberlauf geben kann. Im Beispiel vom Datentyp `int` würde mit `std::numeric_limits<int>::max()+1` ein Überlauf stattfinden. Gerade wenn Sie eine Anwendung erstellen, in der ein Anwender unterschiedli-

che Werte eingeben muss, sollten Sie sich niemals darauf verlassen, dass der Anwender schon das Richtige eingeben wird. Der folgende Codeausschnitt soll Ihnen einen solchen Überlauf demonstrieren:

```
00 //kapitel02/ueberlauf.cpp
01 int val1 = {2147483647+1};
02 int val2 = {2147483647};
03 int val3 = {0};

04 std::cout << "Bitte eine Ganzzahl: ";
05 std::cin >> val3;
06 val2 += val3;
07 std::cout << val1 << std::endl; // -2147483648
08 std::cout << val2 << std::endl;
```

In Zeile 01 findet ein Integer-Überlauf statt, wie die Ausgabe in Zeile 07 auch bestätigt. Hierbei könnte sich der Compiler mit einer Warnmeldung beschweren. Anders sieht es allerdings schon bei der Berechnung in Zeile 06 aus. Wenn der Anwender in Zeile 05 einen Wert größer als 0 eingibt, findet ebenfalls ein Überlauf statt, welcher aber logischerweise nicht mehr zur Übersetzungszeit vom Compiler beanstandet werden kann.

Betrachten Sie am einfachsten die interne Bitdarstellung bei der Berechnung, und Sie werden den Überlauf verstehen:

0111 1111 1111 1111 1111 1111 1111 1111	2147483647
+ 0000 0000 0000 0000 0000 0000 0001	+ 1
-----	-----
1000 0000 0000 0000 0000 0000 0000	-2147483648

Genau genommen handelt es sich bei der Bitdarstellung des Ergebnisses ja tatsächlich um den Wert 2.147.483.648. Aber für die Darstellung von `signed int` stehen Ihnen gewöhnlich 31 Bits (`std::numeric_limits<int>::digits`) zur Verfügung. Das höchste Bit (hier das 32.) ist für das Vorzeichen verantwortlich, weshalb hier ein Vorzeichenwechsel stattfindet und somit der Wert eben -2.147.483.648 lautet. Hätten Sie hier `unsigned int` verwendet, würde der Wert 2.147.483.648 lauten. Aber auch mit `unsigned int` würde bei `std::numeric_limits<int>::max()+1` ein Überlauf stattfinden, wie folgende Bitdarstellung zeigen soll:

1111 1111 1111 1111 1111 1111 1111 1111	4294967295
0000 0000 0000 0000 0000 0000 0001	+ 1
-----	-----
1 0000 0000 0000 0000 0000 0000 0000	0

Da Ihnen für `unsigned int` hier 32 Bits zur Darstellung des Wertes zur Verfügung stehen, wird aus `4.294.967.295+1` der Wert 0 und nicht 4.294.967.296. Hätte `unsigned int` 33 Bits, würde der Wert sogar richtig dargestellt werden. Aber in diesem Fall hat `unsigned int` nun mal nur 32 Bits, und somit wird das 33. Bit (hier durchgestrichen) einfach abgeschnitten.

Bereichsüberschreitung ist ein Fehler!

Auch wenn es hier den Anschein hat, dass ja bei einem Wertüber- bzw. Unterlauf alles wieder von Neuem beginnt (ähnlich wie bei einer Uhr) und auch das Programm weiterhin seinen Dienst tut, sollten Sie trotzdem beachten, dass ein solcher Über- bzw. Unterlauf eigentlich ein Fehler ist. Es ist nicht garantiert, dass alles wieder von vorn beginnt.

Ansätze zum Prüfen von Über- bzw. Unterläufen

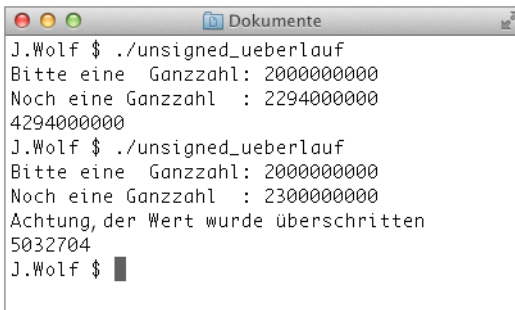
An dieser Stelle angekommen, muss leider erwähnt werden, dass der C++-Standard selbst keine Lösung anbietet, einen Über- bzw. Unterlauf zu überprüfen. Es ist Aufgabe des Programmierers, sicherzustellen, dass keine Wertebereichsüberschreitung stattfindet. Einen möglichen Ansatz, die Überschreitung eines `unsigned`-Typs zu überprüfen, soll das folgende Listing demonstrieren:

```
00 //kapitel02/unsigned_ueberlauf.cpp
01 unsigned int val1 = {0};
02 unsigned int val2 = {0};
03 unsigned int result = {0};

04 std::cout << "Bitte eine Ganzzahl: ";
05 std::cin >> val1;
06 std::cout << "Noch eine Ganzzahl  : ";
07 std::cin >> val2;
08 result = val1 + val2;
09 if( result < val1 ) {
10     std::cout << "Achtung, der Wert wurde überschritten"
11         << std::endl;
12 }
12 std::cout << result << std::endl;
```

Bei diesem Beispiel wird eine einfache Addition durchgeführt. Die Überprüfung eines Überlaufs führen wir in Zeile 09 durch. Das Prinzip ist recht einfach: Wenn das Ergebnis der Addition in `result` kleiner als einer der eingegebenen Summanden ist, dann kann etwas nicht stimmen, und es muss wohl ein Überlauf stattgefunden haben. Zur Überprüfung wird hier vorausgreifend eine `if`-Bedingung (siehe

Abschnitt 3.1, »Bedingte Anweisung«) verwendet. Das Programm bei der Ausführung mit und ohne Überlauf zeigt Ihnen Abbildung 2.2.



```
J.Wolf $ ./unsigned_ueberlauf
Bitte eine Ganzzahl: 2000000000
Noch eine Ganzzahl : 2294000000
4294000000
J.Wolf $ ./unsigned_ueberlauf
Bitte eine Ganzzahl: 2000000000
Noch eine Ganzzahl : 2300000000
Achtung, der Wert wurde überschritten
5032704
J.Wolf $
```

Abbildung 2.2 Das Programm bei der Ausführung: einmal ohne und einmal mit absichtlichem Überlauf

Beispiel nicht ganz »wasserdicht«

Es sollte klar sein, dass eine solche manuelle Überprüfung nur so lange funktioniert, wie zwei gleiche Typen mit ebenfalls erlaubten Werten verwendet werden. Sobald Sie einen konstanten bzw. größeren Wert verwenden, als der Zieltyp aufnehmen kann, ist auch hier der weitere Verlauf wieder unvorhersehbar.

Wesentlich einfacher geht da schon die Überprüfung von vorzeichenbehafteten Typen (*signed*) von Hand, weil Sie hier lediglich anhand des höchsten Bits überprüfen müssen, ob das Vorzeichen gewechselt wurde. Zwar muss hierbei ein etwas komplexerer Code verwendet werden, aber einen möglichen Lösungsansatz will ich Ihnen auch hierzu nicht vorenthalten:

```
00 //kapitel02/signed_ueberlauf.cpp
01 #include <limits>
...
02 int val1 = {0};
03 int val2 = {0};
04 int result = {0};
05 std::cout << "Bitte eine Ganzzahl: ";
06 std::cin >> val1;
07 std::cout << "Noch eine Ganzzahl : ";
08 std::cin >> val2;
09 result = val1 + val2;

10 if( (val1&(1<<std::numeric_limits<int>::digits)) !=
      (result&(1<<std::numeric_limits<int>::digits)) ) {
```

```

11     std::cout << "Achtung! Vorzeichen wurde gewechselt"
        << std::endl;
12 }
13 std::cout << result << std::endl;

```

Die Überprüfung auf einen Vorzeichenwechsel führen wir in Zeile 10 durch. Mit den beiden kryptischen Zeilen überprüfen wir lediglich das höchste Bit von `val1` und `result` mit jeweils einem booleschen Ausdruck. Ist hierbei das höchste Bit von `val1` ungleich (!=) dem höchsten Bit von `result`, wurde wohl (logischerweise) das Vorzeichen gewechselt.

Weitere Informationen

Zugegeben, hier wurden keine ultimativen Lösungen zu diesem Thema geliefert. Das war aber auch gar nicht meine Absicht. Es war mir nur wichtig, Sie als Leser auf das Thema hinzuweisen, so dass Sie sich über Dinge wie *Integer Overflow* von Anfang an im Klaren sind und beim Codedesign darauf achten. Mir ist auch bewusst, dass dieses Thema hier vielleicht zunächst zu viel des Guten war. Für Einsteiger war dieser Abschnitt wohl definitiv zu komplex. Trotzdem wäre ich zufrieden, wenn hier zumindest hängengeblieben ist, dass man nicht einfach arithmetische Berechnungen durchführen und sich darauf verlassen kann, dass der Anwender schon das Richtige eingeben wird.

Wenn Sie auf der Suche nach einer fertigen Bibliothek sind, dann kann ich Ihnen wärmstens die Bibliothek *SafeInt* (<http://safeint.codeplex.com>) ans Herz legen. Die Bibliothek ist dermaßen beliebt, dass sie bei Microsoft Visual C++ 2012 von Haus aus mit der Headerdatei `<safeint.h>` dabei ist.

2.7.4 Rundungsfehler bei Gleitkommazahlen

Nach einem etwas ungemütlicheren Thema muss ich Ihnen leider noch ein weiteres recht ähnliches zumuten. Und zwar betrifft es die Gleitkommazahlen, die nicht immer exakt dargestellt werden können – sprich, es kann zu Rundungsfehlern kommen.

Interna von Gleitkommazahlen

Die Interna von Gleitkommazahlen sind eher Sache der theoretischen Informatik, worüber Sie sich bei Bedarf bitte selbst informieren. In diesem Abschnitt werden nur kurz grundlegende und programmierspezifische Dinge zu den Gleitkommazahlen behandelt. Das Thema Gleitkommazahlen ist sehr umfangreich und kann hier nicht in einem Lehrbuch zu C++ abgehandelt werden.

Gleitkommazahlen können binär nicht immer exakt dargestellt werden. Ursache dafür ist die Darstellungsweise von Gleitkommazahlen durch den Prozessor. Daher können Genauigkeitsverluste und resultierend bei arithmetischen Berechnungen fehlerhafte Ergebnisse auftreten. Damit Sie in C++ nicht über diese Rundungsfehler stolpern, sollten Sie folgende Ratschläge beherzigen:

- ▶ Bei Überprüfungen sollten Sie Gleitkommazahlen niemals auf Gleichheit (`==` bzw. `!=`) testen. Selbst eine Gleitkommazahl wie 1.333 kann Rundungsfehler enthalten. Verwenden Sie dafür immer die Operatoren für größer gleich (`>=`) oder kleiner gleich (`<=`).
- ▶ Entwickeln Sie Programme, in denen Geldbeträge verwaltet werden, sollten Sie ganz auf Gleitkommazahlen verzichten. Verwenden Sie stattdessen Ganzzahlen. Einen Geldbetrag wie 1,99 Euro sollten Sie dann mit 199 Eurocents berechnen.
- ▶ Müssen Sie trotzdem Gleitkommazahlen bei Ihren Projekten verwenden, müssen Sie sich unbedingt umfassend damit auseinandersetzen und gegebenenfalls externe Alternativen (Bibliotheken) in Erwägung ziehen.
- ▶ Entgegen der Intuition können auch Addition und Subtraktion gerne mal Rundungsfehler verursachen. Daher sollten Sie `+` und `-` genauso kritisch betrachten wie `*` und `/`.

Probleme bei der Genauigkeit von Gleitkommazahlen

Neben den Rundungsfehlern müssen Sie auch die Genauigkeit von Gleitkommazahlen im Auge behalten, welchen auch nur endlich ist. Ohne mich in Details zu verstricken: Ein Gleitkommatyp muss das Vorzeichen, den Exponenten und die Mantisse speichern. Dadurch, dass die Mantisse auch begrenzt ist, entstehen irgendwann Ungenauigkeiten. In der Regel kommt das genormte Gleitkommasystem IEEE 754 in C++ zum Einsatz, wo die Nachkommastelle in der Mantisse gespeichert wird. Bei einer einfachen Genauigkeit wie dem Typen `float` stehen hierfür 23 Bits zur Verfügung. Bei der doppelten Genauigkeit wie bei einem `double` hat der Nachkommateil gewöhnlich 52 Bits. Nicht exakt definiert ist die Breite für Typen (`long double`) mit erweiterter Genauigkeit. Das folgende Beispiel soll Ihnen diesen Sachverhalt zur Genauigkeit von Gleitkommazahlen näher demonstrieren:

```
00 // kapitel02/genauigkeit.cpp
01 float fval = {1234.12 / 1.23};
02 double dval = {1234.12 / 1.23};

03 std::cout.precision(10);
04 std::cout << fval << std::endl; // 1003.349609
05 std::cout << dval << std::endl; // 1003.349593
```

In den Zeilen **01** und **02** wird jeweils dieselbe Berechnung durchgeführt und als Ergebnis einmal in ein `float` und einmal in `double` gespeichert. In Zeile **03** stellen wir für die Genauigkeit der Ausgabe von `cout` ein, wie viele Stellen wir sehen wollen. Diese Einstellung hat keinen Einfluss auf den Inhalt der Werte von `float` bzw. `double`. Die Ausgabe in den Zeilen **04** und **05** zeigen jetzt das Problem mit der Genauigkeit von Gleitkommazahlen. `float` ist nach sieben Stellen am Ende und könnte quasi nicht mehr zwischen 1003.349609 und 1003.349593 unterscheiden. Bei `double` ist der Spielraum mit 15 Stellen noch etwas größer.

Genauigkeit reicht mir nicht aus

Wenn Ihnen die Genauigkeit von `double` bzw. `long double` auch nicht mehr ausreicht, bleibt Ihnen nichts anderes übrig, als nach Alternativen zu suchen. Hier würde sich bspw. *Arbitrary Length Integer* für Fließkommazahlen (http://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic) anbieten. Boost bietet hierfür bspw. eine Bibliothek mit an, welche *Arbitrary Length Integer* verwendet. Genauso würde sich die *GNU MP Bignum Library* für Integer anbieten (<http://gmplib.org>) oder *GNU MPFR Library* (<http://www.mpf.fr.org>) für Gleitkommazahlen.

Wozu ist float überhaupt gut?

Da ja `double` der bevorzugte Typ vom Compiler ist und auch wesentlich genauer ist als `float`, stellt sich die Frage, wozu und wann man denn überhaupt `float` verwenden sollte. Die wenigen Beweggründe, die mir hierzu einfallen, sind bspw. eine Schnittstelle (was ja auch eine Hardware sein kann), welche `float` verlangt, und auch der geringere Speicherverbrauch (was allerdings erst bei extrem großen Arrays ins Gewicht fallen dürfte).

2.7.5 Komplexe Zahlen – `<complex>`

Wenn Sie auf der Suche nach komplexen Zahlen mit Real- und Imaginärteil sind, werden Sie mit der Standard-Template-Klasse `complex` im Namensraum `std` fündig. Komplexe Zahlen können Sie mit allen drei Gleitkommatypen `float`, `double` und `long double` verwenden. Hierfür müssen Sie den gewünschten Typ lediglich zwischen spitze Klammern (`<Typ>`) stellen, da es sich um ein Template handelt.

```
01 #include <complex> // benötigte Headerdatei
02 std::complex<double> dcompval{1.0, 2.0}; // C++11-Syntax
03 std::complex<float> fcompval(-1.6, -1.3);
```

Die Headerdatei `<complex>` in Zeile **01** ist immer nötig, wenn Sie komplexe Zahlen verwenden wollen. In Zeile **02** wurde eine komplexe Zahl vom Typ `double` mit einem

Realteil von 1.0 und einen Imaginärteil von 2.0 initialisiert. Hier wurde auch gleich die neue C++11-Initialisierungssyntax verwendet. Klappt diese bei Ihnen nicht, müssen Sie die geschweiften Klammern durch runde austauschen, wie dies mit der komplexen Zahl in Zeile 03 vom Typ `float` gemacht wurde.

Templates (Schablonen)

Das Thema Templates ist auch ein ganz wichtiges Thema in C++ und bekommt selbstverständlich ein extra Kapitel (Kapitel 11, »Template-Programmierung«) in diesem Buch.

Für komplexe Zahlen lassen sich alle arithmetischen Grundrechenoperatoren `+`, `-`, `*` und `/` wie gewohnt verwenden. Auch die Operatoren für Gleichheit oder Ungleichheit lassen sich benutzen. Wollen Sie bei den komplexen Zahlen nur den Real- oder den Imaginärteil, finden Sie hierfür die Methoden `real()` und `imag()` vor. Daneben stehen Ihnen viele weitere Methoden zur Verfügung. Hierzu ein Anwendungsbeispiel mit den komplexen Zahlen:

```
00 //kapitel02/komplexeZahlen1.cpp
01 #include <complex>
02 #include <cmath> // für atan()

03 std::complex<double> cdval1(1.33, 1.66);
04 std::complex<double> cdval2(0.66, 1.99);
05 std::complex<double> cdval3 = cdval1 + cdval2;
06 std::cout << cdval3 << std::endl;

07 std::cout << "Realteil      : " << cdval3.real()
    << std::endl;
08 std::cout << "Imaginärteil : " << cdval3.imag()
    << std::endl;

09 double pi = 4.0 * atan(1.0); // Pi berechnen lassen
10 double phase = pi/4.0; // 45°
11 double magnitude={0.0};
12 std::cout << "Magnitude eingeben: ";
13 std::cin >> magnitude;
14 cdval3 = std::polar(magnitude, phase);
15 std::cout << "Kartesische Zahl : " << cdval3
    << std::endl;
```

In den Zeilen 03 und 04 werden die komplexen Zahlen gleich mit einem Wert initialisiert. Würden Sie bei den komplexen Zahlen auf eine Initialisierung verzichten, so

würden diese automatisch mit $(0.0 + 0.0i)$ initialisiert. In Zeile 05 wird demonstriert, dass sich hiermit auch einfache arithmetische Berechnungen durchführen lassen. In den Zeilen 07 und 08 sehen Sie, wie Sie an den Real- und Imaginärteil des komplexen Typs kommen. Nach den vorbereiteten Werten in den Zeilen 10 bis 13 lassen wir in Zeile 14 eine komplexe kartesische Zahl aus den Polarkoordinaten berechnen. `std::polar()` von Zeile 14 ist eine Funktion aus `<complex>`, und `atan()` aus Zeile 09 gehört zu `<cmath>`.

Eine komplexe Zahl können Sie auch über die Tastatur und das Stream-Objekt `cin` einlesen:

```
00 //kapitel02/komplexeZahlen2.cpp
01 #include <complex>

02 std::complex<double> cdval;
03 std::cout << "Ein komplexe Zahl bitte (n.n, n.n):";
04 std::cin >> cdval;
05 std::cout << "Ihre Eingabe war : " << cdval << std::endl;
06 std::cout << cdval << std::endl;
```

Bei der Eingabe in Zeile 04 müssen Sie die komplexe Zahl in der Form $(1.2, 3.4)$ eingeben, also mit (!) den runden Klammern. Wenn die Eingabe für den »normalen« Anwender einfacher gehalten werden muss, können Sie stattdessen auch zunächst zwei `double`-Werte einlesen und dann den entsprechenden Real- und Imaginärteil einer komplexen Zahl übergeben.

Weitere Methoden

Wie bereits erwähnt, gibt es auch hier weitere Methoden, um mit komplexen Zahlen aus der Headerdatei `<complex>` arbeiten zu können. Hierfür empfehle ich Ihnen, bei Bedarf wieder in einer Referenz Ihrer Wahl nachzublättern (bspw. <http://en.cppreference.com>).

2.7.6 Nützliche mathematische Funktionen – `<cmath>`

Auf der Suche nach soliden trigonometrischen, hyperbolischen, exponentialen und weiteren mathematischen Funktionen dürften Sie in der Bibliothek von `<cmath>` fündig werden. `<cmath>` stammt noch aus klassischen C-Zeiten, aber trotzdem erfreuen sich die Funktionen darin immer noch großer Beliebtheit. Alle Funktionen können mit den Gleitkommatypen `float`, `double` und `long double` verwendet werden. Ein einfaches Beispiel hierzu:

```
01 #include <cmath>
02 double pi = 4.0 * atan(1.0);
03 std::cout << pi << std::endl;
```

Die Headerdatei in Zeile **01** ist unbedingt nötig, wenn Sie Funktionen aus `<cmath>` verwenden wollen. In Zeile **02** verwenden wir die Funktion `atan()` aus der Bibliothek von `<cmath>` und lassen in diesem Beispiel den Wert der Kreiszahl Pi berechnen.

Echte C++-Mathefunktionen

Sind Sie auf der Suche nach echten mathematischen C++-Funktionalitäten, so kann ich Ihnen empfehlen, sich die **Boost-Bibliothek** anzusehen. Dort sind eine Menge an Funktionalitäten zur Mathematik und Numerik enthalten. Mehr können Sie in den Dokumenten unter http://www.boost.org/doc/libs/?view=category_Math nachlesen.

Zwar waren für den C++11-Standard auch einige mathematische Funktionen geplant (siehe C++ Technical Report 1, 2011 oder kurz TR1 C++11; http://en.wikipedia.org/wiki/C%2B%2B_Technical_Report_1#Mathematical_special_functions), aber diese haben es nicht mehr rechtzeitig in den Standard geschafft. Das wird sich wohl bis zum nächsten Standard (im Augenblick **C++14** genannt) ändern, mit dem sie wohl eingeführt werden.

Vektorberechnungen

Sind Sie auf der Suche nach Dingen wie Vektorberechnungen, so sollten Sie sich die Template-Klasse `valarray` aus der gleichnamigen Headerdatei ansehen. Diese Klasse wird allerdings in diesem Buch nicht behandelt.

2.8 Zufallszahlen (neu in C++11)

Mit dem C++11-Standard wurde endlich eine eigene Bibliothek für Zufallszahlen eingeführt. Für das Erzeugen von Zufallszahlen mit der neuen Bibliothek sind zwei Dinge nötig:

1. Generator: Dieser Generator (engl. *Engine*) oder auch Zufallszahlenerzeuger erzeugt einen Zufallswert (hier ist auch die Rede von einem Strom von Zufallswerten) zwischen einem minimalen und maximalen Wert. Hierfür stehen einem mehrere vordefinierte Zufallswerte und Generatoren zu Verfügung.
2. Zufallszahlverteilung: Damit können Sie Zufallszahlen mithilfe des Generators mit unterschiedlichen Wahrscheinlichkeitsverteilungen erzeugen.

Das folgende Listing demonstriert Ihnen die einfachste Anwendung von Zufallszahlen. Einmal erzeugen wir Ganzzahlen und einmal Gleitkommazahlen:

```

00 // kapitel02/zufall.cpp
01 #include <random>
...
02 std::random_device rn;
03 std::mt19937 engine(rn());

04 std::uniform_int_distribution<int> dice(1, 6);
05 std::cout << "Yahtzee: ";
06 for(auto i=1; i<=5; ++i)
07 {
08     std::cout << dice(engine) << " ";
09 }
10 std::cout << std::endl;

11 std::uniform_real_distribution<double> randdouble(1.0, 10.0);
12 for(auto i=1; i<=5; ++i)
13 {
14     std::cout << randdouble(engine) << " ";
15 }
16 std::cout << std::endl;

```

Die Headerdatei `<random>` in Zeile **01** ist nötig, um die neuen Zufallszahlen mit C++11 verwenden zu können. Damit der Generator auch unterschiedliche Zufallszahlen zurückliefert, benötigen wir einen Startwert (engl. *seed*), welchen wir in Zeile **02** anlegen. In Zeile **03** wird der Generator mit diesem Startwert initialisiert. `mt19937` ist einer von mehreren vordefinierten Generatoren. Nachdem der Generator initialisiert wurde, richten wir die Zufallszahlenverteilung in Zeile **04** ein. Hier verwenden wir einen diskreten Verteiler für Ganzzahlen zwischen 1 und 6. In den Zeilen **06** bis **10** spielen wir das Würfelspiel *Yahtzee* (oder auch Kniffel), wobei in Zeile **08** der eingerichtete Zufallsgenerator in einer `for`-Schleife, die fünfmal durchlaufen wird, die Zufallszahlen produziert. Dasselbe wird nochmals in den Zeilen **11** bis **16** demonstriert, nur dass hier in Zeile **11** ein kontinuierlicher Verteiler für Gleitkommazahlen zwischen 1.0 und 10.0 eingerichtet wurde. Abbildung 2.3 zeigt das Programm bei der Ausführung.

Kurze Einführung

Das Thema Zufallszahlen wurde an dieser Stelle nur kurz angerissen und vereinfacht dargestellt, um Ihnen ein häufiges Rezept für zufällige Ganzzahlen zu liefern. Auch wurde hier wieder ein wenig auf Dinge wie u.a. Schleifen vorgegriffen, weshalb der Code gerade für Einsteiger noch recht komplex erscheinen dürfte.


```
J.Wolf $ ./zufall
Yahtzee: 6 2 3 1 1
9.11672 9.43956 7.7757 1.24791 1.00359
J.Wolf $ ./zufall
Yahtzee: 6 3 5 6 4
5.68185 8.58783 8.28707 1.76295 8.19792
J.Wolf $ ./zufall
Yahtzee: 5 2 3 1 1
1.36582 8.061 8.35013 4.43136 9.40699
J.Wolf $ █
```

Abbildung 2.3 Hier werden bei jedem Aufruf neue Zufallszahlen generiert.

2.9 Konvertieren von Typen

Jetzt kommen wir zu einem sehr wichtigen Thema. Und zwar geht es um das Mischen verschiedener Datentypen. Zwangsläufig muss beim Mischen verschiedener Datentypen einer der Operanden in einen neuen Zieltyp konvertiert werden. Bei solchen Konvertierungen kann es (zwangsläufig) zur Veränderung des Wertes kommen, wie dies bspw. der Fall ist, wenn Sie einen `double`-Wert nach `int` konvertieren, wo logischerweise die Nachkommastelle verworfen wird. Am idealsten und einfachsten wäre es, wenn Sie erst gar keine Konvertierung durchführen müssten und einfach dieselben Operanden verwenden könnten. Aber dies ist leider nicht immer machbar. Daher haben Sie die Möglichkeit, die Konvertierung entweder dem Compiler zu überlassen, der diese dann nach seinen Regeln durchführt, oder Sie greifen selbst ein und führen eine explizite Umwandlung des Datentyps durch. Mit C++11 ist es neuerdings auch möglich, die automatische Konvertierung zu beschränken, um ein versehentliches Konvertieren von bspw. `double` nach `int` zu unterbinden.

2.9.1 Automatische (implizite) Typumwandlung

Wenn Sie dem Compiler eine Typumwandlung überlassen, geht dieser wie folgt vor (Abbildung 2.4):

- ▶ Verwenden Sie eine Zuweisung, wird automatisch der Wert, der auf der rechten Seite des Zuweisungsoperators steht, in den Wert umgewandelt, der auf der linken Seite steht.
- ▶ Bei arithmetischen Berechnungen bzw. Vergleichen von zwei verschiedenen Typen wird automatisch immer der kleinere Datentyp in den größeren Datentyp umgewandelt.
- ▶ Wird an einer Funktion (siehe Abschnitt 5.5, »Funktionsparameter«) ein anderer Typ als Argument übergeben, wie in der Funktion vereinbart wurde, so wird automatisch in den Typ der Funktion konvertiert.

Verlust von Informationen (Narrowing, implizite Verengung)

Natürlich sollten Sie sich immer bewusst sein, dass es bei einer Typumwandlung zu Verlusten von Informationen kommen kann, umso ärgerlicher, wenn die Typumwandlung unbedacht oder versehentlich erfolgt ist. Dass der Nachkommateil einer Gleitkommazahl bei der Zuweisung an eine Ganzzahl nicht mehr vorhanden ist, sollte jedem klar sein. Schlimmer wird es dann schon, wenn der Quelltyp größer als der Zieltyp ist. Dies ist bspw. gegeben, wenn Sie ein `int` als Quelltyp einem `short` als Zieltyp zuweisen. Hier kann es zu einem Überschreiten des Wertebereichs kommen. Ebenfalls sehr problematisch ist das Mischen von Typen mit unterschiedlichen Vorzeichen (`signed`, `unsigned`), wobei es zu undefinierten Problemen kommen kann. Der Compiler warnt Sie gewöhnlich (hängt von der Einstellung ab), wenn Sie `signed` und `unsigned` bspw. bei Vergleichen mischen, da es hierbei passieren kann, dass ein Vergleich von einem `unsigned int`-Wert mit 4.294.967.295 und einem `signed int`-Wert von `-1` (`unsigned==signed`) gleich `true` zurückgibt, weil beide Bitdarstellungen identisch sind!

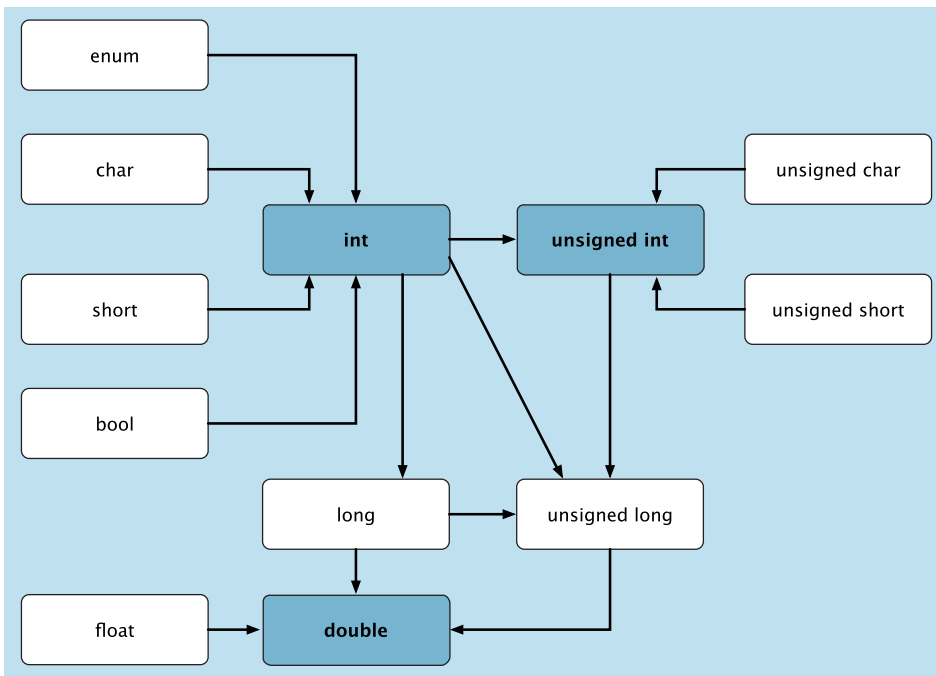


Abbildung 2.4 Hier wurden zwar nicht alle Typen aufgelistet, aber die Grafik soll zeigen, dass der Compiler bei der Verwendung von unterschiedlichen Typen immer versucht, in den nächsthöheren Typ umzuwandeln.

Hierzu nun ein einfaches Beispiel, welches diese impliziten Typumwandlungen in der Praxis zeigen soll:

```

00 // kapitel02/implizitkonvert.cpp
01 void funktion( int ival ) {
02     std::cout << "funktion(): " << ival << std::endl;
03 }
...
04 double dval = 123.123;
05 funktion(dval); //=123

06 int ival = dval; //=123
07 std::cout << "ival: " << ival << std::endl;

08 short sval1 = 32767;
09 std::cout << "sval1 + 1 = " << sval1+ 1 << std::endl;
10 short sval2 = sval1 + 1; // Bereichsüberschreitung
11 std::cout << "sval2 = " << sval2 << std::endl;

12 long lval = sval1 + 1;
13 std::cout << "lval = " << lval << std::endl;

```

In Zeile **05** wird ein `double`-Wert als Argument an die Funktion (Zeile **01** bis **03**) übergeben, welche allerdings als Argument einen `int`-Typ erwartet (siehe Zeile **01**). Der Compiler führt in der Funktion eine implizite Konvertierung durch, so dass in der Funktion nur noch der Integer-Wert 123 (Zeile **02**) ausgegeben wird. Der Nachkommateil wird verworfen. Selbiges geschieht in Zeile **06**, wo eine `double`-Wert 123.123 einem `int`-Typ zugewiesen wird. Der Compiler konvertiert auch hier implizit den `double`-Wert in einen `int`-Wert, und der Nachkommateil wird verworfen. Diese implizite Verengung des Typs wird auch als *Narrowing* bezeichnet.

In Zeile **08** erhält `sval1` den maximal möglichen Wert, den ein `short` speichern kann. Dass die arithmetische Berechnung in Zeile **09** trotzdem korrekterweise 32768 ausgibt und kein Überlauf stattfindet, liegt an der automatischen Promotion (= Ausweitung). Mehr zur automatischen Promotion folgt nach der Beschreibung des Listings. Zur Demonstration übergeben wir in Zeile **10** den maximalen Wert von `short` +1 der `short`-Variablen `sval`, was in diesem Fall dann, wie die Ausgabe in Zeile **11** zeigt, zu einem Wertüberlauf führt und ein Fehler im Programm ist. In Zeile **12** machen wir nochmals dasselbe, nur ist der Zieltyp hier `long`, und da `long` wesentlich breiter als `short` ist, gibt es mit dieser Typumwandlung überhaupt keine Probleme.

Narrowing vermeiden

Um diesen Abschnitt nicht falsch zu verstehen, folgt noch ein Hinweis. Sofern es irgendwie möglich ist, sollten Sie immer versuchen, ganz auf eine implizite Verengung (*Narrowing*) zu verzichten.

Vom Compiler bevorzugt ...

Wenn Sie Abbildung 2.4 betrachten, fragen Sie sich sicherlich, warum hier die Typen `int`, `unsigned int` und `double` hervorgehoben wurden. Es wurde zwar schon das eine oder andere Mal indirekt erwähnt, aber hierbei handelt es sich um eine spezielle Form der Umwandlung, die Sie als Programmierer gar nicht mitbekommen (und deren Ergebnis Sie in Listing *implizitkonvert.cpp* in Zeile 09 sehen konnten). Die Rede ist von einer automatischen Promotion (eingedeutscht so viel wie »Aufweitung«). Wenn Sie quasi eine arithmetische Operation durchführen, wird eine Promotion auf `int`, `unsigned int` oder `double` durchgeführt.

Hierbei gibt es eine integrale Promotion, bei der ein Typ wie `bool`, `char`, `signed char` oder `short` auf ein `int` bzw. `unsigned char`, `unsigned short` auf ein `unsigned int` »aufgeweitet« werden. Bei einer Gleitkommapromotion wird dann ein `float` nach `double` aufgeweitet. Dies hat den Vorteil, dass im Grunde mit dem natürlichsten Typ auf dem System gearbeitet wird, womit die arithmetischen Berechnungen eben am schnellsten sind.

Diese Typen werden nicht »aufgeweitet«

Keine Promotion findet statt, wenn einer der Operanden einen größeren Wertebereich wie `int`, `unsigned int` oder `double` besitzt, wie dies bspw. bei `long int`, `long long int` oder `long double` der Fall wäre.

2.9.2 Automatische Typumwandlung beschränken (C++11)

Wollen Sie eine automatische Verengung (*Narrowing*) von Datentypen einschränken, dann brauchen Sie einfach nur die neu in C++11 eingeführte einheitliche Initialisierung mit den geschweiften Klammern zu verwenden. Damit können Sie quasi versehentliche Konvertierungen abfangen. Folgendes Listing zeigt Ihnen, wie Sie das *Narrowing* im neuen C++11-Standard einschränken können:

```
00 // kapitel02/nonarrowing.cpp
01 double dval = 123.123;
02 int ival={dval}; // Fehler !!! Narrowing

03 short sval1 = {32767};
04 short sval2 = {sval1 + 1}; // Fehler !!! Narrowing

05 long lval = {sval1 + 1}; // OK!
```

In Zeile 02 findet ein *Narrowing* von `double` nach `int` statt. Die Daten von `dval` passen aber nicht komplett in `ival` und da hier die neue einheitliche Initialisierungsliste verwendet wurde, ist es jetzt ein Fehler. Selbiges gilt in Zeile 04, wo der Ausdruck `sval1+1`

zwischen den geschweiften Klammern nicht mehr in `sval2` passt. Die Konvertierung in Zeile 05 hingegen ist problemlos möglich.

Einheitliche Initialisierung bevorzugen

Mit der einheitlichen Initialisierung zwischen den geschweiften Klammern werden versehentliche und unschöne Verengungen von Typen vermieden, daher empfehle ich Ihnen, diese künftig auch regelmäßig zu verwenden. Und wenn Sie trotzdem mal ein *Narrowing* benötigen, dann sollten Sie den Typ explizit konvertieren, um zu zeigen, dass Sie sich hier im Klaren sind, was Sie tun. Die explizite Typumwandlung ist Thema des nächsten Abschnitts. Zu diesem Thema muss allerdings noch hinzugefügt werden, dass viele Compiler das *Narrowing* zwischen den geschweiften Klammern noch nicht (richtig) beherrschen. Aber viele Compiler geben mindestens eine Warnmeldung aus, dass ein *Narrowing* stattfindet.

2.9.3 Explizite Typumwandlung

Da Sie jetzt wissen, wie der Compiler implizit vorgeht, um Typen zu konvertieren, was passiert, wenn Sie Typen mischen, und wie Sie eine (versehentliche) implizite Verengung (*Narrowing*) vermeiden können, kommen wir jetzt zur expliziten Typumwandlung. Mit ihr können Sie als Programmierer zeigen, dass Sie genau wissen, was Sie tun, und dass es sich nicht um eine versehentliche Konvertierung handelt, bei der es eventuell zu einem Datenverlust kommen kann.

Durch vorausschauende Planung eine Typumwandlung vermeiden

Es kann nicht oft genug erwähnt werden: In den meisten Fällen lässt sich eine Typumwandlung mit ein wenig vorausschauender Planung des Codes vermeiden. Auch wenn C++ spezielle Operatoren für die Typumwandlung anbietet, ist es meistens trotzdem mehr eine Lösung »mit Gewalt« als »mit Gefühl«.

Hierzu nun eine Übersicht der verschiedenen Cast-Operatoren, welche C++ zur Verfügung stellt. Dabei wird immer der Wert von `ausdruck` in den TYP umgewandelt:

- ▶ `static_cast<TYP>(ausdruck)`
Das ist der Cast-Operator, den Sie für die typischen Standard-Typumwandlungen verwenden sollten. Dabei wird empfohlen, diesen Operator bei jeder Typumwandlung zu verwenden, damit deutlich wird, dass hier eine (gewollte) Typumwandlung stattfindet.
- ▶ `dynamic_cast<TYP>(ausdruck)`
Dieser Operator ist ähnlich wie `static_cast`, nur findet eine Überprüfung zur Laufzeit statt, wenn der Compiler bei der Übersetzung den Typ noch nicht kennen sollte.

Der Operator kann verwendet werden, um Zeiger oder Referenzen zwischen abgeleiteten Klassen umzuwandeln. Für Details dazu ist es an dieser Stelle noch zu früh.

► `const_cast<TYP>(ausdruck)`

Mit diesem Operator können Objekte, die mit `const` gekennzeichnet wurden und eigentlich nicht mehr geändert werden können, vorübergehend außer Kraft gesetzt werden. In der Praxis kann dies bspw. »nützlich« sein, wenn Sie einen konstanten Zeiger oder Referenzen an eine Funktion übergeben, welche keine konstanten Daten erwartet. Dieser Operator ist mit Vorsicht zu genießen, weil es keine Garantie gibt, dass die konstanten Daten dann tatsächlich auch geändert werden können.

► `reinterpret_cast<TYP>(ausdruck)`

Dieser Cast-Operator ist wohl eher ein Exot und kann auf einer sehr tiefen Ebene die unterschiedlichsten Typumwandlungen durchführen. Allerdings hängt das Ergebnis auch häufig von der Implementierung des Operators ab.

Hierzu noch ein paar einfache Beispiele mit dem `static_cast`-Operator im Einsatz:

```
00 // kapitel02/staticcast.cpp
...
01 void funktion( int ival ) {
02     std::cout << "funktion(): " << ival << std::endl;
03 }
...
04 double dval = {123.123};
05 funktion(static_cast<int>(dval));
06 int ival=static_cast<int>(dval); //=123
07 std::cout << "ival: " << ival << std::endl;

08 short sval1 = {32767};
09 long lval = static_cast<long>(sval1 + 1);
10 std::cout << "lval = " << lval << std::endl;

11 char ch1 = 'Z';
12 std::cout << static_cast<int>(ch1) << std::endl;

13 unsigned int val1 = {4294967295};
14 unsigned short val2 = static_cast<short>(val1);
```

In Zeile **05** wird der `static_cast`-Operator verwendet, um den `double`-Wert in ein `int` zu konvertieren und als Argument der Funktion (Zeile **01** bis **03**) zu übergeben. Auch wenn der Operator nicht unbedingt nötig wäre, sollten Sie sich angewöhnen, diesen trotzdem zu verwenden, um deutlich zu machen, dass ein Typ konvertiert wird. Dieselbe Typumwandlung wurde auch noch bei der Zuweisung in Zeile **06** durchgeführt.

In Zeile 09 wird ebenfalls der `static_cast`-Operator verwendet, obwohl es auch hier nicht nötig gewesen wäre. Aber auch hier soll deutlich gemacht werden, dass eine Typumwandlung hier explizit gewollt wurde und es kein Versehen war. In Zeile 12 wird dank der Konvertierung eines `char` in ein `int` der dezimale Wert des Buchstaben 'Z' ausgegeben. Diese Konvertierung von `char` nach `int` dürfte wohl auch die einzige sinnvolle in diesem Beispiel gewesen sein. Bei allen anderen hätte man sich wohl eher auf einen einheitlichen Typ festlegen können.

Trotz aller Technik ist man nicht vor Informationsverlusten geschützt, wie Zeile 14 zeigen soll. Der Wert von `unsigned int` hat nun mal keinen Platz in `unsigned short`, und der Wert in `unsigned short` wird ein ganz anderer sein. Gegebenenfalls kann man hier noch den `static_cast` in einer einheitlichen C++11-Initialisierung zwischen geschweifte Klammern stellen, wo sich der Compiler dann über das *Narrowing* beschweren sollte.

Vorteile der C++-Cast-Operatoren

Wenn Sie schon unbedingt eine einfache Typumwandlung durchführen müssen, dann sollten Sie stets den `static_cast`-Operator verwenden und dies nicht dem Compiler mit der impliziten Umwandlung überlassen. Zum einen zeigen Sie damit an, dass Sie hier bewusst eine Umwandlung vornehmen wollen. Im Falle eines Fehlers mit seltsamen Ergebnissen werden Sie so auch schneller fündig, weil Sie nur nach `static_cast` im Code suchen müssen. Das hilft enorm bei der Lesbarkeit des Codes (auch wenn die Verwendung von `operator<TYP>(ausdruck)` zunächst etwas kryptisch wirken mag).

Alter C++-Cast im Funktionsstil

Der Vollständigkeit halber soll hier auch noch der alte C++-Cast im Funktionsstil erwähnt werden. Er ähnelt ein wenig dem alten C-Cast, nur dass dieser eher an einen Funktionsaufruf erinnert:

```
01 double dval = {123.123};  
02 int ival(dval); // C++-Cast im Funktionsstil  
03 int ival2 = (int)dval; // C-Style-Cast
```

Hier wird in Zeile 02 ein `double`-Wert in ein `int` umgewandelt. In der Praxis wird aber empfohlen, stattdessen den neueren `static_cast` zu verwenden. In der letzten Zeile wird der C-Style-Cast verwendet.

Kapitel 3

Kontrollstrukturen

Kontrollstrukturen werden in fast allen Programmiersprachen verwendet. Sie werden benötigt, um den Ablauf eines Programms zu steuern. Hierzu unterscheidet man zwischen bedingten Anweisungen (bzw. Verzweigungen) und Schleifen. Mithilfe der booleschen Algebra werden diese Steuerkonstrukte beeinflusst.

Um in Ihrem Programm auf verschiedene Zustände und Eingaben reagieren zu können, benötigen Sie Kontrollstrukturen. Bedingte Anweisungen (`if`) und Verzweigungen (`else`, `else if` bzw. `switch case`) gehören zusammen mit Schleifen (`while`, `do while` und `for`) zu den wichtigsten Bestandteilen in der (C++)-Programmierung. Auf die einzelnen Steuerkonstrukte wird in diesem Kapitel eingegangen.

An dieser Stelle sollen auch noch kurz die Begriffe *Anweisung(en)* und *Anweisungsblöcke* etwas genauer erläutert werden. Von einer Anweisung spricht man, wenn ein Code in einer Zeile mit einem Semikolon abgeschlossen wird. Folgender Codeauschnitt enthält bspw. zwei Anweisungen:

```
int val=100;                // 1. Anweisung
std::cout << val << std::endl; // 2. Anweisung
```

Mehrere solcher Anweisungen lassen sich in einem Anweisungsblock zusammenfassen, wie Sie dies ja auch schon von der `main()`-Funktion her kennen. Ein solcher Anweisungsblock beginnt mit einer sich öffnenden geschweiften Klammer `{` und endet mit einer sich schließenden geschweiften Klammer `}`:

```
{ // <- Anweisungsblock: Anfang
    int val=100;                // 1. Anweisung
    std::cout << val << std::endl; // 2. Anweisung
} // <- Anweisungsblock: Ende
```

3.1 Bedingte Anweisung

Eine bedingte Anweisung ist ein Abschnitt im Code, der nur unter einer bestimmten Bedingung ausgeführt wird. Eine solche Bedingung kann ein bestimmter Zustand

oder eine bestimmte Eingabe sein. Solch eine bedingte Anweisung besteht aus einer Bedingung und einem Anweisungsblock, der wiederum aus mehreren Anweisungen bestehen kann. Eine bedingte Anweisung lässt sich mit folgendem Pseudocode vereinfacht beschreiben:

```
falls Bedingung zutrifft (true ist), dann
    Anweisung(en)
Ende
```

In der C++-Programmiersprache wird eine solche Bedingung mit dem Schlüsselwort `if` wie folgt formuliert:

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

Hier gilt somit auch: Wenn die Bedingung zwischen den Klammern von `if` wahr ist (`true`, ungleich 0), werden die Anweisung(en) im darauffolgenden Anweisungsblock ausgeführt. Anschließend wird mit der Ausführung hinter dem Anweisungsblock fortgefahren. Trifft die Bedingung hingegen nicht zu (`false`; gleich 0), wird der Anweisungsblock mit den Anweisung(en) dahinter nicht ausgeführt und gleich hinter dem Anweisungsblock mit der Ausführung fortgefahren (Abbildung 3.1).

Logischer Ausdruck

Die runden Klammern hinter `if` müssen verwendet werden, weil es sich hierbei um einen logischen Ausdruck handelt. Logisch heißt hier ganzzahlig. Der Ausdruck der Bedingung kann hierbei einen beliebigen numerischen Wert annehmen, der nach `bool` konvertierbar sein muss. `true` (bzw. ungleich 0) wird immer zurückgegeben, wenn der logische Ausdruck wahr ist. `false` (bzw. gleich 0) wird zurückgegeben, wenn der logische Ausdruck falsch (bzw. unwahr) ist.

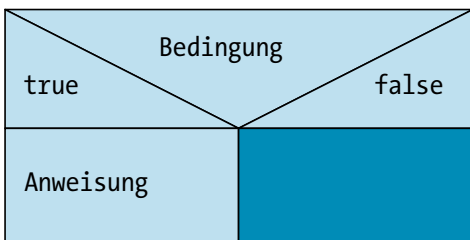


Abbildung 3.1 Struktogramm einer einfachen `if`-Anweisung

Hierzu ein einfacher Code, um die `if`-Anweisung in der Praxis zu testen:

```

00 // kapitel03/if01.cpp
...
01 int val={0};
02 std::cout << "Wert ungleich 0 eingeben: ";
03 std::cin >> val;
04 if(val) {
05     std::cout << val << std::endl;
06 }

```

In diesem Beispiel werden Sie aufgefordert, einen ganzzahligen Wert ungleich 0 einzugeben. Wenn Sie einen Wert ungleich 0 eingegeben haben, ist der logische Ausdruck in Zeile **04** gleich `true`, und der Wert wird im Anweisungsblock in Zeile **05** ausgegeben. Haben Sie hingegen den Wert 0 eingegeben, gibt der logische Ausdruck in `if` gleich `false` zurück, und die Anweisung im Anweisungsblock wird nicht ausgeführt.

Wie bereits eingangs erwähnt, ist es auch möglich, nicht nur einfache Datentypen in den logischen Ausdruck auszuwerten, sondern auch viele andere Zustände. Der eine oder andere wird jetzt vielleicht daran gedacht haben, Folgendes zu versuchen:

```

00 // kapitel03/if02.cpp
...
01 int val={0};
02 std::cout << "Wert ungleich 0 eingeben: ";
03 if(std::cin >> val) {
04     std::cout << val << std::endl;
05 }

```

Dies ist auch möglich, aber das Ergebnis ist jetzt ein wenig anders als in Listing *if01.cpp*. Der logische Ausdruck in Zeile **03** gibt immer `true` zurück, wenn Sie einen gültigen Ganzzahlwert eingegeben haben, wozu in diesem Fall auch 0 gehört! Hier wird also nicht mehr der Eingabewert selbst geprüft, sondern ob die Eingabe über `std::cin` korrekt gewesen ist. Der logische Ausdruck würde hier `false` zurückliefern, wenn Sie einen Buchstaben statt einer Ganzzahl eingeben würden. Mit diesem Konstrukt in Zeile **03** haben Sie also die Möglichkeit auf eine korrekte Eingabe über `std::cin` hin zu prüfen.

Ein Anweisungsblock ohne geschweifte Klammern

Wenn hinter einem `if` nur eine Anweisung folgt, kann der Anweisungsblock mit den geschweiften Klammern auch weggelassen werden. Nur mehrere Anweisungen müssen zu einem Block zusammengefasst werden.

Bezogen auf Listing *if02.cpp* könnten Sie theoretisch auch Folgendes schreiben:

```
if(std::cin >> val)
    std::cout << val << std::endl;
```

3.1.1 Vergleichsoperatoren

Um im logischen Ausdruck zu prüfen, ob etwas gleich, ungleich, größer oder kleiner einem entsprechenden Wert ist, benötigen Sie Vergleichsoperatoren. Hierzu müssen Sie lediglich rechts und links vom Vergleichsoperator einen Operanden setzen, zurückgegeben werden entweder `true` oder `false`, je nachdem, ob der Ausdruck wahr oder falsch war. Hierzu ein paar einfache Beispiele:

```
00 // kapitel03/vergleich.cpp
...
01 int ival01={2};
02 int ival02={10};

03 bool bval01 = ival01 < ival02; // = true
04 bool bval02 = ival01 != ival02; // = true
05 bool bval03 = ival01 == ival02; // = false
06 bool bval04 = ival02 == 10;    // = true
07 bool bval05 = ival01 <= 3;     // = true
```

Sollte der Codeausschnitt aus Listing *vergleich.cpp* noch nicht ganz verständlich sein, dann sollten diese letzten Ungereimtheiten mit der Auflistung und Erläuterung der einzelnen Vergleichsoperatoren in Tabelle 3.1 beseitigt werden.

Operator	Bedeutung	Beispiel	Rückgabewert
<	kleiner	<code>x < y</code>	true, wenn x kleiner als y, ansonsten false
<=	kleiner oder gleich	<code>x <= y</code>	true, wenn x kleiner als oder gleich y, ansonsten false
>	größer	<code>x > y</code>	true, wenn x größer als y, ansonsten false
>=	größer oder gleich	<code>x >= y</code>	true, wenn x größer als oder gleich y, ansonsten false
==	gleich	<code>x == y</code>	true, wenn x gleich y, ansonsten false
!=	ungleich	<code>x != y</code>	true, wenn x nicht gleich y ist, ansonsten false

Tabelle 3.1 Übersicht der Vergleichsoperatoren in C++

Vergleichsoperatoren mit Klassen

Hier sollte nicht der Eindruck entstehen, dass sich Vergleichsoperatoren nur für Ganzzahlen eignen. So finden Sie diese Operatoren auch in vielen Klassen implementiert. In der Containerklasse `std::string` bspw. können Sie die Vergleichsoperatoren für einen lexikalischen Vergleich zweier Zeichenketten verwenden. Natürlich ist es auch möglich, dass Sie in Ihren selbst geschriebenen Klassen später durch eine Operatorüberladung selbst die Vergleichsoperatoren implementieren können, sofern dies sinnvoll erscheint.

Bezogen auf Listing *if01.cpp*, in dem wir überprüft haben, ob der eingegebene Wert ungleich 0 bzw. `false` war, könnten Sie den logischen Ausdruck mit dem Vergleichsoperator `!=` jetzt wie folgt ändern:

```
00 // kapitel03/if01.cpp
...
04 if(val != 0) { // eingegebener Wert in val ungleich 0?
05     std::cout << val << std::endl;
06 }
```

Vom Prinzip her hat sich allerdings in diesem Beispiel nichts geändert. Aber Sie können mit den Vergleichsoperatoren jetzt bspw. gezielter auf die Eingabe bestimmter Werte reagieren. Folgendes Beispiel soll dies demonstrieren:

```
00 // kapitel03/if03.cpp
...
01 int val={0};
02 std::cout << "Wert von 1 und 10: ";
03 if(!(std::cin >> val)) {
04     std::cerr << "Fehler bei der Eingabe\n";
05     return 1; // Programmende
06 }
07 if( val >= 1 ) {
08     if( val <= 10 ) {
09         std::cout << val << std::endl;
10     }
11 }
```

In diesem Beispiel überprüfen wir zunächst in den Zeilen **03** bis **06**, ob überhaupt eine echte Ganzzahl eingegeben wurde. In Zeile **03** wird zunächst der Ausdruck mit `(std::cin >> val)` überprüft, wobei `true` zurückgegeben wird, wenn die Eingabe von `std::cin` korrekt war. Da wir hier aber überprüfen wollen, ob der Ausdruck nicht `true`, sondern `false` war, negieren wir diesen Ausdruck mit dem NOT-Operator `!` innerhalb von `if`. Mehr zu diesem logischen NOT-Operator erfahren Sie gleich im nächsten Abschnitt.

Ist der logische Ausdruck `!(std::cin >> val)` in Zeile 03 gleich `true`, dann wurde keine gültige Ganzzahl eingegeben, und wir geben eine entsprechende Meldung (Zeile 04) auf dem Bildschirm aus und beenden die Funktion in Zeile 05 mit `return`, was innerhalb der `main()`-Funktion auch gleichzeitig das Ende vom Programm ist.

Wurde eine korrekte Ganzzahl eingegeben, überprüfen wir in Zeile 07, ob der eingegebene Wert von `val` größer oder gleich 1 war. Liefert dieser Ausdruck `true` zurück, wird der Anweisungsblock ausgeführt, in dem wir in Zeile 08 eine weitere `if`-Bedingung verschachtelt haben, welche überprüft, ob der eingegebene Wert von `val` kleiner oder gleich 10 war. Wird auch hier `true` zurückgegeben, dann wird die Anweisung in Zeile 09 innerhalb des nächsten Anweisungsblocks ausgeführt, wo der eingegebene Wert von `val` ausgegeben wird – was ja hier nur der Fall ist, wenn `val` einem Wert von 1 bis 10 entspricht.

Das Beispiel wurde hier (bewusst) recht umständlich gelöst und kann selbstverständlich wesentlich einfacher und komfortabler mit den logischen Operatoren realisiert werden.

Vermeiden Sie, wenn möglich, ein Verschachteln von `if`-Anweisungen

Von verschachtelten `if`-Verzweigungen wird, wenn möglich, abgeraten, weil es das Leben bei der Fehlersuche ziemlich erschweren kann. Eine Verschachtelung ist in der Praxis auch selten unbedingt nötig und kann durch ein sorgfältiges Planen des Codes fast immer vermieden werden.

Fehlerquelle = `vs. ==` – verwenden Sie *Yoda Conditions*

An dieser Stelle muss ich noch auf einen häufigen Fehler hinweisen. Und zwar wird immer wieder gerne eine Bedingung wie folgt formuliert:

```
if( val = 100 ) {...}
```

Gewollt war hier sicherlich ein Vergleich mit `==`, ob der Wert von `val` gleich 100 ist und keine Zuweisung von 100 an `val`. Ein guter Compiler sollte hier ohnehin eine Warnung ausgeben. An dieser Stelle könnte ich Ihnen eine *Yoda Condition* empfehlen. Gemäß Yoda: »Vermeiden Fehler du musst« drehen wir auch nur den Inhalt des Ausdrucks um:

```
if( 100 == val ) {...}
```

Würden Sie hier wieder mal aus Versehen den Zuweisungsoperator statt des `==`-Vergleichsoperators verwenden, würde der Compiler einen Fehler ausgeben, weil einem konstanten Wert wie 100 kein Wert zugewiesen werden kann.

3.1.2 Logische Operatoren

Das Beispiel in Listing *if03.cpp* ruft geradezu nach einer Möglichkeit, zwei Bedingungen in `if` zu überprüfen. Um mehrere Bedingungen zu verknüpfen, können Sie logische Operatoren verwenden. In Tabelle 3.2 finden Sie eine Auflistung der logischen Operatoren.

Logischer Operator	Logische Bedeutung
<code>&&</code>	logisches Und (Konjunktion)
<code> </code>	logisches ODER (Disjunktion)
<code>!</code>	logisches NICHT (Negation)

Tabelle 3.2 Die logischen Operatoren in C++

Wenn Sie logische Verknüpfungen erstellen, entsteht ein komplexer logischer Ausdruck. Je nach Art der Verknüpfung liefert dieser mit den logischen Operatoren zusammengesetzte Ausdruck wahr (`true`) oder eben unwahr (`false`) zurück.

Logische Algebra

In der logischen Algebra werden in der Aussagenlogik 0 als falsch (bzw. unwahr) und 1 als wahr interpretiert. In C++ werden für diese booleschen Ausdrücke `false` (für falsch bzw. unwahr) und `true` (für wahr) verwendet.

Mehrere Ausdrücke verknüpfen

Im Grunde können Sie beliebig viele Ausdrücke mit den logischen Operatoren verknüpfen und sie auch miteinander kombinieren. Allerdings sollten Sie beachten, dass ein logischer Ausdruck mit mehr als zwei logischen Verknüpfungen erheblich komplexer werden kann.

Logisches UND (Konjunktion)

Beim logischen UND-Operator wird bspw. nur dann `true` zurückgegeben, wenn alle verknüpften Ausdrücke ebenfalls `true` (also wahr) sind. Wenn nur einer der verknüpften Ausdrücke `false` zurückgibt (unwahr ist), wird der komplette verknüpfte Ausdruck auch `false` (unwahr) zurückgeben. Ein einfaches Beispiel einer solchen UND-Verknüpfung:

```
bool bval = (ausdruck01) && (ausdruck02);
```

Die boolesche Variable `bval` ist nur dann `true`, wenn beide Ausdrücke `ausdruck01` **und** `ausdruck02` gleich `true` zurückgeben. Ist nur einer der beiden Ausdrücke `false` (unwahr), erhält `bval` ebenfalls den Wert `false`.

Unser Beispiel aus Listing *if03.cpp* vom Abschnitt zuvor, in dem wir mit zwei verschachtelten `if`-Bedingungen überprüft haben, ob eine Ganzzahl mit dem Wert von 1 bis 10 eingegeben wurde, können Sie mit einer logischen UND-Verknüpfung wie folgt realisieren:

```
00 // kapitel03/if04.cpp
...
01 if( (val >= 1) && (val <= 10) ) {
02     std::cout << val << std::endl;
03 }
```

In Zeile **01** finden Sie jetzt die logische UND-Verknüpfung innerhalb unserer `if`-Anweisung zwischen zwei Ausdrücken. Hier prüfen wir im linken Ausdruck, ob der eingegebene Wert von `val` größer oder gleich 1 ist, **und** im rechten Ausdruck, ob `val` kleiner gleich 10 ist. Treffen beide logischen mit UND verknüpften Ausdrücke zu, liefern also beide Ausdrücke `true` zurück, ist der komplexe verknüpfte Ausdruck auch `true` (wahr), und die Anweisung aus Zeile **02** im folgenden Anweisungsblock wird ausgeführt. Die Aussagenlogik des logischen UND-Operators schlüsselt Ihnen Tabelle 3.3 auf.

Var_A	Var_B	Var_A && Var_B
true	true	true
true	false	false
false	true	false
false	false	false

Tabelle 3.3 Die Aussagenlogik des logischen UND-Operators

Logisches ODER (Disjunktion)

Beim logischen ODER-Operator wird bereits `true` zurückgegeben, wenn mindestens einer der verknüpften Ausdrücke gleich `true` (wahr) ist. `false` wird nur dann zurückgegeben, wenn alle verknüpften Ausdrücke `false` (unwahr) sind.

```
bool bval = (ausdruck01) || (ausdruck02);
```

Hier wird `bval` gleich `true`, wenn der `ausdruck01` **oder** `ausdruck02` `true` zurückgeben. `bval` ist nur dann `false`, wenn sowohl `ausdruck01` als auch `ausdruck02` beide `false` (unwahr) sind (Tabelle 3.4).

Var_A	Var_B	Var_A Var_B
true	true	true
true	false	true
false	true	true
false	false	false

Tabelle 3.4 Die Aussagenlogik des logischen ODER-Operators

Logisches NICHT (Negation)

Mit dem logischen NICHT-Operator wird ein Ausdruck negiert. Damit können Sie aus einem wahren Ausdruck einen falschen machen und aus einem falschen wiederum einen wahren. Der Operator wird gerne dazu verwendet, einen Ausdruck auf `false` zu testen, also eine wesentlich verständlichere Abkürzung für:

```
if( val == false ) // oder schlimmer: if( val == 0)
```

Anstatt auf `false` (oder gar `0`) zu prüfen, sollten Sie dies mit dem NICHT-Operator wie folgt schreiben:

```
if( !val )
```

if(!val) vs. if(val == 0)

Es wird davon abgeraten, einen Vergleich auf »gleich 0« oder »ungleich 0« direkt hinzuschreiben (wie bspw. `if(val==0)` oder `if(val!=0)`), weil es beim logischen Ausdruck immer um eine implizite Konvertierung nach `bool` geht (und gehen sollte). Das gilt hier dann auch für die direkte Verwendung von `0`, welche letztendlich (implizit) auch wieder (von `int`) in ein `bool` konvertiert werden muss. Zudem macht die direkte Verwendung von `0` bzw. ungleich `0` das Verstehen eines logischen Ausdrucks oft nur noch komplizierter:

```
if(val==0){} // Hm, was ist hier wohl gemeint?
if(!val){}   // Erscheint mir viel logischer
```

In unserem Beispiel in Listing *if03.cpp* haben Sie diesen logischen NICHT-Operator schon im Einsatz gesehen, als wir den Eingabe-Stream `std::cin` überprüft haben:

```
00 // kapitel03/if03.cpp
...
01 if( !(std::cin >> val) ) {
```



```
02  std::cerr << "Fehler bei der Eingabe\n";
03  return 1; // Programmende
04  }
```

Wenn die Eingabe von `std::cin` korrekt war (also eine gültige Ganzzahl eingegeben wurde), gibt der Ausdruck `std::cin >> val` zwischen den innersten Klammern gleich `true` zurück. Betrachten Sie den folgenden Anweisungsblock dahinter mit der Fehlerausgabe und dem Programmende, dann würde der logische Ausdruck ohne den `!`-Operator mit dem dorthin geschriebenen Code keinen Sinn ergeben. Daher negieren wir diesen Ausdruck mit dem `!`-Operator mit `!(std::cin >> val)`, womit geprüft wird, ob der logische Ausdruck `std::cin >> val` gleich `false` ist – was bspw. der Fall wäre, wenn Sie einen Buchstaben statt einer Ganzzahl eingeben würden. Die Aussagenlogik des logischen NICHT-Operators schlüsselt Ihnen Tabelle 3.5 auf.

Var_A	!Var_A
true	false
false	true

Tabelle 3.5 Die Aussagenlogik des logischen NICHT-Operators

3.1.3 Verzweigung (if-else-Anweisung)

Mit einer Verzweigung (auch als Auswahl oder Selektion bekannt) können Sie zusätzlich zu einer vorher formulierten Bedingung noch festlegen, dass ein zweiter Codeabschnitt nur dann ausgeführt wird, wenn die vorher formulierte Bedingung nicht ausgeführt wurde. Natürlich ist es auch möglich, mehrere solche Bedingungen zu formulieren. Eine solche mehrfache Verzweigung (auch Fallunterscheidung genannt) lernen Sie in Abschnitt 3.2, »Fallunterscheidung (mehrfache Verzweigung)«, kennen.

Ein Pseudocode mit einer Verzweigung kann wie folgt formuliert werden:

```
falls Bedingung zutrifft (true ist), dann
    Anweisung(en)
sonst
    Anweisung(en)
Ende
```

In C++ wird eine solche Verzweigung immer mit dem Schlüsselwort `if` für die Bedingung eingeleitet. Erst dann kann die alternative (und optionale) `else`-Verzweigung erfolgen. Eine `else`-Verzweigung kann sinngemäß nur dann verwendet werden, wenn dieser eine `if`-Bedingung vorausgeht. In C++ lässt sich diese Verzweigung wie folgt formulieren:

```

if( Bedingung ) {
    Anweisung(en)1;
}
else {
    Anweisung(en)2;
}
Anweisung(en);

```

Hier gilt somit auch, wenn die Bedingung zwischen den Klammern von `if` wahr ist (`true`, ungleich 0), werden die `Anweisung(en)` im darauffolgenden Anweisungsblock ausgeführt. Trifft die `if`-Bedingung allerdings nicht zu und gibt `false` (bzw. 0) zurück, werden die Anweisungen im Anweisungsblock der alternativen `else`-Verzweigung ausgeführt. Nachdem mindestens einer der beiden Codeabschnitte ausgeführt wurde, wird mit der Programmausführung hinter der `if-else`-Anweisung fortgefahren (Abbildung 3.2).

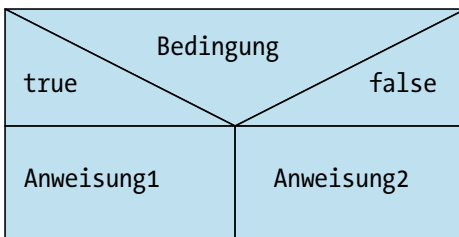


Abbildung 3.2 Struktogramm einer `if-else`-Anweisung

Hierzu soll jetzt nochmals unser Beispiel aus Listing *if03.cpp*, in dem Sie einen Wert von 1 bis 10 eingeben sollten, erweitert werden. Mit der `else`-Verzweigung wollen wir einen alternativen Code einfügen, der ausgeführt wird, wenn der eingegebene Wert von `val` nicht 1 bis 10 gewesen ist:

```

00 // kapitel03/if04.cpp
...
01 if( (val >= 1) && (val <= 10) ) {
02     std::cout << val << std::endl;
03 }
04 else {
05     std::cout << "Ungültiger Wertebereich: "
                << val << std::endl;
06 }

```

Die Zeilen **01** bis **03** sind Ihnen ja noch aus Listing *if03.cpp* bekannt, wo überprüft wird, ob Sie einen Wert von 1 bis 10 für `val` eingegeben haben. Gibt diese Bedingung in Zeile **01** `false` zurück, wird jetzt die alternative `else`-Verzweigung (Zeilen **04** bis **06**)

ausgeführt und gibt jetzt einen Hinweis auf dem Bildschirm aus, dass der eingeebene Wert nicht den Anforderungen entspricht.

3.1.4 Bedingter Ausdruck (?:)

Der ternäre (= *dreielementige*) Auswahloperator `?:` ist im Grunde eine Kurzform der `if-else`-Anweisung. Die Syntax eines solchen bedingten Ausdrucks ist wie folgt formuliert:

Bedingung ? Ausdruck1 : Ausdruck2

Hierbei wird zunächst die Bedingung ausgewertet (was auch ein komplexer Ausdruck sein kann). Ist das Ergebnis der Bedingung gleich `true` (wahr), wird der Ausdruck1 verwendet, ist die Bedingung hingegen `false` (falsch), wird Ausdruck2 verwendet.

Der folgende Codeausschnitt zeigt, wie Sie mithilfe des bedingten Ausdrucks den größeren von zwei Werten ermitteln und einer Variablen zuweisen können:

```
00 // kapitel03/auswahloperator.cpp
...
01 int val1={0}, val2={0};
02 std::cout << "Wert 1: "; std::cin >> val1;
03 std::cout << "Wert 2: "; std::cin >> val2;
04 int max = (val1 > val2)?val1:val2;
05 std::cout << max << std::endl;
```

In Zeile **04** wird als Bedingung überprüft, ob `val1` größer als `val2` ist. Ist dies wahr (= `true`), wird `val1` an `max` zugewiesen. Trifft dies nicht zu (= `false`), wird `val2` an `max` zugewiesen.

Diese Schreibweise aus Zeile **04** ist äquivalent zu folgender `if-else`-Anweisung:

```
if(val01>=val02) {
    max = val01;
}
else {
    max = val02;
}
```

Trotzdem gibt es Beispiele, wo Sie mit dem ternären Operator etwas tun können, was sich mit der `if-else`-Anweisung nicht mehr realisieren lässt. Die folgende `const`-Initialisierung ist so ein Fall:

```
int val1=0, val2=0;
const int max = (val1 > val2)?val1:val2;
```

3.2 Fallunterscheidung (mehrfache Verzweigung)

In C++ haben Sie zwei Möglichkeiten, eine mehrfache Verzweigung (häufig auch als Fallunterscheidung bezeichnet) zu implementieren. Sie können entweder die `switch`-Fallunterscheidung oder mehrere `if-else-if`-Bedingungen formulieren.

3.2.1 Fallunterscheidung mit `switch`

Die Fallunterscheidung mit `switch` kann zur Auswertung eines ganzzahligen Ausdrucks verwendet werden – es kann auch ein *Enum* verwendet werden (siehe Abschnitt 4.7, »Aufzählungstypen (C++11-Version)«). Ganzzahliger Ausdruck bedeutet hier, dass dieser vom Typ `int` (oder in ein `int` konvertierbarer Typ, wie bspw. `char`, `short`, `long`) sein muss. Ein Pseudocode dazu lässt sich wie folgt formulieren:

```
falls der Ausdruck gleich
    Wert1 ist : Anweisung(en)
    Wert2 ist : Anweisung(en)
    Wert3 ist : Anweisung(en)
    ...
    sonst      : Anweisung(en)
Ende
```

Als C++-Code lässt sich dieser Pseudocode mit `switch` wie folgt implementieren:

```
switch( Ausdruck ) {
    case Wert1 : Anweisung(en); break;
    case Wert2 : Anweisung(en); break;
    case Wert3 : Anweisung(en); break;
    ...
    default    : Anweisung(en);
}
```

Innerhalb von `switch` wird der ganzzahlige Ausdruck ausgewertet und mit den ganzzahligen Konstanten der folgenden `case`-Marken verglichen. Wurde eine Übereinstimmung gefunden, werden die entsprechenden Anweisungen hinter den passenden `case`-Marken ausgeführt. Sämtliche `case`-Marken werden als Anweisungsblock zwischen geschweiften Klammern zusammengefasst. Stimmt keine der `case`-Marken mit dem ganzzahligen Ausdruck überein, werden die Anweisungen der (optionalen) `default`-Marke ausgeführt (Abbildung 3.3).

break

Die `break`-Anweisung am Ende einer `case`-Marke ist sehr wichtig. Mithilfe dieser `break`-Anweisung sorgen Sie dafür, dass die Programmausführung aus dem `switch`-Anweisungsblock herausspringt und dahinter mit der Programmausführung fortfährt.

Ein Vergessen einer solchen `break`-Anweisung ist keine Fehler (und kann auch gewollt sein), aber dies bedeutet auch, dass alle anderen Anweisungen ab dieser `case`-Marke ebenfalls bis zur nächsten `break`-Anweisung ausgeführt werden. Ist überhaupt keine `break`-Anweisung vorhanden, werden alle weiteren Anweisungen bis zum Ende des Anweisungsblocks ausgeführt (auch von den `case`-Marken, welche nicht zutreffend sind). Wie gesagt, es kann durchaus gewollt sein, dass Sie alle Anweisungen ab einer bestimmten `case`-Marke abarbeiten wollen.

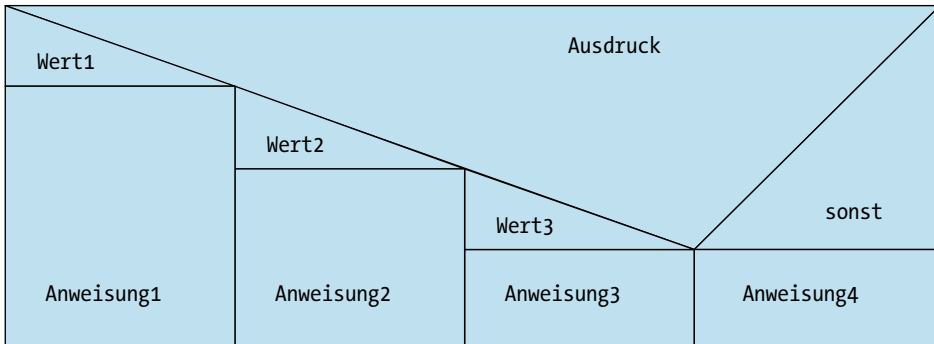


Abbildung 3.3 Struktogramm einer `switch`-Anweisung

Hierzu ein relativ einfacher Codeausschnitt, welcher die `switch`-Fallunterscheidung kurz demonstrieren soll:

```
00 // kapitel03/switch.cpp
...
01 int choice={0};
02 std::cout << "Bitte die Sprache wählen\n";
03 std::cout << "-1- Deutsch\n";
04 std::cout << "-2- Englisch\n";
05 std::cout << "-3- Spanisch\n";
06 std::cout << "Ihre Auswahl : ";
07 if( ! (std::cin >> choice) ) {
08     std::cout << "Fehler bei der Eingabe\n";
09     return 1;
10 }

11 switch( choice ) {
12     case 1 : std::cout << "Willkommen\n";
13             break;
14     case 2 : std::cout << "Welcome\n";
15             break;
16     case 3 : std::cout << "Bienvenido\n";
```

```

17         break;
18     default:  std::cout << choice
                << ":Ungültige Eingabe\n";
19 }

```

Das Beispiel ist sehr trivial, aber ideal, um die `switch`-Fallunterscheidung zu beschreiben. In den Zeilen 01 bis 10 werden Sie aufgefordert, einen Wert von 1–3 einzugeben. Diesen Wert prüfen wir in Zeile 11 als Ausdruck in `switch`. Haben Sie den Wert 1 eingegeben, werden die Anweisungen hinter der `case`-Marke aus Zeile 12 bis zur nächsten `break`-Anweisung ausgeführt. Mit `break` springen Sie aus dem `switch`-Anweisungsblock heraus, und die Programmausführung wird dahinter fortgeführt.

Haben Sie bei der Aufforderung der Eingabe den Wert 2 für `choice` eingegeben, werden die Anweisungen ab der `case`-Marke in Zeile 14 bis zum nächsten `break` ausgeführt. Dasselbe gilt auch für die Eingabe des Wertes 3 und der `case`-Marke in Zeile 16. Haben Sie hingegen einen Wert ungleich 1, 2 oder 3 eingegeben, werden die Anweisungen hinter der `default`-Marke in Zeile 18 ausgeführt.

Würden Sie bei diesem Beispiel die `break`-Anweisung in Zeile 13 entfernen, würden auch die Anweisungen in Zeile 14 ausgeführt und erst mit `break` in Zeile 15 wieder beendet. Würden Sie alle `break`-Anweisungen entfernen, würden alle Anweisungen ab der ersten zutreffenden `case`-Marke ausgeführt. Probieren Sie es einfach selbst aus.

Randnotiz

Natürlich lassen sich hierbei beliebig viele `case`-Marken verwenden.

3.2.2 Mehrfache Verzweigung mit `else-if`-Anweisung(en)

Ein zweite Form der mehrfachen Verzweigung können Sie mit `else if` formulieren. Dabei werden so lange Bedingungen ausgewertet, bis eine gleich `true` (wahr) ist und der Codeabschnitt dieser Bedingung ausgeführt wird. Trifft keine der Bedingungen zu, kann optional auch hier wieder eine alternative `else`-Verzweigung am Ende verwendet werden. Ein Pseudocode dazu sieht wie folgt aus:

```

falls Bedingung1 zutrifft (true ist), dann
    Anweisung(en)
sonst falls Bedingung2 zutrifft (true ist), dann
    Anweisung(en)
sonst
    Anweisung(en)
Ende

```

In C++ lässt sich diese mehrfache Verzweigung mit `else if` wie folgt formulieren:

```
if( Bedingung1 ) {
    Anweisung(en)1;
}
else if( Bedingung2 ) {
    Anweisung(en)2;
}
else {
    Anweisung(en)3;
}
Anweisung(en);
```

Wenn die `Bedingung1` zwischen den Klammern von `if` wahr ist (`true`, ungleich 0), werden die `Anweisung(en)` im darauffolgenden Anweisungsblock ausgeführt. Trifft die `if`-Bedingung allerdings nicht zu und gibt `false` (bzw. 0) zurück, wird als Nächstes die `Bedingung2` zwischen den Klammern von `else if` überprüft. Auch hier gilt, wenn diese `Bedingung2` wahr ist (`true`, ungleich 0), werden die Anweisungen im folgenden Anweisungsblock ausgeführt. Ist auch `Bedingung2` falsch und gibt `false` zurück, werden die Anweisungen im Anweisungsblock der (optionalen) alternativen `else`-Verzweigung ausgeführt. Am Ende der mehrfachen Verzweigung wird dahinter mit der Programmausführung fortgefahren.

Randnotiz

Die Verkettung von `else-if`-Bedingungen kann beliebig um weitere Bedingungsüberprüfungen erweitert werden. Entscheidend ist nur, dass die erste Anweisung eine reine `if`-Anweisung ist, und (optional) kann die letzte Anweisung eine `else`-Verzweigung sein.

Hierzu soll das Beispiel aus Listing *switch.cpp* als gleichwertige Alternative in einer mehrfachen Verzweigung mit `else-if`-Anweisungen umgeschrieben werden:

```
00 // kapitel03/elseif.cpp
...
01 int choice={0};
02 std::cout << "Bitte die Sprache wählen\n";
03 std::cout << "-1- Deutsch\n";
04 std::cout << "-2- Englisch\n";
05 std::cout << "-3- Spanisch\n";
06 std::cout << "Ihre Auswahl : ";
07 if( !(std::cin >> choice) ) {
08     std::cout << "Fehler bei der Eingabe\n";
```

```

09     return 1;
10 }

11 if( 1 == choice ) {
12     std::cout << "Willkommen\n";
13 }
14 else if( 2 == choice ) {
15     std::cout << "Welcome\n";
16 }
17 else if( 3 == choice ) {
18     std::cout << "Bienvenido\n";
19 }
20 else {
21     std::cout << choice << " : Ungültige Eingabe\n";
22 }

```

Das Programm macht in den Zeilen 11 bis 22 nichts anderes als Listing *switch.cpp* in den Zeilen 11 bis 19.

3.3 Schleifen (Wiederholungen)

Eine Schleife ist eine weitere Form einer Kontrollstruktur, womit Sie Anweisungen, zusammengefasst in einem Anweisungsblock (auch Schleifenrumpf, Schleifenkörper genannt), mehrmals wiederholen können. In der Regel wird eine Schleife so lange bzw. so oft wiederholt, wie eine Schleifenbedingung wahr ist. Es ist aber auch möglich, Schleifen ohne eine Schleifenbedingung zu formulieren, welche dann endlos ausgeführt werden – sogenannte Endlosschleifen.

C++ bietet hierbei drei (mit C++11 vier) Formen von Schleifen an:

- ▶ **Kopfgesteuerte Schleife** (`while()`): Bei dieser Schleife wird die Schleifenbedingung überprüft, bevor der Schleifenrumpf ausgeführt wird.
- ▶ **Fußgesteuerte Schleife** (`do while()`): Hier wird die Schleifenbedingung erst nach dem Durchlauf des Schleifenrumpfs überprüft.
- ▶ **Zählschleife** (`for()`): Im Grunde handelt es sich hierbei um eine besondere Form der kopfgesteuerten Schleife.
- ▶ **Mengenschleife** (`for()`): Dies ist eine besondere Form der `for()`-Schleife, welche auch als `foreach`-Schleife aus anderen Programmiersprachen bekannt ist, womit jedes Element einer Menge durchlaufen wird. Diese Range-basierte `for`-Schleife wurde neu in C++11 eingeführt.

3.3.1 Z hloperatoren (Inkrement und Dekrement)

Bevor Sie Schleifen verwenden, sollen hier noch der Inkrement- und Dekrement-Operator vorgestellt werden, welche beide h ufig mit den Schleifen verwendet werden (Tabelle 3.6). Beide Operatoren haben die Aufgabe, einen Wert um 1 zu erh hen (inkrementieren) oder verringern (dekrementieren).

Operator	Name	Alternative 1	Alternative 2
++	Inkrement	var+=1;	var=var+1;
--	Dekrement	var-=1;	var=var-1;

Tabelle 3.6 Inkrement- und Dekrement-Operator in C++

Diese Operatoren k nnen Sie sowohl mit Ganzzahlen als auch mit Kommazahlen verwenden. Allerdings wird bei den Kommazahlen nur der Wert vor dem Komma erh ht bzw. verringert. Der Operator wird einseitig verwendet und hat somit nur einen Operanden. Dabei kann der Operator rechts (Postfix-Schreibweise) oder links (Pr fix-Schreibweise) vom Operanden stehen (Tabelle 3.7).

Schreibweise	Inkrement	Dekrement
Postfix	var++	var--
Pr�fix	++var	--var

Tabelle 3.7 Postfix- und Pr fix-Schreibweise der Z hloperatoren

Es gibt auch einen Unterschied bei der Postfix- und Pr fix-Schreibweise, der hier noch erw hnt werden muss. In der Postfix-Schreibweise wird der Wert um 1 erh ht bzw. verringert, bleibt aber in der aktuellen Anweisung noch unver ndert. Genauer, das ++ bzw. -- nach dem Operanden wird ausgef hrt, **nachdem** der Operand zur ckgegeben wurde.

Bei der Pr fix-Schreibweise hingegen wird der Wert sofort um 1 erh ht bzw. verringert und somit im aktuellen Ausdruck auch gleich verwendet. Auch hier genauer: Das ++ bzw. -- vor dem Operanden wird ausgef hrt, **bevor** der Operand zur ckgegeben wird. Der folgende Codeausschnitt soll diesen Unterschied demonstrieren:

```
00 // kapitel03/postpraefix.cpp
...
01 int var={1};
02 std::cout << var++ << std::endl; // = 1
03 std::cout << var << std::endl;   // = 2
04 std::cout << ++var << std::endl; // = 3
```

Bei der Ausgabe in Zeile **01** wird noch der Wert 1 ausgegeben, weil der Wert von `var` erst nach der Anweisung inkrementiert wird – also nachdem der Operand zurückgegeben wurde. Die Ausgabe Zeile **02** beweist diese Aussage. In Zeile **03** wird die Präfix-Schreibweise verwendet, weshalb der Wert sofort um 1 erhöht wird, was die Ausgabe mit dem Wert 3 auch bestätigt. Analog wie im hier gezeigten Beispiel verhält es sich mit dem Dekrement-Operator zur Reduzierung des Wertes.

Unerwartete Seiteneffekte

Es ist keine gute Idee, eine Variable (oder Speicheradresse) in einer Anweisung mehrmals ändern zu wollen (bspw. zu inkrementieren). Dies kann zu unerwarteten Seiteneffekten führen, welche nicht vorhersehbar ist. Bei einem Code, wie z. B.

```
std::cout << v++ << v++ << v++;
std::cout << v;
```

ist es nicht garantiert, ob die Ausgabe hier 1234 oder 3214 ist (ausgehend davon, dass `v` anfangs mit 1 initialisiert wurde). Verschiedene Testsysteme haben verschiedene Ergebnisse geliefert. Einige Compiler geben, abhängig von der Warneinstellung, bei solch einem Code auch bereits eine Warnung aus.

3.3.2 Kopfgesteuerte Schleife – `while()`

Bei der `while`-Schleife wird der Schleifenrumpf mit Anweisungen so lange wiederholt, wie die Schleifenbedingung `true` (wahr) zurückliefert. Gewöhnlich wird der Inhalt des logischen Ausdrucks innerhalb einer Anweisung im Schleifenrumpf verändert, um diesen beim erneuten Anfang der Schleife wieder zu überprüfen. Hier der Pseudocode für die `while`-Schleife:

```
Wiederhole, solange Schleifenbedingung wahr ist
    Anweisung(en)
    Wert für Schleifenbedingung aktualisieren
Ende
```

In C++ können Sie eine `while`-Schleife wie folgt formulieren:

```
while( Schleifenbedingung ) {
    Anweisung(en);
}
Anweisungen(en)
```

Solange die Schleifenbedingung `wahr` (= `true`) ist, werden die `Anweisung(en)` im Schleifenrumpf ausgeführt. Wenn der Ausdruck von `while` hingegen `falsch` (= `false`) ist, werden die `Anweisung(en)` im Schleifenrumpf nicht mehr ausgeführt, und das Programm fährt hinter dem Schleifenrumpf mit seiner Ausführung fort (Abbildung 3.4).

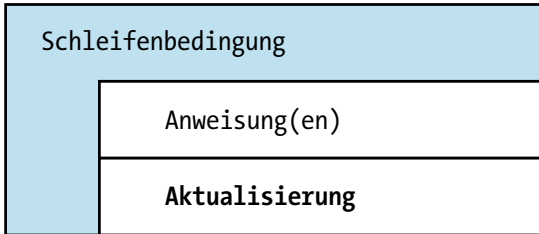


Abbildung 3.4 Struktogramm einer while-Schleife

Hierzu ein einfaches Beispiel, welches die `while`-Schleife in der Praxis demonstrieren soll:

```

00 // kapitel03/while.cpp
...
01 int summe={0}, val={0}, n={0}, i={0};
02 std::cout << "Wie viele Werte wollen Sie addieren: ";
03 std::cin >> n;
04 while( i < n ) {
05     std::cout << i+1 << "ter Summand: ";
06     if( !(std::cin >> val) ) {
07         std::cout << "Fehler bei der Eingabe\n";
08         return 1;
09     }
10     summe+=val;
11     ++i;
12 }
13 std::cout << "Summe aller Werte: " << summe << std::endl;
  
```

Bei diesem Beispiel werden Sie in den Zeilen **02** und **03** zunächst aufgefordert, die Anzahl der Summanden einzugeben, die Sie für die folgende Addition in der Schleife verwenden wollen. Hiermit haben Sie dann quasi die Abbruchbedingung für die `while`-Schleife in Zeile **04** definiert. Solange der Wert von `i` kleiner als die Anzahl der erforderlichen Summanden (hier `n`) ist, wird der Schleifenrumpf aus den Zeilen **04** bis **09** ausgeführt. Im Schleifenrumpf werden Sie jetzt aufgefordert, den 1. Summanden einzugeben (Zeilen **05** bis **09**). Den eingegebenen Wert addieren wir zum Gesamtwert (Zeile **10**), wofür die Variable `summe` verwendet wird.

Von ganz besonderer Wichtigkeit ist Zeile **11**, wo wir die Variable `i` um 1 inkrementieren. Damit haben Sie quasi die Schleifenbedingung aktualisiert. Am Ende des Schleifenrumpfs geht es wieder zurück zum Anfang der Schleife in Zeile **04**, wo wir wieder die Schleifenbedingung (`i < n`) überprüfen. Ist diese wahr (`= true`), beginnt ein weite-

rer Schleifendurchlauf. Dies wird so lange wiederholt, bis die Schleifenbedingung in Zeile 04 falsch ist (= false) und die Schleife beendet wird. Anschließend fährt das Programm mit seiner Ausführung hinter dem Schleifenrumpf fort. In unserem Fall wird hierbei noch die Summe alle addierten Werte in Zeile 13 ausgegeben.

Endlosschleife

Die Gefahr bei den Schleifen ist, das Vergessen oder falsche Aktualisieren für das Abbruchkriterium der Schleifenbedingung. Würden Sie im Beispiel das Inkrementieren der Zeile 11 vergessen, würde das Programm in einer Endlosschleife ausgeführt, die ohne fremde Hilfe nicht mehr beendet werden könnte.

3.3.3 Fußgesteuerte Schleife – do while()

Die do-while-Schleife funktioniert im Prinzip wie die eben beschriebene while-Schleife, nur wird hier die Schleifenbedingung erst am Ende des Schleifenrumpfs überprüft. Das bedeutet auch, dass alle Anweisungen im Schleifenrumpf mindestens einmal ausgeführt werden. Hier der Pseudocode der fußgesteuerten Schleife:

Start

Anweisung(en)

Wert für Schleifenbedingung aktualisieren

Wenn Schleifenbedingung wahr ist, wieder mit „Start“ anfangen

In C++ lässt sich diese Form der Schleife wie folgt schreiben:

```
do{
    Anweisung(en);
}while( Schleifenbedingung );
Anweisung(en);
```

Eingeleitet wird der Schleifenrumpf für die fußgesteuerte Schleife mit dem Schlüsselwort `do`. Die folgenden `Anweisung(en)` zwischen `{}` werden jetzt auf jeden Fall mindestens einmal ausgeführt. Am Ende des Schleifenrumpfs wird die Schleifenbedingung mit `while` überprüft. Ist diese wahr (= true) findet ein erneuter Schleifendurchlauf ab dem Schlüsselwort `do` statt. Trifft die Schleifenbedingung nicht mehr zu (= false) wird die Schleife beendet und die Programmausführung fährt dahinter fort (Abbildung 3.5).

Beachten Sie hierbei auch, dass Sie am Ende der do-while-Schleife das Semikolon hinter `while` nicht vergessen.

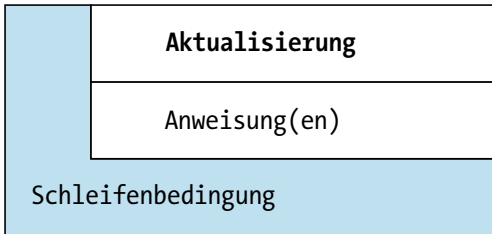


Abbildung 3.5 Struktogramm der do-while-Schleife

Hierzu folgt nun wieder ein einfaches Beispiel, welches die do-while-Schleife in der Praxis demonstrieren soll:

```

00 // kapitel03/dowhile.cpp
...
01 int calc={0}, val={0}, wahl={0};
02 std::cout << "Wert eingeben: ";
03 if( !(std::cin >> calc)) {
04     std::cout << "Fehler bei der Eingabe\n";
05     return 1;
06 }
07 do {
08     std::cout << "-1- Addieren\n";
09     std::cout << "-2- Subtrahieren\n";
10     std::cout << "-0- Ergebnis und Ende\n";
11     std::cout << "Ihre Wahl : ";
12     if( !(std::cin >> wahl) ) {
13         wahl=0; // Schleife beenden
14     }
15     if(wahl != 0) {
16         std::cout << "Wert eingeben: ";
17         if( !(std::cin >> val) ){
18             wahl=0; // Schleife beenden
19         }
20     }
21     switch(wahl) {
22         case 1 : calc+=val; break;
23         case 2 : calc-=val; break;
24         default: std::cout << "Ergebnis: " << calc << std::endl;
25     }
26 }while(wahl != 0);

```

Das Beispiel stellt einen einfachen Taschenrechner für einfache Additionen und Subtraktionen dar. In den Zeilen 02 bis 06 geben Sie den ersten Wert wie für Taschen-

rechner ein. In Zeile 07 startet unser `do-while`-Schleifenrumpf und wird mindestens einmal durchlaufen. Zunächst erstellen wir ein kleines Menü, in dem der Anwender auswählen kann, was er machen will (Zeilen 08 bis 14). Wenn die Auswahl nicht 0 war, geben Sie in den Zeilen 16 bis 19 den zweiten Wert ein, den Sie addieren bzw. subtrahieren wollen (abhängig von Ihrer Wahl).

Je nachdem, was Ihre Wahl gewesen ist, wird die folgende `switch`-Fallunterscheidung ihre Arbeit verrichten. Hierbei wird entweder addiert (`wahl=1`), subtrahiert (`wahl=2`) oder das Ergebnis ausgegeben (`wahl=0`). In Zeile 26 ist das Ende der `do-while`-Schleife, wo die Schleifenbedingung überprüft wird, ob der Wert von `wahl` nicht gleich 0 ist. Solange diese Schleifenbedingung wahr (`true`) ist, können bzw. wollen Sie ja weiterrechnen, und der Schleifenrumpf wird erneut ab Zeile 07 bei `do` ausgeführt.

Im Beispiel wurde außerdem bei einer falschen Eingabe (bspw. Buchstabe statt einer Zahl) damit reagiert, dass `wahl` dann den Wert 0 zugewiesen bekommt (Zeilen 13 und 18), womit das Ergebnis trotzdem ausgegeben und die Schleifenbedingung auch sauber beendet wird.

3.3.4 Zählschleife – `for()`

Die `for`-Schleife ist eine spezielle Form einer kopfgesteuerten Schleife. Diese Schleife ist recht ähnlich zur `while`-Schleife, nur dass hier zusätzlich zur Schleifenbedingung noch im Schleifenkopf eine Initialisierung und Aktualisierung einer Schleifenvariablen erfolgen **kann**.

Der Pseudocode würde in etwa dem der `while`-Schleife entsprechen, daher hierzu ein etwas anderer Pseudocode zur `for`-Schleife:

```
for Schleifenvariable mit Anfangszahl initialisieren
to Endzahl step Schrittweite do Schleifenrumpf
```

Diese Schleife ist also so gedacht, dass der Rechner von einer Anfangszahl bis zu einer Endzahl in einer bestimmten Schrittweite hochzählt und dabei jedes Mal den Schleifenrumpf mit den Anweisungen ausführt. Die Schleifenvariable wird auch als *Iterator* bezeichnet und sollte in der Praxis auch im Schleifenkopf geändert werden und nicht im Schleifenrumpf, was als schlechter Programmierstil gilt und eigentlich auch nicht mehr im Sinne der `for`-Schleife wäre. Dann könnte man auch gleich wieder die `while`-Schleife verwenden. Die `for`-Schleife ist also die klassische **Zählschleife**.

Folgendermaßen wird die `for`-Schleife in C++ formuliert:

```
for( Initialisierung; Bedingung; Aktualisierung ) {
    Anweisung(en);
}
Anweisung(en)
```

Zuallererst, und auch nur ein einziges Mal, wird die Initialisierung im Schleifenkopf durchgeführt. Anschließend wird die Schleifen(Bedingung) überprüft, ob diese wahr (= true) oder falsch (= false) ist. Wird true zurückgegeben, werden die Anweisung(en) im Schleifenrumpf ausgeführt. Wurden alle Anweisungen ausgeführt, erfolgt die Aktualisierung der Schleifenvariablen im Schleifenkopf. Daraufhin beginnt alles erneut mit dem Überprüfen der Schleifen(Bedingung). Die Schleife wird beendet, wenn die Schleifen(Bedingung) falsch ist (= false), wonach das Programm hinter dem Schleifenrumpf fortgeführt wird (Abbildung 3.6).

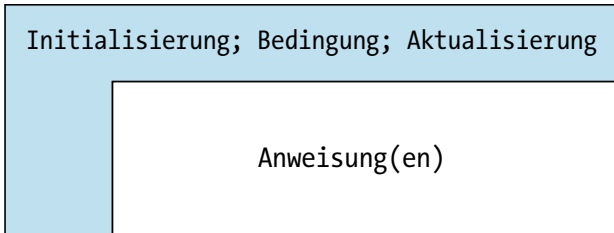


Abbildung 3.6 Struktogramm der for-Schleife

Hierzu ein einfaches Beispiel, welches diese Zählschleife demonstrieren soll:

```
00 // kapitel03/for.cpp
...
01 int ende={0};
02 std::cout
    << "Wie oft soll die Schleife durchlaufen werden:";
03 if( !(std::cin >> ende) ) {
04     std::cout << "Fehler bei der Eingabe\n";
05     ende=0;
06 }
07 for(int count=0; count < ende; ++count) {
08     std::cout << count+1 << ". Schleifendurchlauf\n";
09 }
```

In diesem Beispiel werden Sie gefragt, wie oft Sie die Schleife ausführen wollen (Zeilen 01 bis 06). In der for-Schleife aus Zeile 07 initialisieren Sie zunächst den Anfangswert von count (unsere Zählvariable) mit 0. Dann wird die Schleifenbedingung überprüft, ob die Zählvariable count kleiner ist als der von uns eingegebene Wert ende. Ist diese Bedingung wahr (= true), wird der erste Schleifendurchlauf im Schleifenrumpf (Zeilen 08 bis 09) ausgeführt. Hier ist dies nur eine einfache Ausgabe auf dem Bildschirm, welche Auskunft gibt, wie oft die Schleife bereits durchlaufen wurde. Nach dem Schleifendurchlauf wird die Zählvariable count um den Wert 1 inkrementiert (++count). Als Nächstes wird erneut die Schleifenbedingung überprüft, ob die Zählvariable count kleiner als der von uns eingegebene Wert ende ist. Dieser

Vorgang wird jetzt so lange wiederholt, bis die Schleifenbedingung falsch ist (= false). Nach dem Ende der `for`-Schleife wird mit der Ausführung hinter dem Schleifenrumpf fortgefahren.

Flexible `for`-Schleife

Hier muss noch erwähnt werden, dass die `for`-Schleife wesentlich flexibler ist, als es hier dargestellt wurde. Im Prinzip ist es auch möglich, dass Sie bei den drei durch ein Semikolon getrennten Anweisungen reinschreiben können, was Sie wollen, solange die Ausdrücke syntaktisch korrekt sind. Es müssen nicht mal alle drei Ausdrücke in der `for`-Schleife besetzt werden. Mindestens vorhanden sein müssen beide Semikolons wie `for(;;)` was allerdings dann eine Endlosschleife wäre.

3.3.5 Mengenschleife – Range-based `for` (C++11)

In C++11 wurde die Funktion der `for`-Schleife um eine Range-basierte `for`-Schleife erweitert. Zwar konnte man bereits mithilfe der (externen) *Boost*-Bibliothek eine solche Schleife implementieren, aber jetzt ist diese Form auch endlich im Standard enthalten. In anderen Programmiersprachen (und auch in der *Boost*-Bibliothek) ist diese Schleife unter dem Namen *foreach* bekannt.

Diese Schleife erlaubt es Ihnen, ganz komfortabel durch die einzelnen Elemente eines Containers zu iterieren. Aber auch herkömmliche rohe Arrays (auch als C-style-Arrays bekannt) lassen sich damit sehr komfortabel durchlaufen. Genauer steht Ihnen die neue Syntax für folgende C++-Elemente zur Verfügung:

- ▶ rohe Arrays (C-style-Arrays) mit bekannter Größe
- ▶ Standardcontainer (früher auch STL genannt), welche die Elementfunktionen `begin()` und `end()` enthalten. Auch auf die Klasse `std::string` kann die neue Range-`for` verwendet werden.
- ▶ Initialisierungslisten mit der neuen vereinheitlichten Klammersyntax
- ▶ bei eigenen Klassen, welche die Elementfunktionen `begin()` und `end()` enthalten bzw. implementiert werden müssen
- ▶ bei einer Überladung von `begin()` und `end()`

Das folgende Listing zeigt Ihnen, wie Sie ohne großen Aufwand durch die einzelnen Elemente eines `vector`-Containers (`vector`, siehe Abschnitt 4.1.1, »Standardcontainer `std::vector`«) durchlaufen können:

```
00 // kapitel03/for_cpp11.cpp
01 #include <vector>
...
```



```

02 std::vector<int> ivec{ 11, 22, 33, 44, 55 };
03 for( auto ival : ivec ) {
04     std::cout << ival << std::endl;
05 }

```

Hier sehen Sie auch gleich den Vorteil des Schlüsselwortes `auto`, mit dem sich der Compiler um den benötigten Typ für `ival` kümmert.

Mehr zur Range-basierten for-Schleife

Die neue Range-basierte `for`-Schleife hier gleich komplett breitzutreten, würde Ihnen zu viele Vorkenntnisse abverlangen, daher finden Sie bei den passenden Kapiteln immer wieder kurze Abschnitte und Beispiele, wie Sie etwas mit der neuen Range-basierten `for`-Schleife anstelle von der alten `for`-Schleife einfacher machen können. Denn das ist es, was dieses neue Range-`for` letztlich tut: einem das Programmierleben einfacher zu machen!

3.3.6 Endlosschleife

In der Informatik ist eine Endlosschleife eine Schleife, welche immer wieder erneut ausgeführt wird und nur dadurch beendet werden kann, wenn die Ausführung von außen abgebrochen wird. Solche Endlosschleifen können auf folgende zwei Arten entstehen:

- Fehler bei der Programmierung: Wenn der Programmierer eine falsche Abbruchbedingung in einer Schleife definiert, welche nie eintritt, bspw.:

```

01 int i={1};
02 while( i < 100 ) {
03     std::cout << "Endlosschleife\n";
04 }

```

Die Abbruchbedingung in Zeile 02 wird hier niemals erreicht, weil die Schleifenvariable `i` im Schleifenrumpf nicht mehr aktualisiert wird (bspw. mit `++i`). Hier wird endlos der Text »Endlosschleife« auf dem Bildschirm ausgegeben.

- Eine gewollte Endlosschleife: Es gibt durchaus Anwendungen, die in einer Endlosschleife ausgeführt werden und auf etwas warten. Da wären Dinge, wie bspw. ein Webserver, der auf eine Anfrage wartet, oder eine Maus, die auf eine Bewegung wartet. Auf Ihrem System laufen viele solcher Endlosschleifen (auch als Ereignisschleifen bezeichnet) gleichzeitig ab. Ein einfacher Codeausschnitt dazu:

```

01 while( true ) {
02     std::cout << "Endlosschleife\n";
03 }

```

Diese Schleife wird so lange ausgeführt, solange die Schleifenbedingung `true` ist, und das ist, in diesem Fall, für immer der Fall. Zwar gibt es durchaus die Möglichkeit, solche Endlosschleifen mit einer `break`-Anweisung zu beenden, aber in solch einem Fall ist es keine echte Endlosschleife mehr.

3.3.7 Kontrollierte Sprunganweisungen

Mit den Schlüsselwörtern `break` und `continue` können Sie kontrollierte Sprünge innerhalb einer Schleife durchführen.

return und exit

Auch mit `return` können Sie eine Schleife beenden, aber dieser Befehl ist schon etwas weitgreifender und beendet zusätzlich auch gleich eine ganze Funktion. Im Falle der `main()`-Funktion wäre dies auch gleichzeitig das Ende des Programms. Mit `exit()` hingegen wird das Programm sofort und unverzüglich beendet – beides also eher nicht geeignet, um Schleifen zu beenden.

Die break-Anweisung

Die `break`-Anweisung kennen Sie bereits von der `switch`-Fallunterscheidung. In einer Schleife machen Sie mit einer `break`-Anweisung einen kontrollierten Sprung aus dem umfassenden Anweisungsblock – genauer beenden Sie den Anweisungsblock, worin sich `break` befindet. Der `break`-Ausdruck kann allerdings nicht willkürlich irgendwo hingesetzt werden, sondern kann nur innerhalb von Schleifen und `switch` verwendet werden.

Schleifenende: break vs. Bedingung

Auch wenn `break` eine Schleife manuell beenden kann, sollten Sie dies nicht missbrauchen, um Schleifen nur noch damit zu beenden. Ein sauberer Programmierstil ist es immer noch, eine Schleife ordentlich mit einer Schleifenbedingung zu beenden. Sofern Sie mehrere Schleifen verschachtelt haben, bezieht sich die `break`-Anweisung nur auf die aktuelle Schleife.

Hierzu ein einfaches Beispiel mit einer `break`-Anweisung:

```
00 // kapitel03/break.cpp
...
01 int ival={0}, sum={0};
02 while(ival != -1) {
03     std::cout << "Bitte Ganzzahl eingeben (-1=Ende): ";
04     if( std::cin >> ival ) {
```

```

05     sum+=ival;
06 }
07 else if( ival != -1 ) {
08     std::cout << "Fehler bei der Eingabe\n";
09     break;
10 }
11 }
12 std::cout << "Gesamtsumme: " << sum << std::endl;

```

In diesem Beispiel wurde die `break`-Anweisung in Zeile **09** eingebaut, für den Fall, dass der Anwender eine falsche Eingabe gemacht hat. Und zwar wird diese `break`-Anweisung ausgeführt, wenn die Eingabe in Zeile **04** nicht richtig (= `false`) und der eingegebene Wert nicht `-1` (Zeile **07**) gewesen ist, was hier auch die Abbruchbedingung für die Schleife darstellt. In diesem Beispiel ist die `break`-Anweisung recht nützlich, weil bei einer falschen Eingabe eines Buchstabens z. B. das Programm gegebenenfalls in einer Endlosschleife laufen könnte. Probieren Sie es am besten selbst aus. Hierzu nochmals dasselbe Beispiel ohne die Überprüfungen und `break`:

```

...
int ival={0}, sum={0};
while(ival != -1) {
    std::cout << "Bitte Ganzzahl eingeben (-1=Ende): ";
    std::cin >> ival;
    sum+=ival;
}
std::cout << "Gesamtsumme: " << sum << std::endl;
...

```

Die `continue`-Anweisung

Die `continue`-Anweisung beendet, im Gegensatz zur `break`-Anweisung, nur den aktuellen Schleifendurchlauf und springt wieder zur Schleifenbedingung zurück, um gleich gegebenenfalls den nächsten Schleifendurchlauf auszuführen, wenn die Schleifenbedingung immer noch `true` ist. Hierzu ein Minimal-Beispiel:

```

00 // kapitel03/continue.cpp
...
01 for(int it=0; it < 20; ++it) {
02     if( !(it % 2) || !(it % 5) ) {
03         continue;
04     }
05     std::cout << it << std::endl;
06 }

```

In diesem Beispiel werden in einer `for`-Schleife die Werte 1 bis 19 hochgezählt. In Zeile **02** wird überprüft, ob der Wert `it` durch 2 oder nicht durch 5 teilbar ist (also keinen Rest bei einer Division zurückgibt). Ist der Wert von `it` durch 2 oder 5 teilbar, sind wir an einer Weiterarbeit mit diesem Wert nicht interessiert und springen in Zeile **03** mit `continue` wieder hoch zum nächsten Schleifendurchlauf. Wird jedoch ein Rest bei einer Division durch 2 oder 5 zurückgegeben, dann wird der Wert in Zeile **05** ausgegeben.

Das Beispiel lässt sich allerdings auch ohne `continue` realisieren:

```
01 for(int it=0; it < 20; ++it) {
02     if( (it % 2) && (it % 5) ) {
03         std::cout << it << std::endl;
04     }
05 }
```

An dieser Stelle noch folgender Warnhinweis: Bei der Verwendung von `continue` werden gerne mal Endlosschleifen produziert. Folgendes Beispiel soll diese zeigen:

```
01 int it={0};
02 while( it < 20 ) {
03     if( !(it % 2) || !(it % 5) ) {
04         continue;
05     }
06     std::cout << it << std::endl;
07     ++it;
08 }
```

Hier wurde die `for`-Schleife durch eine `while`-Schleife ausgetauscht. Das Problem liegt jetzt in den Zeilen **03** und **04**. Sobald die `if`-Bedingung in der Zeile zutrifft und `true` zurückgibt, wird wieder zum Schleifenanfang mit `continue` gesprungen. Der Wert von `it` wird jetzt allerdings nie mehr aktualisiert, weil Zeile **07** niemals mehr erreicht wird, da die Bedingung in Zeile **03** immer `true` ist und somit immer wieder mit `continue` zum Anfang der Schleife gesprungen wird. Das Problem lässt sich einfach beheben, indem Sie zwischen den Zeilen **03** und **04** den Wert von `it` mit `++it` erhöhen.

Kapitel 4

Jenseits der Basisdatentypen

In diesem Kapitel lernen Sie etwas höhere Typen kennen. Diese Typen werden wiederum aus gewöhnlichen Basisdatentypen gebildet. Hierzu zählen u.a. Arrays, Zeiger, Referenzen oder Strukturen. Zwar gehören hier auch Klassen dazu, aber diese werden gesondert in Kapitel 7 des Buches behandelt, wo es dann auch um die objektorientierte Programmierung geht.

In diesem Kapitel finden Sie auch das eine oder andere etwas komplexere Thema. Grundlegend werden hier zwar die Arrays, Strings, Strukturen, Unions, Aufzählungstypen, Zeiger und Referenzen behandelt, aber oft finden sich hier auch passende Themen, die etwas über den Einstiegsbereich hinausgehen und gerade für Anfänger etwas kompliziert erscheinen. So habe ich bspw. bei den Arrays auch gleich das assoziative Array kurz beschrieben oder bei den Strings auch gleich ein wenig die neue Unicode-Unterstützung von C++11 und deren Konvertierungsmöglichkeiten mit reingepackt. Dies sind zunächst mal alles Themen, womit man am Anfang mit C++ nicht viel zu tun haben wird. Aber ich denke, es ist sinnvoll, darüber gleich von Anfang an Bescheid zu wissen.

4.1 Arrays

Als Array (dt. Feld) wird eine Datenstruktur bezeichnet, in der Sie Daten von einem (üblicherweise) einheitlichen Datentyp speichern können. Der Zugriff auf die Daten erfolgt über einen Index – was bei einem gewöhnlichen Array ein ganzzahliger Wert ist. Ein Array mit N -Elementen beginnt in C++ immer mit dem Index 0, und das letzte Element hat den Index $N-1$. Bei einem Array können Sie sich darauf verlassen, dass diese in einem Feld gespeicherten Elemente auch hintereinander im Speicher abgelegt sind, also direkt aufeinanderfolgend.

In C++ haben Sie zwei (mit C++11 drei) Möglichkeiten, Arrays als Datenstruktur zu verwenden. Die bessere, komfortablere und wesentlich sicherere Möglichkeit führt über den Standardcontainer `std::vector`. Die zweite Möglichkeit sind die rohen Arrays (auch C-Style-Arrays genannt), deren Anwendung allerdings wesentlich umständlicher und auch fehleranfälliger ist. Daher haben die C++-Entwickler in C++11 mit einer

neuen Standardcontainer-Klasse `std::array` eine würdige und vor allem sicherere Alternative für das rohe Array zur Verfügung gestellt.

Natürlich sollte man hier nicht die Zeichenketten (engl. Strings) vergessen zu erwähnen, weil diese letztendlich auch wieder nur Arrays sind – am nächsten sind sie einem Vektor vom Typ `char`.

Neben einem gewöhnlichen Array gibt es in C++ auch noch ein *assoziatives Array*. Ein assoziatives Array verwendet keinen numerischen Index wie ein gewöhnliches Array, sondern einen sogenannten Schlüssel zur Indizierung und somit Adressierung der einzelnen Elemente.

4.1.1 Standardcontainer `std::vector`

Die gängigste und C++-üblichste Form, um Arrays zu verwenden, ist mit der Standardcontainer-Klasse `vector` implementiert. Um diese Containerklasse verwenden zu können, müssen Sie den gleichnamigen Header `<vector>` mit einbinden. Alles, was mit der Standardcontainer-Klasse `vector` verwendet werden kann, ist im Namensraum `std` enthalten.

Das Datenmodell von `std::vector` ist als dynamisches Array implementiert (Abbildung 4.1). Das bedeutet hier, dass Sie jederzeit zur Laufzeit weitere Elemente hinzufügen können. Der Standard schreibt allerdings nicht vor, wie dieses dynamische Array implementiert ist.

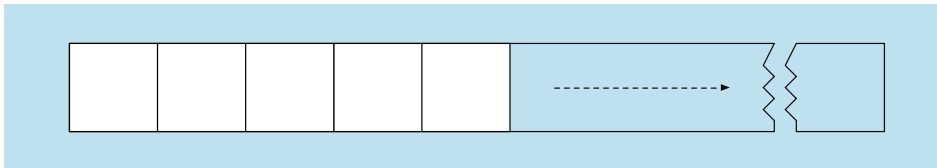


Abbildung 4.1 Die Datenstruktur eines `std::vector`

Array anlegen

Um ein neues Array des Standardcontainers `std::vector` anzulegen, wird folgende Syntax verwendet:

```
std::vector<Typ> Name (N);
```

Den Typ, von dem Sie ein Array erstellen wollen, müssen Sie zwischen den spitzen Klammern angeben. Mit Name geben Sie den Bezeichner für das Array an, und zwischen die runden Klammern schreiben Sie die Anzahl der Elemente für N, welche das Array vom Datentyp Typ speichern kann. Ein Array mit 100 Elementen vom Datentyp `int` legen Sie somit wie folgt an:

```
std::vector<int> iArray(100); // 100 int-Werte
```

Sehr komfortabel ist hierbei auch gleich, dass alle Elemente automatisch mit 0 initialisiert werden.

Wie schon erwähnt, sind die Arrays mit `std::vector` dynamische Arrays, und Sie müssen daher nicht zwangsläufig die Anzahl der Elemente angeben, also wie viele Arrays gespeichert werden können. Sie können daher auch hergehen und Folgendes verwenden:

```
std::vector<int> iArray; // Leer!!!
```

Komplettes Array direkt initialisieren (C++11)

Wollen Sie Elemente direkt beim Anlegen eines Arrays initialisieren, dann ist dies erst seit dem neuen C++11-Standard und der neuen vereinheitlichten Initialisierungssyntax möglich:

```
01 std::vector<int> iArray{ 1, 2, 3, 5, 8, 13 }; // C++11
02 std::vector<double> fArray={ 1.0, 1.5, 2.5, 4.0 }; // C++11
```

In Zeile **01** sehen Sie ein `int`-Array mit sechs `int`-Elementen, welche alle sofort mit einem Wert initialisiert wurden. Ähnlich ist es in Zeile **02**, nur ist es hier ein `double`-Array mit vier sofort initialisierten Elementen.

Zugriff auf die Elemente

Für den lesenden wie auch schreibenden Zugriff auf die einzelnen Elemente des Arrays über den Index steht Ihnen der klassische Indizierungsoperator `[]` oder die Methode `at()` (eine Funktion der Klasse `std::vector`) zur Verfügung. Hierzu beide Versionen im Einsatz:

```
00 // kapitel04/vector01.cpp
01 #include <vector>
...
02 std::vector<int> iArray(10);

03 iArray[0] = 12;
04 std::cout << "Bitte Wert eingeben: ";
05 std::cin >> iArray[1];
06 std::cout << "Ihre Eingabe: " << iArray[1] << std::endl;

07 iArray.at(2) = 34;
08 std::cout << "Bitte Wert eingeben: ";
09 std::cin >> iArray.at(3);
10 std::cout << "Ihre Eingabe: " << iArray.at(3)
    << std::endl;
```


In Zeile **02** wird ein `int`-Array mit zehn Elementen mithilfe des Standardcontainers `std::vector` angelegt. In Zeile **03** wird dem ersten Element mithilfe des Indizierungsoperators `[]` der Wert 12 zugewiesen. Achten Sie darauf, dass das erste Element den Index 0 hat! Genauso einfach können Sie auf den Index mithilfe des Indizierungsoperators über die Standard-Streams `std::cin`, `std::cout` oder `std::cerr` auf die einzelnen Array-Elemente zugreifen, wie in den Zeilen **05** und **06** gezeigt wird, wo das zweite Element eingelesen und ausgegeben wird.

Index 0 bis N-1

Beachten Sie bitte, dass das erste Element in einem Array immer den Index 0 besitzt und das letzte Element den Index N-1 (N = Anzahl der Elemente).

Dasselbe, das eben mit dem Indizierungsoperator gemacht wurde, erreichen Sie jetzt auch mit der Methode `at()`. In Zeile **07** wird dem dritten Element im Array der Wert 34 zugewiesen. Auch das Einlesen über `std::cin` (Zeile **09**) und die Ausgabe mit `std::cout` (Zeile **10**) funktionieren problemlos mit der Methode `at()` und dem entsprechenden Index (hier dem vierten Element).

Der Unterschied zwischen dem Indizierungsoperator `[]` und der Methode `at()` liegt darin, dass bei `at()` auch Grenzbereichüberschreitungen des Arrays abgefangen werden. Dies ist möglich, indem beim Überschreiten des gültigen Wertebereichs die Ausnahme `out_of_range` geworfen wird. Der Operator `[]` hingegen überprüft nicht die Grenzen eines Arrays. Ein einfacher Codeausschnitt soll zeigen, was hiermit gemeint ist:

```
01 std::vector<int> iArray(10);
02 iArray[11] = 12;      // Bereichsüberschreitung !!!
03 iArray.at(12) = 34;   // Bereichsüberschreitung !!!
```

In Zeile **02** findet eine Bereichsüberschreitung statt. Hier wird dem elften Element der Wert 12 zugewiesen, obwohl das Array nur Platz für zehn Elemente enthält. Von nun an ist das weitere Verhalten des Programms undefiniert. In Zeile **03** wird ebenfalls in einen nicht erlaubten Bereich geschrieben, aber dank der Elementfunktion `at()` wird hier die Ausnahme `out_of_range` geworfen. Ausnahmen werden in Kapitel 10, »Ausnahmebehandlung (Exceptions)«, umfassender beschrieben. Trotzdem hier ein kurzer Codeausschnitt, wie Sie in Ihrem Programm auf eine solche Ausnahme reagieren können:

```
01 #include <vector>
02 #include <stdexcept> // benötigter Header
...
03 std::vector<int> iArray(10);
```

```

04 try { // Codeausschnitt, der eine Ausnahme werfen könnte
05     iArray.at(12) = 34;
06 }
07 catch( std::out_of_range& range ) { // out_of_range abf.
08     std::cerr << "Bereichsüberschreitung: "
                << range.what() << std::endl;
09 }

```

Indizierungsoperator [] oder at() verwenden?

Eine allgemeine Empfehlung kann hier nicht abgegeben werden. Der Vorteil des Indizierungsoperators ist, dass hierbei keine extra Überprüfung stattfinden muss und er somit auch etwas effizienter ist. Wenn es allerdings nicht zu sehr auf die Geschwindigkeit ankommt, dann ist die Methode `at()` wesentlich sicherer, weil hiermit der gefährliche Pufferüberlauf abgefangen werden kann. Wenn Sie Einsteiger in der C++-Programmierung sind, würde ich Ihnen zunächst immer zu `at()` raten. Ansonsten müssen Sie eben zwischen Sicherheit und Performance abwägen.

Zugriffszeit von `std::vector`

Der Standardcontainer `std::vector` hat eine sehr gute Performance, wenn Sie neue Elemente am Ende hinzufügen oder entfernen wollen. Weniger geeignet ist dieser Container, wenn Sie Elemente dazwischen einfügen oder entfernen wollen, weil alle anderen Elemente dahinter um eine Position nach vorne bzw. hinten geschoben werden müssen. Es muss quasi für jedes Element dahinter der Zuweisungsoperator verwendet werden.

Einem leeren Array von `std::vector` können Sie nicht einfach mit dem Indizierungsoperator oder `at()` einen Wert zuweisen. Dabei wird häufig auf die Methode `push_back(T)` zurückgegriffen, womit das neue Element vom Typ `T` hinten am Array hinzugefügt wird. Um die Speicherverwaltung müssen Sie sich hierbei nicht kümmern:

```

01 std::vector<int> iArray; // Leer !!!
02 iArray[0] = 100;         // Undefiniertes Verhalten!!!
03 iArray.at(1) = 200;      // Wirft out_of_range
04 std::vector<int> iArray2; // Leer !!!
05 iArray2.push_back(100);   // iArray2[0]=100
06 iArray2.push_back(200);   // iArray2[1]=200

```

In Zeile **01** legen wir ein leeres Array für `int`-Werte an. Da hier die Anzahl der Elemente für das Array nicht angegeben wurde, sind die Zuweisungen der Zeilen **02** und **03** fehlerhaft. In Zeile **04** legen wir erneut ein leeres Array für `int`-Werte an. Die Zeilen

O5 und **O6** mit `push_back()` sind korrekt. Hiermit wird das Array dynamisch um ein Element erweitert, und der neue Wert jeweils am Ende angefügt.

pop_back()

Das Gegenstück zu `push_back()` finden Sie mit `pop_back()`, womit das letzte Element vom Container entfernt wird, wenn dieser noch nicht leer ist. Ob ein Container leer ist, können Sie mit der Methode `empty()` überprüfen.

Alle Elemente durchlaufen

Um alle Elemente zu durchlaufen, müssen Sie wissen, wie viele Elemente enthalten sind. Diese Informationen können Sie sich mit der Methode `size()` ermitteln lassen. Mithilfe von `size()` können Sie sich auch vor falscher Verwendung des Indizierungsoperators `[]` schützen:

```
01 std::vector<int> iArray;
...
02 if( iArray.size() > 10 ) {
03     iArray[10] = 100;
04 }
```

Hier würde nur Zeile **O3** ausgeführt, wenn die Anzahl der Elemente von `iArray` größer als 10 ist, wie dies in der `if`-Bedingung der Zeile **O2** mithilfe der Elementfunktion `size()` überprüft wird.

Da Sie jetzt wissen, wie Sie Anzahl der Elemente eines Arrays ermitteln können, sollte es kein Problem mehr sein, mithilfe einer `for`-Schleife alle Elemente zu durchlaufen:

```
00 // kapitel04/vector02.cpp
01 #include <vector>
...
02 std::vector<int> iArray(10);
03 for(int it=0; i < iArray.size(); ++i ) {
04     iArray.at(i) = i+i+1;
05 }
06 for(int i=0; i < iArray.size(); ++i ) {
07     std::cout << iArray.at(i) << std::endl;
08 }
```

In der `for`-Schleife der Zeilen **O3** bis **O5** durchlaufen wir alle zehn Elemente von `iArray` und weisen mithilfe von `at()` einen Wert zu (hier sind es lauter ungerade Zahlen von 1 bis 19). Dank `size()` kennen wir die Anzahl der Elemente, die wir hier hinzufügen können. Dasselbe machen wir nochmals in den Zeilen **O6** bis **O8**, nur geben wir hier die zuvor zugewiesenen Werte auf dem Bildschirm aus.

Range-basierte for-Schleife (C++11)

Mit Einführung der neuen Range-basierten for-Schleife (siehe Abschnitt 3.3.5, »Mengenschleife – Range-based for (C++11)«) in C++11 lässt sich das Durchlaufen der einzelnen Elemente wesentlich einfacher realisieren. Hierzu ein Beispiel mit der neuen Range-basierten for-Schleife:

```
00 // kapitel04/vector03.cpp
01 #include <vector>
...
02 std::vector<int> iArray(10);
03 int ival=1;
04 for(auto &elem : iArray) {
05     elem = ival++;
06 }
07 for(const auto &elem : iArray) {
08     std::cout << elem << std::endl;
09 }
```

Das Beispiel mit der neuen Range-basierten for-Schleife durchläuft hierbei alle Elemente von `iArray.begin()` (dem ersten Element) bis `iArray.end()` (hinter dem letzten Element). Dies ist mit allen Standardcontainern ohne Weiteres möglich, wenn diese die Methoden `begin()` und `end()` implementiert haben.

Referenz (&) bei »auto« nicht vergessen

An dieser Stelle muss ich kurz auf Referenzen vorgehen, welche erst in Abschnitt 4.4, »Referenzen«, behandelt werden. Und zwar geht es um das &-Zeichen bei `auto &it`, welches hier sehr wichtig ist. Vergessen Sie hierbei nämlich die Referenz (also das &), dann würden Sie quasi mit einer Kopie des Elements arbeiten. Ein schreibender Zugriff auf das Element hätte somit keinen Effekt, und der Compiler würde sich hoffentlich mit einer nicht verwendeten Variablen melden. Mit dem Referenzzeichen wird eine Referenz verwendet und keine Kopie gemacht. Für Einsteiger gilt hier zunächst einfach nur: Vergessen Sie & bei `auto` in solchen Schleifen nicht!

4.1.2 Rohe Arrays (C-Style-Array)

Neben dem Standardcontainer `std::vector` haben Sie noch die Möglichkeit, rohe Arrays (auch C-Style-Arrays genannt) zu verwenden (Abbildung 4.2). Allerdings kommen diese Arrays recht spärlich daher und lassen sich (für C++-Verhältnisse) auch recht sperrig verwenden. So sind bspw. keine speziellen Funktionen für rohe Arrays vorhanden. Auch an eine dynamische Erweiterung ist bei den rohen Arrays zunächst nicht zu denken. Zwar lässt sich rein theoretisch ein dynamisches Array mithilfe von

Zeigern programmieren, aber dies ist doch sehr aufwendig und auch nicht nötig, weil es in C++ hierfür ja bereits `std::vector` gibt. Und wenn Sie unbedingt ein Array mit einer fixen Größe benötigen, steht Ihnen seit C++11 auch mit `std::array` eine Alternative zur Verfügung.

Des Weiteren sind Sie bei den rohen Arrays schutzlos den Überläufen von Arrays ausgeliefert. Zwar ist es prinzipiell möglich, ohne Fehler zu programmieren, aber die Vergangenheit hat leider gezeigt, dass Pufferüberläufe (engl. *Buffer Overflows*) ein ziemlich gängiges Problem sind.

Wozu also noch rohe Arrays verwenden? Da ist heutzutage eigentlich nur noch alter Code, in dem eben diese rohen Arrays verwendet werden. Manche werden hier vielleicht noch Geschwindigkeitsgründe nennen, aber das stimmt mittlerweile auch nicht mehr so ganz. Ein optimierter Vektor vom Typ `std::vector` sollte mittlerweile **mindestens** genauso schnell auf ein Array über Iteratoren zugreifen können wie auf ein rohes C-Style-Array.



Abbildung 4.2 Die Datenstruktur eines rohen Arrays

Arrays anlegen

Das Anlegen und der Zugriff auf rohe C-Style-Arrays geschieht allein über den Indexoperator `[]`. Die Syntax, um ein solches Array anzulegen, sieht wie folgt aus:

```
Typ Name[N];
```

Auch hier müssen Sie mit `Typ` den Datentyp angeben, von dem Sie zwischen den eckigen Klammern die Anzahl der Elemente festlegen – was hier auch eine Ordinalzahl (Ganzzahl) sein muss. Mit `Name` geben Sie den Bezeichner für das Array an. Ein rohes Array mit 100 Elementen vom Datentyp `int` legen Sie hierbei wie folgt an:

```
int rawArray[100];
```

Beachten Sie, dass die einzelnen Elemente bei lokalen C-Style-Arrays im Gegensatz zur Containerklasse `std::vector` nicht mit 0 initialisiert werden.

Initialisieren mit einer Initialisierungsliste

Im Gegensatz zum Standardcontainer `std::vector` konnten Sie ein C-Style-Array bereits vor dem C++11-Standard mit einer Initialisierungsliste explizit initialisieren, was im C++11-Standard als vereinheitlichte Initialisierung bekannt ist. Folgende Mög-

lichkeiten stehen also zur Verfügung, um ein Array gleich beim Anlegen zu initialisieren:

```
01 int rawArray01[3] = { 111, 222, 333 };
02 double rawArray02[] = { 1.5, 2.5, 5.0, 7.5 };
03 int rawArray03[100] = { 100 };
04 double rawArray04[] {100, 200, 300, 400}; // C++11
05 int rawArray[10]{ 1, 2, 3, 5, 8, 13, 21 };// C++11
```

In Zeile **01** wurden alle drei Elemente des Arrays gleich mit einer Liste von Werten initialisiert. Wenn Sie eine Initialisierungsliste verwenden, können Sie auch auf die Längenangabe für das Array verzichten, wie Sie es in Zeile **02** sehen können, wo ein Array mit vier Elementen gefüllt wird. In Zeile **03** wird das erste Element mit 100 initialisiert. Der Vorteil an dieser Art der zu kurzen Initialisierungsliste ist, dass die restlichen Elemente automatisch mit 0 initialisiert werden. Somit können Sie davon ausgehen, wenn Sie mindestens ein Element in einem großen Array in einer Initialisierungsliste initialisieren, werden alle anderen Elemente ebenfalls mit 0 initialisiert. Die Verwendungen der Initialisierungsliste ohne den Zuweisungsoperator, wie in den Zeilen **04** und **05** demonstriert, funktioniert nur im neuen C++11-Standard.

Zugriff auf die Elemente

Die einzelnen Elemente eines C-Style-Arrays werden mit dem Namen des Arrays und dem Indexwert zwischen den eckigen Klammern erreicht. Auch hier wird mit dem ersten Element bei 0 angefangen zu zählen. Das letzte Element wiederum hat auch hier den Indexwert $N-1$ (N = Anzahl der Elemente). Hierzu ein Beispiel, wie Sie auf die einzelnen Elemente eines rohen Arrays zugreifen können:

```
00 // kapitel04/rawArray01.cpp
...
01 int rawArray[10] = { 0 };
02 rawArray[0] = 12;
03 std::cout << "Bitte Wert eingeben: ";
04 std::cin >> rawArray[1];
05 std::cout << "Ihre Eingabe: " << rawArray[1]
    << std::endl;
```

In Zeile **01** legen Sie ein Array mit zehn Elementen an, welches hier dank der Initialisierungsliste automatisch auch gleich mit 0 initialisiert wurde. In Zeile **02** wird mithilfe des Indizierungsoperators `[]` der Wert 12 dem ersten Element mit dem Indexwert 0 zugewiesen. Ebenso können Sie mithilfe des Indizierungsoperators über die Standard-Streams `std::cin`, `std::cout` und `std::cerr` auf die einzelnen Elemente zugreifen, wie hier in den Zeilen **04** und **05** mit dem Einlesen und der Ausgabe des zweiten Elements im Array demonstriert wird.

Kein Schutz vor Pufferüberläufen

Beachten Sie, dass Sie bei C-Style-Arrays keine Komfortfunktionen wie `at()` haben, womit Sie sich vor Pufferüberläufen schützen können. Bei rohen Arrays sind Sie selbst dafür verantwortlich, dass Folgendes nicht passiert:

```
01 int rawArray[10] = { 0 };
02 rawArray[11] = 12; // Fehler: Unerlaubter Speicherzugriff
```

Alle Elemente durchlaufen

Auch hier gilt, um alle Elemente in einer Schleife durchlaufen zu können, müssen Sie zunächst wissen, wie viele Elemente in diesem Array enthalten sind. Ein reines `sizeof(T)` liefert Ihnen auch hier nur die Gesamtgröße in Bytes von einem Array zurück. Daher müssen Sie die Gesamtgröße noch durch die Größe des Typs dividieren, damit Sie die Anzahl der Elemente des Arrays erhalten – also: `sizeof(Array-name)/sizeof(Typ)`. Mithilfe dieser kleinen Berechnung können Sie sich vor Pufferüberläufen bei der Verwendung des Indizierungsoperators schützen:

```
01 int rawArray[10] = { 0 };
02 if(sizeof(rawArray)/sizeof(int) > 10) {
03     rawArray[10] = 12;
04 }
```

Durch die Überprüfung in Zeile **02** wird die Zuweisung in Zeile **03** nur ausgeführt, wenn die Anzahl der Elemente größer ist als 10.

Alle Elemente eines C-Style-Arrays können Sie jetzt wie folgt in einer `for`-Schleife durchlaufen:

```
00 // kapitel04/rawArray02.cpp
...
01 int rawArray[10] = { 0 };
02 for( int i=0; i < sizeof(rawArray)/sizeof(int); ++i ) {
03     rawArray[i] = i+1;
04 }
05 for( int i=0; i < sizeof(rawArray)/sizeof(int); ++i ) {
06     std::cout << rawArray[i] << std::endl;
07 }
```

In der ersten `for`-Schleife in den Zeilen **02** bis **04** werden alle zehn Elemente dank `sizeof(rawArray)/sizeof(int)` durchlaufen und mit den Werten 1 bis 10 versehen. Dasselbe wird nochmals in den Zeilen **05** bis **07** gemacht, nur geben wir hier die zuvor zugewiesenen Werte auf dem Bildschirm aus.

Meistens ist allerdings die Verwendung der `sizeof/sizeof`-Methode übertrieben, und man macht sich das Leben hiermit eigentlich nur unnötig kompliziert. In der Praxis kann man daher auch beruhigt hergehen und einfach Konstanten oder gar Literale verwenden. Im konkreten Beispiel in Listing *rawArray02.cpp* kann daher auch einfach direkt 10 oder eine Konstante hingeschrieben werden:

```
...
01 const unsigned int MAX = 10; // Konstante
02 int rawArray[MAX] = { 0 };
03 for( int i=0; i < MAX; ++i ) {
04     rawArray[i] = i+1;
05 }
06 for( int i=0; i < MAX; ++i ) {
07     std::cout << rawArray[i] << std::endl;
08 }
...
```

Range-basierte for-Schleife (C++11)

Wie auch schon zuvor beim Standardcontainer `std::vector` demonstriert wurde, können Sie sich auch bei einem C-Style-Array das Leben einfacher machen, wenn Sie das komplette Array durchlaufen wollen, indem Sie die in C++11 eingeführte Range-basierte `for`-Schleife verwenden. Das folgende Beispiel ist Listing *rawArray02.cpp* gleichwertig:

```
00 // kapitel04/rawArray03.cpp
...
01 int rawArray[10] = { 0 };
02 int ival=1;
03 for(auto &it : rawArray) {
04     it = ival++;
05 }
06 for(const auto &it : rawArray) {
07     std::cout << it << std::endl;
08 }
```

4.1.3 Standardcontainer `std::array` (C++11)

Es wurde bereits kurz erwähnt, dass in C++11 mit `std::array` ein neuer Standardcontainer eingeführt wurde. Wenn Sie ein Array mit fester Anzahl an Elementen benötigen, bietet Ihnen `std::array` u. U. die bessere Performance an, weil der Speicher vom Stack (wenn möglich) reserviert wird. Natürlich sollte hier klar sein, dass die Größe von `std::array` nicht mehr dynamisch erweitert werden kann wie bei `std::vector`.

Zum Weiterlesen

In diesem Abschnitt finden Sie lediglich die Einführung zur Standardcontainer-Klasse `std::array` mit den wichtigsten Funktionen zum Hausgebrauch. Sind Sie auf der Suche nach einer umfassenden Funktionsreferenz, kann ich Ihnen die Website <http://en.cppreference.com> empfehlen.

Array anlegen und direkt initialisieren

Für das Anlegen von `std::array` benötigen Sie zunächst die Headerdatei `<array>`. Alle nötigen Funktionalitäten sind im Namensraum `std` enthalten. Das Anlegen eines Arrays vom Standardcontainer `std::array` geschieht mit folgender Syntax:

```
std::array<Typ, N> Name;
```

Zwischen den spitzen Klammern schreiben Sie hier mit `Typ` den Datentyp. Die Anzahl der Elemente, welche Sie im Array speichern wollen, geben Sie daneben, durch ein Komma getrennt, mit `N` an. Der `Name` ist der Bezeichner für das Array. Ein Array mit 100 Elementen vom Datentyp `int` legen Sie hierbei wie folgt an:

```
std::array<int, 100> iarray;
```

Auch hier gilt, wie schon beim C-Style-Array, dass die einzelnen Elemente nicht mit 0 initialisiert werden, wie dies bei der Standardcontainer-Klasse `std::vector` der Fall ist. Da `std::array` ohnehin C++11 ist, können Sie diese direkt mit der neuen einheitlichen Initialisierungsliste mit Werten initialisieren. Hierzu einige Beispiele:

```
01 #include <array> // benötigter Header
02 std::array<int, 10> iarray{};
03 std::array<double, 5> darray; // Achtung!!!
04 std::array<short, 10> sarray{ 12, 34, 56, 78 };
05 std::array<long, 0> larray;
```

In Zeile **02** legen Sie ein Array vom Typ `int` mit zehn Elementen an. Durch die einheitliche Initialisierung mit den geschweiften Klammern am Ende werden automatisch auch gleich alle Elemente mit 0 initialisiert. In Zeile **03** wurde ein `double`-Array mit fünf Elementen angelegt, aber die Elemente haben hier einen undefinierten Wert. So etwas sollte man immer vermeiden, und ich denke, es dürfte kein allzu großer Aufwand sein, hier die geschweiften Klammern zu setzen.

In Zeile **04** legen wir ein `short`-Array mit zehn Elementen an. Die ersten vier Elemente werden gleich über die Initialisierungsliste mit Werten versehen. Die restlichen sechs Elemente werden dann automatisch mit 0 initialisiert. Sie können sehr schön sehen, dass hier das Prinzip der C-Style-Arrays beibehalten wurde. Ein Konstrukt wie in Zeile **05** mit null Elementen ist zwar prinzipiell möglich, aber nutzlos, weil es nicht mehr erweitert werden kann.

Zugriff auf die Elemente

Der Zugriff auf die einzelnen Elemente eines Arrays vom Standardcontainer `std::array` kann jetzt genauso komfortabel wie schon bei `std::vector` über den Indizierungsoperator `[]` oder die Methode `at()` erfolgen. Hierzu ein Beispiel mit Funktionen, die Sie bereits von `std::vector` her kennen:

```
00 // kapitel04/array.cpp
01 #include <array>
...
02 std::array<int, 10> iarray{};
03 std::cout << "Bitte Wert eingeben: ";
04 std::cin >> iarray.at(0);
05 std::cout << "Bitte Wert eingeben: ";
06 if( iarray.size() > 1 ) {
07     std::cin >> iarray[1];
08 }
09 for( int it=2; it < iarray.size(); ++it ) {
10     iarray.at(it) = it*2;
11 }
12 for( auto it : iarray ) {
13     std::cout << it << std::endl;
14 }
```

Das Listing enthält nichts Neues mehr, was nicht schon zuvor im Abschnitt zum Standardcontainer `std::vector` behandelt wurde. Daher wird auch nicht mehr auf die Einzelheiten eingegangen.

Logischerweise sind Array-erweiternde Funktionen wie `push_back(T)` wie beim Standardcontainer `std::vector` in `std::array` nicht mehr vorhanden, weil `std::array` nicht mehr dynamisch erweitert werden kann. Ebenso gibt es keine weiteren Funktionen, welche an der Größe des Arrays irgendetwas verändern könnten.

In Tabelle 4.1 finden Sie nochmals alle drei Möglichkeiten für ein Array aufgelistet.

Typ	Beispiel	Dynamisch	Funktionen
C++-Vektor	<code>std::vector<int> x(10);</code>	ja	ja
C-Style-Array	<code>int x[10];</code>	nein	nein
C++11-Array	<code>std::array<int, 10> x;</code>	nein	ja

Tabelle 4.1 Ein Überblick über die sequenziellen Datentypen. Mit »Dynamisch« ist hier die Speichergröße gemeint und mit »Funktionen«, ob Memberfunktionen (Methoden) vorhanden sind.

4.1.4 Assoziatives Array

Das Thema ist an dieser Stelle vielleicht noch etwas zu früh und komplex, aber es sollte trotzdem zumindest kurz erwähnt werden, wenn wir schon beim Thema Array sind. Es wurde bereits anfangs erwähnt, dass assoziative Arrays eigentlich keine Arrays im eigentlichen Sinne sind.

Zum Weiterlesen

Die assoziativen Arrays werden nochmals ausführlicher in Abschnitt 12.2.2, »Assoziative Containerklassen«, zu den Standardcontainern von C++ behandelt.

Bei einem assoziativen Array wird, statt einer gewöhnlichen Ganzzahl, ein nichtnumerischer Schlüssel (meistens eine Zeichenkette) verwendet, um die enthaltenen Elemente zu adressieren. In der Praxis können Sie sich diese wie folgt vorstellen:

```
postleitzahlen["Frankfurt"] = 60311;
```

In C++ können Sie ein solches assoziatives Array mit den Standardcontainern `std::map` oder `std::unordered_map` (neu dabei in C++11) erstellen. Beide Container haben ebenfalls den `[]`-Operator implementiert. Hierzu ein Beispiel, wie Sie dies in C++ realisieren können, ohne an dieser Stelle schon allzu umfangreich darauf einzugehen, weil hier auf einige Dinge vorgegriffen wird:

```
00 // kapitel04/assoziativArray.cpp
01 #include <string>
02 #include <map>
...
03 std::map<std::string, int> postleitzahlen;
04 postleitzahlen["Frankfurt"] = 60311;

05 std::string ort("");
06 unsigned int plz=0;
07 std::cout << "Ort: ";
08 std::cin >> ort;
09 std::cout << "Postleitzahl: ";
10 std::cin >> plz;
11 postleitzahlen[ort]=plz;
12 std::cout << "Postleitzahl von " << ort
    << " : " << postleitzahlen[ort] << std::endl;
13 for( const auto& pos : postleitzahlen ) {
14     std::cout << pos.first << " : "
        << pos.second << std::endl;
15 }
```

In Zeile 03 legen wir ein leeres assoziatives Array mit dem Standardcontainer `map` an. Der Schlüssel (engl. *key*) ist vom Typ `std::string` und der Wert (engl. *value*) vom Typ `int`. So definiert, können Sie jetzt eine Zeichenkette als Schlüssel zwischen dem `[]`-Operator setzen und so einen bestimmten Wert zuweisen, wie in Zeile 04 demonstriert wird. Dasselbe funktioniert auch interaktiv über `std::cin` und `std::cout`, wie in den Zeilen 05 bis 12 gezeigt wird. Auch ganz komfortabel lassen sich alle Elemente des assoziativen Arrays mit der neuen Range-basierten `for`-Schleife von C++11, wie sie in den Zeilen 13 bis 15 verwendet wurde, durchlaufen und (hier) ausgeben. Mit `first` erreichen Sie den Wert des Schlüssels und mit `second` den Wert, der dem Schlüssel zugeordnet ist.

4.2 Strings (Zeichenketten)

Eine Zeichenkette (engl. *String*) ist eine sequenzielle Folge von Zeichen aus einem definierten Zeichensatz – genau genommen ist eine Zeichenkette wiederum nur ein Array vom Datentyp `char`. In C++ stehen Ihnen zwei Möglichkeiten von Zeichenketten zur Verfügung. Da wäre zum einen der Standardcontainer `std::string` für eine variable Länge von Zeichen mit vielen nützlichen Methoden. Auf der anderen Seite finden Sie auch hier Zeichenketten in roher Form wieder – auch C-Style-Strings genannt.

4.2.1 Standardcontainer `std::string`

Die Standardklasse `std::string` erlaubt einen bequemen und sichereren Umgang mit Strings und bietet alle gängigen Funktionen, die man mit Strings gewöhnlich verwendet und benötigt. Auch um den Speicherplatz muss man sich keine Gedanken machen. Um `std::string` verwenden zu können, müssen Sie die Headerdatei `<string>` einbinden. Sämtliche Typdefinitionen, Instanziierungen, globale Funktionen und Operatoren sind auch hier im Namensraum `std` enthalten.

Zum Weiterlesen

In diesem Abschnitt finden Sie lediglich die Einführung zur Standardcontainer-Klasse `std::string` zum Hausgebrauch. Sind Sie auf der Suche nach einer umfassenden Funktionsreferenz, kann ich Ihnen die Website <http://en.cppreference.com> empfehlen.

Die Verwendung von `std::string` ist denkbar einfach, wie schon bei gewöhnlichen Datentypen. Einen String anzulegen und mit einem Inhalt zu initialisieren, ist auf verschiedene Arten möglich. Einen Überblick darüber soll der folgende Codeausschnitt zeigen:

```

00 // kapitel04/string01.cpp
01 #include <string>
...
02 std::string str01("Eine Textfolge");
03 std::string str02 = str01;
04 std::string str03{"C++11-like"}; // C++11
05 std::string str04 = "So geht es auch";
06 std::string str05(4, 'X');
07 std::string str06;
08 std::cout << "Einen String bitte: ";
09 std::getline(std::cin, str06);
10 std::cout << "Ihre Eingabe: " << str06 << std::endl;

```

In Zeile **02** erzeugen Sie einen String, der auch gleich initialisiert wird. Mit einer einfachen Zuweisung wird der String kopiert, wie dies in Zeile **03** demonstriert wird. Selbstverständlich können Sie auch die vereinheitlichte C++11-Initialisierungssyntax verwenden, wie Zeile **04** zeigt. In Zeile **05** wird ein String beim Erzeugen gleich mit einem konstanten String versehen. Einen String können Sie auch gleich mit einer bestimmten Anzahl an Zeichen füllen, wie Zeile **06** zeigt, wo der String mit vier 'X'-Zeichen gefüllt wird. In Zeile **07** wird ein leerer String erzeugt.

In Zeile **09** wird von der Tastatur über den Stream `std::cin` mit der Funktion `getline()` eine ganze Zeile eingelesen. Würden Sie hier den Eingabeoperator `>>` verwenden, würde nur bis zum ersten Leerzeichen eingelesen. Die Ausgabe über `std::cout` funktioniert wie schon bei gewöhnlichen Datentypen, wie Sie in Zeile **10** auch sehen können.

std::vector<char>

Wie bereits erwähnt wurde, sind die Strings wiederum nichts anderes als ein Array vom Typ `char`. Sie könnten dafür durchaus `std::vector<char>` verwenden. Aber `std::string` ist doch etwas spezieller, und es werden bei Strings auch speziellere Funktionen als bei gewöhnlichen Arrays benötigt. Daher sollten Sie für Strings auch auf die Klasse `std::string` zurückgreifen.

Zugriff auf einzelne Zeichen

Auch beim Zugriff auf einzelne Zeichen müssen Sie bei `std::string` auf nichts verzichten und können auch schon wie bei `std::vector` auf den Indizierungsoperator `[]` und die Methode `at()` zurückgreifen – natürlich auch hier mit dem Unterschied, dass mit `[]` nicht auf eine Bereichsüberschreitung geprüft wird, wie dies mit `at()` der Fall ist. Das folgende Beispiel soll Ihnen kurz den Zugriff auf die einzelnen Zeichen in `std::string` demonstrieren:

```

00 // kapitel04/string02.cpp
01 #include <string>
...
02 std::string str01("Eine Textfolge");
03 std::cout << "3. Buchstabe: " << str01.at(2)
    << std::endl;
04 std::cout << "Letzter Buchstabe: " << str01.back()
    << std::endl;
05 std::cout << "6. Buchstabe: " << str01[5] << std::endl;
06 for( size_t i=0; i < str01.size(); ++i ) {
07     if( str01.at(i) == 'e' ) {
08         str01.at(i) = 'a';
09     }
10     std::cout << str01.at(i) << "-";
11 }
12 std::cout << std::endl;
13 for( const auto& ch : str01 ) { // C++11-like
14     std::cout << ch << ":";
15 }
16 std::cout << std::endl;

```

In Zeile **03** wird auf den dritten Buchstaben des Strings `str01` zugegriffen und dieser ausgegeben. Die Elementfunktion `back()`, welche in Zeile **04** verwendet wird und auf den letzten Buchstaben zugreift, ist erst seit C++11 dabei. Das Gegenstück dazu wäre die Elementfunktion `front()` (auch neu in C++11). Beide Elementfunktionen sind bei `std::vector` schon länger dabei. Der Zugriff mit dem Indizierungsoperator `[]` ist auch hier dabei (Zeile **05**), nur wird bei einer Bereichsüberschreitung hiermit keine Ausnahme geworfen. In den Zeilen **06** bis **12** durchlaufen wir den kompletten String Buchstabe für Buchstabe und geben diesen auf dem Bildschirm aus. Alle Buchstaben 'e' im String werden hierbei durch ein 'a' ersetzt. Auch hier kann man wieder die Range-basierte `for`-Schleife aus C++11 verwenden, wie dies in den Zeilen **13** bis **16** getan wurde.

4.2.2 Rohe Strings (C-Style-Strings)

Die rohen Strings (oder C-Style-Strings) sind wieder gewöhnliche Arrays vom Typ `char`. Daher gilt alles, was Sie bereits über die C-Style-Arrays erfahren haben (siehe letzter Abschnitt) auch für C-Style-Strings. Hinzu kommt noch bei den rohen Strings, dass Sie sich zusätzlich um das String-Endezeichen `'\0'` (auch Nullzeichen mit dem Wert 0) kümmern müssen, was das Arbeiten mit den C-Style-Strings recht umständlich und (wie die Vergangenheit gezeigt hat) fehleranfällig macht. Es musste hierbei also immer darauf geachtet werden, dass noch Platz für ein Zeichen mehr vorhanden ist. Für Arbeiten mit den C-Style-Strings finden Sie in der Headerdatei `<cstring>`

(nicht zu verwechseln mit `<string>`) noch Funktionen für Vergleiche, Kopieren, Zusammenfügen usw. – aber auch diese Funktionen wurden in der Vergangenheit häufig falsch verwendet.

Verwenden Sie `std::string` statt C-Style-Strings

Dies sollte man gleich vorweg sagen. Sofern Sie keine eigene Klasse auf tieferer Ebene schreiben müssen, spricht in reinem C++-Code nichts mehr für die C-Style-Strings. Das Buch wird daher nur kurz auf die alten C-Style-Strings eingehen, weil es ja auch alten Code gibt und Sie vielleicht einen C-Style-String in einen `std::string` oder umgekehrt umwandeln müssen, weil es eine Schnittstelle so will.

Der folgende Codeausschnitt soll Ihnen kurz zeigen, wie Sie C-Style-Strings verwenden können:

```
00 // kapitel04/carray01.cpp
...
01 char carray01[16];
02 char carray02[20] = {'a', 'b', 'c', '\0'};
03 char carray03[20] = {"abc"};
04 char carray04[] = {"abc"};

05 std::cout << "Eingabe machen: ";
06 std::cin >> carray01; // Gefährlich!!!
07 std::cout << "Ihre Eingabe: " << carray01 << std::endl;

08 std::cout << "Eingabe machen: ";
09 std::cin.getline(carray01, 15);
10 std::cout << "Ihre Eingabe: " << carray01 << std::endl;
```

Der leere String in Zeile **01** kann 16 Zeichen speichern. Beachten Sie, dass ein so leer angelegtes Array mit undefinierten Werten vorbelegt ist. Im Array der Zeile **02** können 20 Zeichen gespeichert werden. Die ersten vier Zeichen werden mit 'a', 'b', 'c' und '\0' vorbelegt. Die restlichen Zeichen werden automatisch mit 0 initialisiert (genauer '\0'). Die umständliche Angabe von Zeichen für Zeichen können Sie sich sparen. Dasselbe wie in Zeile **02** bewirken Sie auch in Zeile **03**. Bei der Verwendung von doppelten Hochkommata können Sie auf das String-Endezeichen verzichten, weil dies dann automatisch hinzugefügt wird. Bei der Initialisierungsliste darf auch die Indexangabe fehlen, wenn Sie, wie in Zeile **04**, ein konstantes Stringliteral zuweisen.

Das Einlesen eines C-Style-Strings wie in Zeile **06** über `std::cin` ist sehr bedenklich, weil hier keine Längenüberprüfung stattfindet. Außerdem haben Sie auch hierbei das Problem, dass `std::cin` beim ersten Leerzeichen das Einlesen abbricht.

Die bessere Alternative stellt hier die Methode `getline()` dar, wie diese in Zeile 09 verwendet wird. Hiermit können Sie die Länge der Eingabe beschränken, und zusätzlich werden hier auch die Leerzeichen mit eingelesen.

Zugriff auf einzelne Zeichen

Der Zugriff auf einzelne Zeichen funktioniert auch mit dem Indizierungsoperator `[]`. Ein Zugriff auf einzelne und alle Zeichen soll der folgende Codeausschnitt demonstrieren:

```
00 // kapitel04/carray02.cpp
01 #include <cstring> // für strlen()
...
02 char carray01[16]="Hello Welt";
03 std::cout << "2. Buchstabe: " << carray01[1]
    << std::endl;
04 for(size_t it=0; it < strlen(carray01); ++it) {
05     if(carray01[it] == 'e') {
06         carray01[it] = 'a';
07     }
08     std::cout << carray01[it] << ":";
09 }
10 std::cout << std::endl;

11 for( auto it : carray01 ) {
12     std::cout << it << "-";
13 }
14 std::cout << std::endl;
```

In Zeile 03 wird mit `[]`-Operator ungeprüft (nicht zu empfehlen) auf ein einzelnes Zeichen zur Ausgabe zugegriffen. In der `for`-Schleife (Zeile 04 bis 09) werden alle Buchstaben durchlaufen. Die Anzahl der Zeichen erhalten Sie mit der Funktion `strlen()` aus der Headerdatei `<cstring>` (ohne das String-Endezeichen `!`). In der Schleife ersetzen wir außerdem den Buchstaben `'e'` durch den Buchstaben `'a'`, wenn die `if`-Bedingung aus Zeile 05 `true` zurückliefert. Natürlich können Sie den kompletten (!) String auch mit der neuen Range-basierten `for`-Schleife durchlaufen, wie dies in den Zeilen 11 bis 13 gemacht wurde.

`std::string` in C-Style-String umwandeln

Wie bereits erwähnt, werden Sie in der Praxis in C++ wohl eher selten noch reine C-Style-Strings implementieren. Häufiger allerdings werden Sie es mit älteren Schnittstellen zu tun bekommen, welche einen rohen C-Style-String erwarten. Für solche Zwecke stehen Ihnen drei Möglichkeiten zur Verfügung:

1. Die Methode `c_str()` der Klasse `std::string` gibt einen gültigen C-Style-String aus einem `std::string` zurück (inklusive `'\0'`).
2. Die Methode `data()`. Beachten Sie allerdings, dass diese Elementfunktion (vor C++11) keinen gültigen C-Style-String zurückgibt, weil diese nicht garantiert, dass `'\0'` am Ende hinzugefügt wird.
3. Mit der Methode `copy()` können Sie eine bestimmte Anzahl an Zeichen eines `std::string` in einen C-Style-String kopieren. Auch hier wird kein `'\0'` am Ende hinzugefügt, was Sie daher selbst machen müssen.

Hierzu einige Beispiele, wie Sie diese Möglichkeiten in der Praxis verwenden können:

```
00 // kapitel04/string2cstring.cpp
01 #include <string>
...
02 void funktion(const char* cstr) {
03     std::cout << cstr << std::endl;
04 }

05 std::string str("Ein einfacher Text");
06 funktion(str.c_str());

07 const char* ptr;
08 ptr = str.c_str();
09 funktion(ptr);

10 str += " mehr Text"; // Achtung!!!
11 funktion(ptr);      // Argument ptr ungültig

12 char buf[32];
13 size_t length = str.copy(buf, 32);
14 buf[length]='\0';
15 std::cout << "C-Style-String: " << buf << std::endl;
```

In den Zeilen **02** bis **04** finden Sie eine einfache Funktion, die eine alte Schnittstelle darstellen soll, die einen C-Style-String erwartet. Beachten Sie, dass es in der realen Welt durchaus üblich ist, dass Sie den Code zu solch einer Schnittstelle nicht zur Verfügung haben. Die Schnittstelle verwendet hier einen **Zeiger**, welche erst in Abschnitt 4.3, »Zeiger«, behandelt werden. Kümmern Sie sich bei diesem Beispiel zunächst noch nicht darum. In Zeile **06** führen wir eine Konvertierung von `std::string` zu einem C-Style-String gleich im Funktionsaufruf durch. Sie sind nicht gezwungen, die Konvertierung gleich hier durchzuführen. Sie können den String auch vor der Verwendung (wie hier mit dem Funktionsaufruf) konvertieren und dann diesen C-Style-String als Argument an die Funktion übergeben, wie dies in den Zeilen **07** bis **09**

demonstriert wird. In Zeile 10 werden zum String noch weitere Zeichen hinzugefügt, wodurch `ptr` ungültig wird. Daher ist die Verwendung aus Zeile 11 nicht mehr gültig, obwohl Sie in den meisten Fällen vermutlich zu funktionieren scheint. Der weitere Verlauf ist allerdings jetzt undefiniert.

Konvertierten String gleich verwenden

Sie sollten daher einen `std::string` in einem C-Style-String erst dann konvertieren, wenn Sie diesen benötigen, und ihn dann sofort verwenden. Sobald eine nicht konstante Elementfunktion `std::string` verändert, wird auch der C-Style-String ungültig!

In den Zeilen 12 bis 15 wird außerdem noch die Konvertierung eines `std::string` in einen C-Style-String mithilfe der Elementfunktion `copy()` demonstriert.

C-Style-String einem `std::string` zuweisen

Wollen Sie hingegen einen C-Style-String einem `std::string` zuweisen oder mit `std::string`-Objekten mischen (vergleichen, hinzufügen, anhängen usw.), so dürften Sie so gut wie nie Probleme bekommen. In der Regel findet hier fast immer eine automatische Typkonvertierung von `const char*` nach `std::string` statt.

`char*` vs. `const char*`

Der Typ für Stringlitterale wurde im C++-Standard von `char*` nach `const char*` geändert. Daher erhalten Sie auch eine Warnmeldung des Compilers, wenn Sie bspw. ein Stringliteral vom Typ `char*` einem `std::string` zuweisen. Damit alter C++-Code weiterhin abwärtskompatibel ist, funktioniert die implizite, aber veraltete (»deprecated«) Konvertierung von `char*` nach `std::string` nach wie vor.

Im folgenden Listing finden Sie einfache Zuweisungen von C-Style-Arrays an `std::string`-Objekte. In den meisten Fällen funktionieren die Zuweisungen ohne Beanstandungen vom Compiler. Ähnlich einfach funktioniert diese auch mit den vielen vorhandenen Methoden von `std::string`.

```
00 // kapitel04/cstring2string.cpp
01 #include <string>
...
02 char carray01[100] = {"Ein einfacher Text"};
03 char carray02[] = {"Viele C-Style-Strings"};
04 const char *cstr01 = {"Noch mehr Text"};
05 char *cstr02 = {"Viel mehr Text"}; // Deprecated...

06 std::string str01(carray01);
```

```
07 std::string str02 = carray02;
08 std::string str03{cstr01};
09 str03 += carray01;
10 std::string str04={cstr02}; //... but still works
```

Dem Beispiel muss nicht mehr viel hinzugefügt werden. Über Zeile 05 sollte sich Ihr Compiler beschweren, weil das Stringliteral hier nicht `const` ist, was der C++-Standard aber fordert. Aber, wie bereits erwähnt, lässt sich die Zuweisung in Zeile 10 aus abwärtskompatiblen Gründen trotzdem ausführen.

4.2.3 Unterstützung von Unicode (C++11)

Für breitere Strings wie Unicode oder den asiatischen Zeichensatz wurden neben `std::string` mit `std::wstring`, `std::u16string` und `std::u32string` weitere drei Typen vordefiniert. `std::u16string` und `std::u32string` kamen neu mit C++11 hinzu (Tabelle 4.2). Alle drei Typen sind wie auch `std::string` eine Spezialisierung der Klasse `basic_string<>`. Alle drei Typen von Strings haben im Grunde dieselben Schnittstellen wie `std::string`. Somit ist im Grunde alles, was Sie von `std::string` her kennen, auch hier gültig.

Stringtyp	Codierung	Zeichentyp	Präfix
string	UTF-8	char	"Standard-String" u8"Standard-String"
u16string	UTF-16	char16_t	u"UTF-16-String"
u32string	UTF-32	char32_t	U"UTF-32-String"
wstring	2 oder 4 Bytes	wchar_t	L"Wide-String"

Tabelle 4.2 Übersicht zu den neuen Stringtypen

In Abschnitt 2.4.2, »Unicode-Unterstützung«, in dem Sie die Typen `char16_t` und `char32_t` kennengelernt haben, wurde die neue Unicode-Unterstützung bereits kurz erwähnt. Aus diesen beiden neuen Zeichentypen werden dann die neuen Stringtypen `u16string` und `u32string` gebildet:

```
01 string str_u8{u8"Eurozeichen: \u20AC"};
02 u16string str_u16{u"Eurozeichen: \u20AC"};
03 u32string str_u32{U"Eurozeichen: \u20AC"};
04 wstring str_wide{L"Eurozeichen: \u20AC"};
```

In Zeile 01 sehen Sie einen UTF-8-codierten String, der auf `char` basiert. Unicode-Stringlitterale für UTF-8 werden mit dem Präfix `u8` in den Quelltext geschrieben. In

Zeile **02** sehen Sie einen UTF-16-codierten String, basierend auf `char16_t`, der in einem `u16string` gespeichert wird. Für ein UTF-16-Stringliteral benötigen Sie das Präfix `u`. Dasselbe nochmals in Zeile **03** nur als UTF-32-codierter String, basierend auf `char32_t` und mit dem Präfix `U`. Schon seit C++98 existierten auch Wide-Strings mit `wstring` (siehe Zeile **04**), welche auf `wchar_t` basieren, aber aus mangelnder Portabilität weniger zu gebrauchen sind. Diese Wide-Stringlitterale benötigen das Präfix `L`.

Konvertierungen

Auch für die Konvertierung zwischen den verschiedenen Codierungen stehen mit dem Klassen-Template `codecvt` verschiedene Facetten zur Verfügung. Zusätzlich steht hierfür mit dem neuen `wstring_convert<...>` aus dem Namensraum `std` eine Möglichkeit zur Verfügung, Zeichenketten direkt ineinander umzuwandeln. Zwischen den spitzen Klammern von `wstring_convert` müssen Sie nur die entsprechende Konvertierungsfacetten eingeben. Ohne hier genauer darauf einzugehen, könnte eine Konvertierung von UTF-16 nach UTF-8 theoretisch wie folgt realisiert werden:

```
00 #include <locale>
01 u16string str_u16{u"Eurozeichen: \u20AC"};
02 wstring_convert<codecvt_utf8_utf16<char16_t>,char16_t> cv;
03 string str16to8 = cv.to_bytes(str_u16);
04 u16string str8to16 = cv.to_bytes(str16to8);
```

In Zeile **01** sehen Sie einen UTF-16-codierten String. In Zeile **02** erzeugen wir ein neues `wstring_convert`-Objekt mit der Facette `codecvt_utf8_utf16` für eine Konvertierung von UTF-16 nach UTF-8. Jetzt ist es ein Leichtes, mithilfe des Objekts den UTF-16-String zu einem UTF-8-String zu konvertieren, wie es in Zeile **03** gemacht wird. Mit `wstring_convert` kann dann die Konvertierung auch in die andere Richtung erfolgen, wie Zeile **04** zeigen soll.

Tabelle 4.3 zeigt, welche Facetten Ihnen zur Verfügung stehen, die Sie zwischen die spitzen Klammern von `wstring_convert<...>` setzen können.

Facette	Konvertierung
<code>codecvt_utf8_utf16<char16_t></code>	UTF-16/UTF-8
<code>codecvt_utf8<char32_t></code>	UTF-8/UTF-32
<code>codecvt_utf8<char16_t></code>	UTF-8/UCS-2/4
<code>codecvt<char16_t,char,std::mbstate_t></code>	UTF-16/UTF-8
<code>codecvt<char32_t,char,std::mbstate_t></code>	UTF-32/UTF-8

Tabelle 4.3 Neue Konvertierungsfacetten

Zur Drucklegung des Buches waren gerade die neuen Konvertierungsfacetten noch nicht richtig implementiert und funktionierten teilweise noch gar nicht, daher ist auch der Code hier nur rein theoretischer Natur.

4.2.4 Rohstringliterale (Raw-String) (C++11)

Wollen Sie statt der Steuerzeichen wie `\n`, `\t`, " oder einfach nur `\` auf dem Bildschirm ausgeben, mussten Sie diese bisher immer mit einem weiteren `\` schützen, damit der Compiler dieses Zeichen nicht mehr als Steuerzeichen verwendet. Ein Beispiel:

```
std::string str("Zeile\\nTabulator\\tZeile\\n");
```

Damit werden auf dem Bildschirm auch die Zeichen `\n` und `\t` mit ausgegeben und nicht als Steuerzeichen für eine neue Zeile bzw. als Zeilenvorschub interpretiert. Mit C++11 können für solche Zwecke jetzt rohe Stringliterale verwendet werden. Solche RAW-Stringliterale werden mit `R"(Raw String)"` definiert. Bezogen auf den eben gezeigten String `str` würde dies wie folgt aussehen:

```
std::string rawStr( R"(Zeile\nTabulator\tZeile\n)");
```

Auch hier werden jetzt die Zeichen `\n` und `\t` auf dem Bildschirm ausgegeben und nicht als Steuerzeichen verwendet. Nur klappt dies jetzt ohne das Voranstellen des `\`-Zeichens. Der Vorteil dürfte bei diesen Beispielen vielleicht etwas trivial sein, aber sobald Sie komplexe reguläre Ausdrücke (siehe Kapitel 14, »Reguläre Ausdrücke (C++11)«) definieren, dürften Sie die rohen Stringliterale zu schätzen wissen. Ein Ausdruck, mit dem Sie bspw. überprüfen, ob es sich bei einem String um eine E-Mail-Adresse handelt, kann als rohes Stringliteral wie folgt definiert werden:

```
std::regex exp(R"([\w-]+(?:\.[\w-]+)*@(?:[\w-]+\.)+[a-zA-Z]{2,7})");
```

Ohne das neue rohe Stringliteral würde dieser Ausdruck viel wüster aussehen:

```
std::regex exp("[\\w-]+(?:\\.\\[\\w-]+)*@(?:[\\w-]+\\.\\.)+[a-zA-Z]{2,7}");
```

Heredoc mit rohen Stringliteralen

Wer sich mit der Shell-Programmierung oder mit Programmiersprachen wie Perl, PHP, Python oder Ruby auskennt, der dürfte wohl auch mit *Heredoc* (auch here document) vertraut sein. Bei sogenannten *Heredocs* lassen sich ganze Textabschnitte definieren, in denen auch enthaltene Steuerzeichen, Einzüge und Zeilenumbrüche im Text bewahrt bleiben. Ein Beispiel:

```
std::string heredocs = R"xxx(
/*****
/* Hier kann alles hingeschrieben werden;      */
/* was Sie wollen; auch Zeichen wie \n \t usw.  */
/* werden so ausgegeben, wie Sie diese hier    */
/* sehen können.                               */
*****/
)xxx";
```

Die Ausgabe vom String heredoc ist hier tatsächlich:

```
/*****
/* Hier kann alles hingeschrieben werden;      */
/* was Sie wollen; auch Zeichen wie \n \t usw.  */
/* werden so ausgegeben, wie Sie diese hier    */
/* sehen können.                               */
*****/
```

Obwohl der komplette Text eigentlich als Kommentar eingefasst wurde, wird dieser als String interpretiert und mitsamt den enthaltenen Steuerzeichen so ausgegeben, wie Sie diesen tatsächlich geschrieben haben.

4.3 Zeiger

Ich habe mir lange überlegt, die Zeiger in einen Anhang zu verbannen, aber mich dann doch dazu entschlossen, diese hier bei den fortgeschrittenen Typen zu behandeln. Sie können an dieser Stelle durchaus dieses Kapitel überspringen und bei Bedarf nachholen.

Mit den (rohen) Zeigern (engl. *Pointer*) verhält es sich wie schon mit den rohen Arrays bzw. rohen Strings. An Zeigern haftet ebenfalls ein schlechtes Image angestaubten Cs, und diese werden in C++ deshalb so ziemlich vermieden. Trotzdem sollte man als C++-Programmierer die Zeiger zumindest im Repertoire haben. In der gängigen C++-Praxis gibt es zwar genügend Alternativen, um nicht auf Techniken wie rohe Zeiger zurückgreifen zu müssen (wie Sie in diesem Abschnitt teilweise auch erfahren werden). Aber auch hier gilt, wie schon bei den rohen Arrays, dass es sein kann, dass Sie eine alte Schnittstelle zu einer Klasse oder Funktion haben, bei der eben rohe Zeiger verwendet werden. Daher kann es nützlich sein, sich damit auszukennen. Außerdem gibt es dann doch das eine oder andere Thema, für das man zumindest die Grundlagen von Zeigern kennen sollte (bspw. komplexe Datenstrukturen wie binäre Bäume oder Polymorphie, siehe Abschnitt 9.7, »Polymorphie mithilfe von virtuellen Methoden«).

Die Alternativen, um keine Zeiger verwenden zu müssen

Anstatt überhaupt dynamische Sachen mithilfe von rohen Zeigern zu realisieren, sollten Sie immer die fertigen Sachen von C++ verwenden, welche Ihnen die Standardbibliothek in Form verschiedener Container anbietet (bspw. `std::vector`, `std::array`, `std::string` usw.). Diese Standardcontainer übernehmen alles Ungemütliche für Sie, so dass Sie sich nur noch um deren »Verwendung« und nicht »Implementierung« kümmern müssen. Und wenn Sie dann doch mal in Verlegenheit kommen, dynamisch zur Laufzeit des Programms Speicherplatz reservieren zu müssen, dann sollten Sie sich (wenn möglich) die neuen klugen Zeiger (Smart Pointer) von C++11 ansehen, anstatt `new` und `delete` zu verwenden. Mehr zu den Smart Pointern können Sie in Abschnitt 16.3, »Smart Pointer (C++11)«, nachlesen. Dann wäre da noch die Möglichkeit, Zeiger als Parameter für Funktionen oder Elementfunktionen zu verwenden. Auch hier finden Sie mit den Referenzen (siehe Abschnitt 4.4, »Referenzen«) eine passende Alternative.

Trotzdem muss angemerkt werden, dass hier rohe Zeiger nicht als »schlecht« abgewertet werden sollen. So arbeiten intern ja schließlich auch die Standardbibliotheken und Standardcontainer (mit deren Iteratoren) mit Zeigern, und vieles wird bzw. kann dort (nur) über Zeiger realisiert (werden). Es wird nur eben vor dem C++-Anwender versteckt, damit sich dieser nicht mehr um umständliche Implementierungen und die vielen (fehleranfälligen) Kleinigkeiten kümmern muss. Die Sprache C++ ist hier mehr eine »Bereit-zur-Verwendung«-Sprache geworden, und Sie müssen sich weniger Gedanken um die Implementierung einer bestimmten Technik machen. Natürlich lässt Ihnen C++ nach wie vor die Freiheiten, alles selber zu machen, aber in der Praxis ist dies wohl eher selten sinnvoll und meistens sehr fehleranfällig.

Zeiger deklarieren

Der Sinn und Zweck von Zeigern ist es, anstatt ganze Datenmengen im Speicher hin und her zu bewegen, einfach nur die virtuelle Anfangsadresse auf diese Datenmenge zu verwenden. Zeiger speichern hierbei nur diese Anfangsadresse. Hier die Syntax eines solchen Zeigers:

```
Typ *name;
```

Den Zeiger erkennen Sie am Sternchen zwischen dem Typ und dem Namen. Da Zeiger strengstens typisiert sind, muss der Typ des Zeigers auch dem Typ der Daten entsprechen, auf die dieser verweist (oder auch einfach nur »zeigt«). Mit `name` geben sie den Bezeichner an, mit dem Sie den Zeiger im Programm dann verwenden können. Das kennen Sie bereits von den gewöhnlichen Datentypen. Einen Zeiger, der nur auf `int`-Typen verweisen kann, würden Sie somit wie folgt deklarieren:

```
int *iptr;
```

Der Zeiger mit dem Bezeichner `iptr` kann jetzt nur verwendet werden, um auf Adressen von `int`-Objekten zu verweisen.

Speichergröße von Zeigern

Die Speichergröße von Zeigern hängt **nicht** vom Typ des Zeigers ab. Ein Zeiger benötigt letztendlich so viel Platz, um eine Speicheradresse speichern zu können.

Streng typisiert und Zeigerarithmetik

Zeiger sind streng typisiert, damit eine gewisse Zeigerarithmetik (Vergleiche, Subtrahieren zweier Zeiger, Addition/Subtraktion eines Zeigers und einer Ganzzahl) auch funktioniert. Wenn Sie bspw. einen Zeiger auf ein Array verwenden würden, kann nur anhand der Speichergröße des assoziierten Typs die Adresse der Vorgänger- oder Nachfolgerelemente berechnet werden.

Adresse im Zeiger speichern

Um einen Zeiger verwenden zu können, müssen Sie darauf achten, dass der Zeiger eine gültige Adresse erhält und nicht irgendwohin verweist. Hier liegt häufig die Gefahr bei der Verwendung von rohen Zeigern. Der folgende Codeausschnitt zeigt Ihnen, wie Sie einen Zeiger vom Typ `int` auf einen einfachen Datentyp vom Typ `int` verweisen lassen:

```
01 int *ptr01 = 0L;
02 int ival = 123456;
03 ptr01 = &ival;
```

In Zeile **03** haben Sie mithilfe des Adressoperators `&` dem Zeiger `ptr01` die Speicheradresse der Variablen `ival` zugewiesen. Die Verwendung des Adressoperators ist hier von ganz großer Bedeutung, weil Sie sonst, ohne den Adressoperator, dem Zeiger quasi den Wert `123456` übergeben würden, und sobald der Zeiger mit dieser Adresse (Zeiger speichern Adressen!) `123456` arbeiten würde, wäre der weitere Ablauf des Programms undefiniert.

Da auch hier der Inhalt des Zeigers, der ja eine Adresse ist, auf einen zufälligen Wert verweist, sollten Sie auch einen Zeiger gleich von Anfang an mit einem gültigen Wert (`0`, `0L` oder mit C++11: `nullptr`) initialisieren, wie hier in Zeile **01** geschehen.

Adressen ermitteln

Sie können die Adressen von Datentypen und Zeigern jederzeit mit dem Adressoperator verwenden und auch ausgeben. Angaben wie `&daten_Typ` oder `&zeiger_auf_typ` geben Ihnen die fixe Adresse im Speicher zurück:

```
01 int *ptr = 0L;
02 int ival = 123456;
03 ptr = &ival;
04 std::cout << *ptr << std::endl;
05 std::cout << ptr << std::endl;
06 std::cout << &ptr << std::endl;
07 std::cout << &ival << std::endl;
```

In Zeile **04** greifen Sie indirekt auf den Wert von `ival` zurück (auch *Dereferenzierung* genannt). Zeile **05** gibt die Adresse aus, welche der Zeiger speichert. Hier ist es die Adresse von `ival`. In Zeile **06** wird die Adresse des Zeigers selbst und in Zeile **07** die Adresse von `ival` ausgegeben.

Zeiger dereferenzieren

Für den Zugriff auf den Wert eines Speicherobjekts, auf das ein Zeiger verweist, wird der Indirektionsoperator `*` verwendet. Dieser indirekte Zugriff auf den Wert eines Speicherobjekts wird auch als *Dereferenzierung* bezeichnet. Betrachten Sie den folgenden einfachen Codeausschnitt dazu:

```
00 // kapitel04/zeiger01.cpp
...
01 int *ptr=0L;
02 int ival = 123456;
03 ptr = &ival;
04 *ptr = 654321;

05 std::cout << *ptr << std::endl; // 654321
06 std::cout << ival << std::endl; // 654321
```

In Zeile **03** bekommt der Zeiger `ptr` die Adresse von `ival`. In Zeile **06** greifen Sie jetzt mit dem Indirektionsoperator `*` vor dem Bezeichner des Zeigers indirekt auf den Wert von `ival` zurück und ändern den Wert von `ival` von 123456 auf 654321, wie die indirekte (Zeile **05**) und direkte Ausgabe (Zeile **06**) über `std::cout` anschließend auch beweisen.

Null-Zeiger: 0, 0L und nullptr

Wenn Sie mit rohen Zeigern arbeiten, sollten Sie sich angewöhnen, den Wert (bzw. die Adresse) zu überprüfen. Die Verwendung von undefinierten Zeigern führt zu einem Speicherzugriff-Fehler und somit zu einem undefinierten Verhalten für den weiteren Programmablauf. Ob ein Zeiger auf ein gültiges Speicherobjekt verweist, können Sie mit einer Überprüfung auf 0, 0L oder nullptr (neu in C++11) testen:

```
int *ptr=0L;
if( iptr == 0L ) { // Alternativ auch: if(!iptr)
    // Fehler! Ungültige Adresse !!
}

// ... oder C++11-like:
int *ptr{nullptr}; // Alternativ auch: if(!iptr)
if( iptr == nullptr ) {
    // Fehler ungültige Adresse
}
```

Der nullptr wurde mit C++11 eingeführt, um die Lesbarkeit des Programms zu verbessern und um gefährliche Zweideutigkeiten zu vermeiden. Damit wird bspw. deutlich, dass es sich hierbei um einen Zeiger (nullptr) und nicht um eine Ganzzahl (0 bzw. 0L) handelt.

NULL aus <cstdlib>

Das Makro NULL aus der Headerdatei <cstdlib> ist ein Relikt aus alten Zeiten und gehört eher zur C-Programmierung. Es ist in C++ eigentlich nur aus Kompatibilitätsgründen vorhanden und sollte in der Praxis nicht mehr verwendet werden.

Hier ein kurzer Codeausschnitt, welcher die Probleme mit NULL in der Vergangenheit kurz aufzeigen soll:

```
01 void foo(const char *str) { cout << "const char*\n";}
02 void foo(int val) { cout << "int\n";}
...
03 foo("Text"); // foo(const char*)
04 foo(100); // foo(int)
05 foo(NULL); // ??? Und jetzt ???
06 foo(nullptr); // foo(const char*)
```

Der Funktionsaufruf in Zeile 05 soll Ihnen das Problem zeigen, welchem man mit NULL früher begegnete. In C++ wurde hier zwar jetzt foo(int) aufgerufen, aber mit

NULL hat man ja nicht unbedingt einen Ganzzahlwert im Sinn! Das Problem war ganz einfach, dass in C++ NULL als 0 bzw. 0L und in C als (void*)0 implementiert war.

Rohe Zeiger und rohe Arrays

An dieser Stelle sollten Sie vielleicht auch noch die Ähnlichkeiten zwischen den rohen Zeigern und den rohen Arrays kennen, was durchaus sehr verwirrend sein kann. Folgender Codeausschnitt soll Ihnen den Sachverhalt demonstrieren:

```
00 // kapitel04/zeiger02.cpp
...
01 int iarray[10] = { 1, 2, 3, 4 ,5};

02 std::cout << *iarray << std::endl;
03 std::cout << iarray[0] << std::endl;
04 std::cout << *(iarray + 2) << std::endl;
05 std::cout << iarray[2] << std::endl;
```

In Zeile **01** wurde eine rohes C-Style-Array mit zehn Elementen angelegt. In den Zeilen **02** und **03** wird jeweils auf das erste Element im Array zugegriffen und es ausgegeben. Zeile **03** dürfte nichts Neues mehr für Sie darstellen, aber Zeile **02** dürfte den einen oder anderen doch etwas überraschen. Dasselbe geschieht in den Zeilen **04** und **05** nochmals, wo wir auf das dritte Element im rohen Array zugreifen bzw. es hier ausgeben.

Solche Beispiele sind es dann häufig, weshalb rohe Zeiger und rohe Arrays oftmals »gleichgesetzt« werden. Dass `*array` und `array[0]` bzw. `*(array+2)` und `array[2]` denselben Effekt haben, ist, dass ein Zugriff mit dem Indizierungsoperator `[]` letztendlich auch nur vom Compiler bei der Übersetzung in eine Zeigerversion »zerfällt« bzw. umgewandelt wird.

Rohe Zeiger vs. rohe Arrays

Während der Inhalt bei einem rohen C-Style-Array beim Programmstart schon eine feste Adresse enthält, welche nicht mehr verschoben oder in der Größe geändert werden kann, kann der Inhalt eines Zeigers jederzeit geändert werden und auf eine andere Adresse verweisen.

Zum Abschluss will ich Ihnen noch zeigen, wie Sie mit rohen Zeigern durch rohe Arrays iterieren (sie durchlaufen) können:

```
00 // kapitel04/zeiger03.cpp
...
01 int *iptr=0L;
```

```

02 int iarray[5] = { 1, 2, 3, 4 ,5 };
03 iptr = &iarray[0]; // oder: iptr = iarray

04 for(size_t it=0; it < sizeof(iarray)/sizeof(int); ++it) {
05     std::cout << *(iptr + it) << std::endl;
06 }

```

In Zeile **03** erhält der Zeiger `iptr` die Adresse des ersten Elements im Array `iarray`. Alternativ hätten Sie hier auch nur `iptr=iarray` verwenden können. In der nun folgenden Schleife in den Zeilen **04** bis **06** werden jetzt die einzelnen Elemente des Arrays mit dem Zeiger durchlaufen (iteriert) und ausgegeben. Die Klammerung ist nötig, weil diese eine höhere Priorität als das Sternchen hat und somit auch vor dem Indirektionsoperator ausgeführt wird.

Ich will das Thema hier jetzt nicht noch mehr ausrollen, weil es doch zu C-lastig ist und die bessere C++-Alternative hierzu klar die Standardcontainer wie `std::vector` oder `std::array` mit Iteratoren sind.

Hierzu dasselbe Beispiel nochmals mit dem Standardcontainer `std::array`:

```

00 // kapitel04/zeiger04.cpp
01 #include <vector>
...
02 std::vector<int> vec(5);
03 for( int i = 0; i<5; ++i) {
04     vec.at(i) = i+1; // Werte von 1-5 zuweisen
05 }
06 for( std::vector<int>::iterator it=vec.begin();
      it != vec.end(); ++it) {
07     std::cout << *it << std::endl;
08 }

```

In Zeile **06** bis **08** werden Iteratoren verwendet, um den Standardcontainer `vector` zu durchlaufen. Intern stellen Iteratoren letztendlich auch wieder nur eine spezielle Form von Zeigern dar, aber die Verwendung ist hierbei doch schon erheblich einfacher und sicherer. Wen hier die lange Verwendung von `std::vector<int>::iterator` stört, der kann im neuen C++11-Standard einfach stattdessen das Schlüsselwort `auto` davorsetzen. Auf Iteratoren wird noch gesondert in Abschnitt 12.4, »Iteratoren«, etwas umfassender eingegangen.

Wer sich im Beispiel noch etwas näher an den C-Style-Arrays orientieren will, kann hier die neu in C++11 eingeführte Containerklasse `std::array` mit der neuen Range-basierten `for`-Schleife und dem Schlüsselwort `auto` verwenden:

```

00 // kapitel04/zeiger05.cpp
01 #include <array>
...
02 std::array<int, 5> arr={ 1, 2, 3, 4, 5 };
03 for(const auto &it : arr) {
04     std::cout << it << std::endl;
05 }

```

Ich finde, dieses Beispiel demonstriert sehr schön, wie einem die Neuerungen von C++11 das Programmiererleben erheblich erleichtern können.

Speicherplatz zur Laufzeit anfordern (C++03)

Ein weiterer Anwendungszweck von Zeigern ist das dynamische Anfordern von Speicherplatz zur Laufzeit. Mit rohen Arrays mussten Sie zwischen `[N]` eine fixe Größe mit `N` angeben. Wenn Sie aber nicht genau sagen können, wie viele Elemente von den rohen Arrays benötigt werden, können Sie mithilfe eines Zeigers und dem `new`-Operator Speicher zur Laufzeit anfordern. Mit `new` fordern Sie den Speicher an, und mit einem Zeiger müssen Sie diesen Speicher verwalten. Hierzu die Syntax:

```

01 Typ *ptr1 = new Typ;
02 Typ *ptr2 = new Typ[N];

```

Mit dem Beispiel in Zeile **01** fordern Sie vom System (genauer vom Heap-Speicher) so viel Speicher zur Laufzeit an, wie der angegebene `Typ` hinter `new` benötigt. Hierdurch verweist ein Zeiger `ptr1` auf die Anfangsadresse dieses Speicherbereichs. Das Beispiel in Zeile **02** hingegen stellt ein dynamisches Array dar, wo Sie `N`-Elemente vom `Typ` anfordern. Auch hier verweist ein Zeiger `ptr2` auf die Anfangsadresse dieses Speicherblocks. Hierzu ein kurzer einfacher Codeausschnitt, welcher Ihnen ein solches dynamisches Array bei der Ausführung zeigen soll:

```

00 // kapitel04/zeiger06.cpp
...
01 unsigned int n=0;
02 std::cout << "Anzahl der Elemente: ";
03 std::cin >> n;
04 int *iptr = new int[n];
05 for(unsigned int i=0; i < n; ++i ) {
06     *(iptr + i) = i+1;
07 }
08 for(unsigned int i=0; i < n; ++i ) {
09     std::cout << iptr[i] << std::endl;
10 }
...

```

Zunächst werden Sie abgefragt, für wie viele Elemente vom Typ `int` Sie Speicher reservieren wollen. In Zeile **04** reservieren wir dann mit dem `new`-Operator `n`-Elemente vom Typ vom Heap. Der Zeiger `iptr` sollte auf die Anfangsadresse des zurückgegebenen Speicherblocks verweisen. In den Zeilen **05** bis **07** weisen wir den einzelnen Elementen in einer Schleife Werte von 1 bis `n` zu. Statt der Zeigerschreibweise wie in Zeile **06** können Sie hierbei ebenso den Indizierungsoperator verwenden, wie dies in Zeile **09** bei der Ausgabe der einzelnen Elemente demonstriert wird.

Memory Leak (Speicherleck) 1

Hängen Sie jetzt an den eben geschriebenen Code aus Listing *zeiger06.cpp* noch folgende Zeilen an:

```
...
11 int *iptr02 = new int;
12 *iptr02 = 100;
13 iptr = iptr02; // ursprüngliche Adresse verloren
```

In Zeile **13** haben Sie dem Zeiger `iptr` die Adresse des Zeigers `iptr02` zugewiesen. `iptr02` hat zuvor Speicher für ein `int`-Element vom Heap angefordert. In diesem Beispiel verweisen quasi beide Zeiger auf denselben Speicherbereich. Das eigentliche Problem ist jetzt aber, dass Sie niemals mehr Zugriff auf den Speicherbereich haben, den Sie zuvor in Zeile **04** angefordert und mit `iptr` verwaltet haben. Der ehemalige Speicherbereich auf dem Heap ist jetzt quasi verwaist und wird zur Laufzeit des Programms niemals mehr freigegeben. Hiermit haben Sie ein klassisches Speicherleck geschaffen (engl. *Memory Leak*).

Als Alternative würde sich hier wieder der neu in C++11 eingeführte `std::unique_ptr` (siehe Abschnitt 16.3.3, »Unique Pointer«) eignen. Ein solcher Zeiger teilt seine Ressourcen nicht mit anderen Zeigern und würde bei einer Zuweisung wie in Zeile **13** bereits zur Übersetzungszeit mit einer Fehlermeldung abbrechen.

Nicht benötigten Speicher wieder freigeben und Memory Leak 2 (C++03)

Ein weiteres Speicherleck wird häufig mit einem vergessenen `delete` bzw. `delete[]` erzeugt, was ja im Beispiel von Listing *zeiger06.cpp* ebenfalls vernachlässigt wurde. Wenn Sie bspw. immer nur Speicher vom Programm anfordern und nicht mehr benötigten Speicher nicht wieder freigeben, wird das Programm langsam den Arbeitsspeicher füllen und irgendwann nicht mehr reagieren. In den Beispielen in diesem Buch (wie eben in *zeiger06.cpp*) mag dies recht trivial sein, weil die Programme mal schnell ausgeführt werden und der Speicher bei Beendigung des Programms ohnehin automatisch wieder freigegeben wird. Bei Programmen allerdings, die Stunden, Tage oder gar Monate laufen (wie bspw. Server-Anwendungen), ist eine solches Nichtfreigeben von Speicher fatal.

Speicher, den Sie mit `new` angefordert haben und auf dessen Anfangsadresse ein Zeiger verweist, geben Sie wie folgt wieder frei:

```
01 delete iptr02;  
02 delete [] iptr;
```

Mit der ersten Version (Zeile **01**) geben Sie ein einfaches Speicherobjekt frei, auf das der Zeiger `ptr` verweist. Bei der Freigabe in Zeile **02** geben Sie einen zusammenhängenden Speicher (Array) frei, weshalb Sie hier zusätzlich die eckigen Klammern mit angeben müssen. Häufig passiert der Fehler, dass bei einem dynamischen Array die eckigen Klammern vergessen werden und somit nur das erste Element im Array freigegeben wird.

Auch hier gilt, wenn Sie wirklich auf diese Weise ein dynamisches Array erstellen wollen, sollten Sie, wenn möglich, eher auf den neu in C++11 eingeführten Zeiger `std::unique_ptr` zurückgreifen. Bei diesem klugen Zeiger müssen Sie sich nicht mehr um die Freigabe des Speichers mit `delete` bzw. `delete []` kümmern. Das übernimmt `std::unique_ptr` für Sie. Mehr dazu erfahren Sie in Abschnitt 16.3.3, »Unique Pointer«.

Verwenden Sie die Standardcontainer

Da Sie mit `std::vector` oder `std::string` schon kurz einige der vielen Standardcontainer kennengelernt haben, werden Sie bereits ahnen, dass man sich den Aufwand für dynamische Speicherreservierungen mit den Zeigern und `new` bzw. `delete` mittlerweile sparen kann. Wenn Sie nicht wirklich einen speziellen Grund haben, rohe Zeiger zu benutzen, dann empfehle ich Ihnen, die Standardcontainer zu verwenden. Und wenn Sie schon Zeiger für die Speicherreservierung verwenden müssen, dann sollten Sie, wenn möglich, die neu in C++11 eingeführten klugen Zeiger (hier `std::unique_ptr`) verwenden (siehe Abschnitt 16.3, »Smart Pointer (C++11)«). Verwenden Sie die Zeiger nur dann, wenn Sie 100 %ig wissen, was Sie da tun und warum Sie das tun. Wenn Sie nicht sicher sind, ob Sie rohe Zeiger verwenden sollten oder etwas anderes, dann verwenden Sie die Zeiger nicht, weil Sie eben nicht wissen, warum Sie diese verwenden sollten. ☺

Da es sich bei diesem Buch nun mal um ein umfassendes Handbuch handelt und es noch viel alten Code zur Überarbeitung auf der Welt gibt, sollten Sie auch hierüber grundlegend Bescheid wissen. Sie sollten aber jetzt auch wissen, wie man es bei neuen Projekten besser und einfacher machen kann!

Damit Sie sich auch ein Bild davon machen können, wie das Beispiel aus Listing *zeiger06.cpp* mit `std::unique_ptr` realisiert werden kann, finden Sie hierzu einen Codeausschnitt dazu. Das Beispiel funktioniert logischerweise nur mit Compilern, welche den C++11-Standard beherrschen.

```

00 // kapitel04/zeiger07.cpp
01 #include <memory> // für unique_ptr -> C++11
...
02 unsigned int n=0;
03 std::cout << "Anzahl der Elemente: ";
04 std::cin >> n;
05 std::unique_ptr<int[]> iptr(new int[n]);
...
06 for(unsigned int i=0; i < n; ++i ) {
07     iptr[i]= i+1;
08 }
09 for(unsigned int i=0; i < n; ++i ) {
10     std::cout << iptr[i] << std::endl;
11 }

12 std::unique_ptr<int> iptr02(new int);
13 *iptr02=101;
14 iptr = iptr02; // Geht nicht mehr! Fehler!
...

```

In Zeile **05** sehen Sie, wie Sie mit `std::unique_ptr` Speicherplatz für ein rohes `int`-Array anfordern können. Auf die einzelnen Elemente können Sie ganz bequem mit dem `[]`-Operator zugreifen. Um es auch gleich vorausgreifend zu erwähnen, der `std::unique_ptr` hat keinen `->`-Operator. Der Hauptvorteil, ein rohes Array mit `std::unique_ptr` anzufordern, ist ganz klar, dass Sie zum einen den Speicher nicht mehr manuell freigeben müssen und dass zum anderen auch das richtige `delete []` (mit den eckigen Klammern) aufgerufen wird. Ein beliebter Fehler ist es, die eckigen Klammern mal zu vergessen, wenn diese benötigt werden.

Die zweite Verwendung von `unique_ptr` sehen Sie in Zeile **12**, wo wir Speicher für ein einzelnes Element anfordern. Im Grunde nichts Besonderes. Das Hauptaugenmerk liegt hierbei aber auf Zeile **14**, wo wir ja mit gewöhnlichen Zeigern die Anfangsadresse vom reservierten `int`-Array verschlampt und ein Memory Leak erzeugt hatten. Bei `std::unique_ptr` kann uns dies nicht mehr passieren, weil der Compiler hier die Übersetzung mit einem Fehler abbricht. Und sollten Sie trotzdem mal in Verlegenheit kommen, einen Zeiger mit einem anderen Zeiger »teilen« zu müssen, so gibt es seit C++11 auch hier mit `std::shared_ptr` (siehe Abschnitt 16.3.1, »Shared Pointer«) das passende Gegenstück zu rohen Zeigern.

Und wie bereits mehrfach erwähnt und es immer wieder empfohlen wird, sollten für solch einfache Dinge eigentlich überhaupt keine so umständlichen Speicherreservierungen durchgeführt werden, wenn nicht unbedingt notwendig, und stattdessen die Standardcontainer verwendet werden. In diesem Beispiel würde ich Ihnen empfeh-

len, `vector<int>` zu verwenden. Hierzu sehen Sie das Beispiel aus Listing *zeiger06.cpp* nochmals, nur jetzt wesentlich C++-gerechter und sinnvoller mit dem Standardcontainer `std::vector`:

```
00 // kapitel04/zeiger08.cpp
01 #include <vector>
...
02 unsigned int n=0;
03 std::vector<int> varray;
04 std::cout << "Anzahl der Elemente: ";
05 std::cin >> n;
06 for(unsigned int i=0; i < n; ++i ) {
07     varray.push_back(i+1);
08 }
09 for(unsigned int i=0; i < varray.size(); ++i ) {
10     std::cout << varray.at(i) << std::endl;
11 }
```

Da muss nicht mehr viel hinzugefügt werden. Der Vorteil der Verwendung von `std::vector` in diesem Fall ist ersichtlich. Sie müssen sich weder Gedanken um die Speicherreservierung machen noch um irgendwelche Zeiger, die Chaos verursachen können, und auch Speicherlecks brauchen Sie nicht zu fürchten.

4.4 Referenzen

Die Referenzen werden für Sie an dieser Stelle noch nicht allzu viel Sinn ergeben. Aber spätestens wenn Sie Funktionen oder Methoden für benutzerdefinierte Typen (wie Klassen) schreiben, werden Sie den Vorteil der Referenzen gegenüber Zeigern zu schätzen wissen. Idealerweise bevorzugen Sie auch hier die Referenzen gegenüber rohen Zeigern bei den Parametern.

Referenzen sind vom Grundprinzip her nichts anderes als Aliase (ein anderer Name) für ein Speicherobjekt. Beim Anlegen einer solchen Referenz muss diese daher auch schon mit einem bereits bestehenden Speicherobjekt initialisiert werden. Referenzen erkennen Sie daran, dass bei der Deklaration nach dem Typ und vor dem Bezeichner das `&` steht. Der Unterschied zwischen Referenzen und Zeigern ist der, dass eine Referenz nur ein Alias auf ein anderes Speicherobjekt ist. Zeiger hingegen können Adressen von anderen Speicherobjekten aufnehmen.

Folgender kurzer Codeausschnitt soll Ihnen eine solche Referenz demonstrieren:

```

00 // kapitel04/referenzen01.cpp
01 #include <iostream>
...
02 int ivar = 123456;
03 int &ref_int = ivar;

04 std::cout << ivar << std::endl;
05 std::cout << ref_int << std::endl;
06 ref_int = 654321;
07 std::cout << ivar << std::endl;
08 std::cout << ref_int << std::endl;
09 std::cout << &ivar << std::endl;
10 std::cout << &ref_int << std::endl;

```

Die Initialisierung von `ivar` an die Referenz finden Sie in Zeile **03** vor. Auch hier sollten Sie noch wissen, dass Sie Referenzen nur einmal initialisieren können. In diesem Beispiel haben wir in Zeile **03** mit `ref_int` ein Synonym für `ivar` angelegt, welches zur Laufzeit des Programms nicht mehr geändert werden kann. Weisen Sie der Referenz trotzdem etwas zu, dann hätte dies nur den Effekt, dass der Wert von `ivar` geändert würde. Die Zuweisung über das Synonym `ref_int` in Zeile **06** führt dazu, dass der Wert von `ivar` geändert wird, wie die Ausgabe in den Zeilen dahinter auch beweist.

Im Gegensatz zu den Zeigern erfolgt die Übergabe eines neuen Wertes, wie in Zeile **06** geschehen, wie bei einer gewöhnlichen Variablen. Umständliche Zuweisungen wie mit dem Indirektionsoperator `*` bei den Zeigern entfallen bei den Referenzen.

Dass Referenzen wirklich nur Synonyme für das eigentliche Ziel sind, beweist die Ausgabe der Zeilen **09** und **10**, welche beide die Adresse von `ivar` zurückgeben. Daher gibt es auch keine Möglichkeit, die Adresse einer Referenz zu ermitteln.

Wie bereits zu Beginn des Abschnitts erwähnt, machen die Referenzen erst wirklich Sinn, wenn Sie diese als Parametertypen oder Rückgabewerte von Funktionen verwenden. Mehr dazu erfahren Sie u.a. in Abschnitt 5.5.2, »Funktionsparameter als Referenz (Call by Reference)«.

Merke

Wenn Sie eine Referenz mit einem neuen Speicherobjekt initialisieren, wird kein neues Speicherobjekt angelegt, sondern nur ein weiterer Name für ein schon vorhandenes Speicherobjekt!

4.5 Strukturen

Mit den Strukturen können Sie unterschiedliche Variablen verschiedener Datentypen zusammenfassen und verwenden. Dies kann bspw. sehr nützlich sein, wenn Sie eine Adressverwaltung entwickeln wollen. Folgendermaßen können Sie unterschiedliche Typen in einer Struktur zusammenfassen und definieren:

```
01 struct adressen {
02     std::string nachname;
03     unsigned int plz;
04     std::string ort;
05     unsigned int tel;
06 };
```

Mit dem Schlüsselwort `struct` und den sich öffnenden geschweiften Klammern (Zeile **01**) leiten Sie die Zusammenfassung (oder auch den Verbund) unterschiedlicher Typen mit dem Bezeichner (hier `adressen`) ein. Anschließend folgen die einzelnen Strukturmitglieder (auch Member genannt oder einfach zu Deutsch Mitglied(er)), in denen die eigentlichen Daten gespeichert werden (hier mit den Zeilen **02** bis **05**). Am Ende wird dieser Verbund von Datentypen wieder mit einer schließenden geschweiften Klammer und (ganz wichtig) einem Semikolon beendet.

In der Regel wird eine solche Definition außerhalb einer (`main`-)Funktion an einem globalen Ort (bspw. in einer Headerdatei) festgelegt, damit dieser Datentyp in allen Funktionen möglichst global zur Verfügung steht. Natürlich können (und sollten) Sie eine Struktur vielleicht auch in einem eigenen Namensraum definieren, um sich so vor Namenskonflikten zu schützen. Aber eine weitere Erläuterung hierzu geht an dieser Stelle zu weit. Selbstverständlich ist es nicht verboten, eine Struktur innerhalb einer Funktion zu definieren, aber dann ist die Struktur auch nur innerhalb dieser Funktion gültig. Auf Dinge wie Namensräume, globale und lokale Zugriffe und mehr wird noch gesondert in Kapitel 6, »Modularisierung«, eingegangen.

Im neuen C++11-Standard können Sie den einzelnen Strukturmitgliedern gleich einen Wert bei der Strukturdefinition übergeben:

```
01 struct adressen {
02     std::string nachname{""}; // C++11
03     unsigned int plz{0};      // C++11
04     std::string ort{""};      // C++11
05     unsigned int tel{0};      // C++11
06 };
```

Die Initialisierung in den Zeilen **02** und **04** von `std::string` kann man sich ja sparen, weil der Standardcontainer dafür sorgt, dass `std::string` automatisch mit einem lee-

ren String initialisiert wurde. Aber die Initialisierungen in den Zeilen 03 und 05 sind sehr nützlich, weil damit sichergestellt wird, dass nicht mit nichtinitialisierten Werten gearbeitet wird. Aber wie gesagt, dieses Feature funktioniert erst mit dem C++11-Standard.

Strukturen in Strukturen

Hier sollte noch hinzugefügt werden, dass eine Struktur selbstverständlich auch andere Strukturen als Strukturmitglied enthalten darf.

Sie könnten in einer C++-Struktur als Member (Mitglied) durchaus auch Methoden (Elementfunktionen oder Memberfunktionen) und andere klassentypische Sachen verwenden, aber in der Praxis wird hierfür eher selten eine Struktur (`struct`) verwendet, sondern eine Klasse mit dem Schlüsselwort `class`.

Strukturen(`struct`) und Klassen (`class`)

An dieser Stelle sollte auch gleich die Information erfolgen, dass eine Struktur im Prinzip auch nur eine Klasse ist, allerdings mit dem bedeutenden Unterschied, dass bei einer Struktur alle Elemente standardmäßig öffentlich (`public`) sind und somit beliebig von außen verwendet werden können. Eine Klasse hingegen erlaubt von Haus aus keinen Zugriff von außen auf die einzelnen darin enthaltenen Mitglieder. Hier muss man Mitglieder explizit mit dem Schlüsselwort `public` öffentlich machen oder mit `private` in der Klasse einschließen. Etwas `private` zu machen, bedeutet letztendlich, dass nur innerhalb der Klasse auf die Mitglieder zugegriffen werden kann. Solche Mitglieder (bzw. auch Member) können Methoden (auch Elementfunktionen oder Memberfunktionen genannt) oder Daten (häufig als Eigenschaften oder Attribute bezeichnet) sein. Allerdings müssen Sie sich jetzt noch keine Gedanken um `public` oder `private` oder Mitglieder machen. Die Klassen werden noch gesondert in Kapitel 7, »Grundlagen zu den Klassen«, des Buches behandelt. Trotzdem macht es gerade daher Sinn, dass Sie sich hier schon mit den Strukturen vertraut machen, weil Sie dabei vieles lernen (bspw. Zugriff auf die Mitglieder), was dann auch später bei den Klassen gültig ist.

Strukturvariable anlegen

Wenn Sie eine Struktur definiert haben, können Sie einen neuen Typ davon, wie gewöhnliche Variablen oder Arrays, erzeugen. Hierzu einige Möglichkeiten:

```
01  adressen eineAdresse;
02  adresse *adressenPtr=0L;
03  adressen rawArrayAdressen[10];
04  std::vector<adressen> vecAdressen;
```

In Zeile **01** haben Sie einen neuen Adressensatz mit dem Bezeichner `eineAdresse` angelegt, womit Sie eine komplette Adresse mit Nachname, Postleitzahl, Ort und Telefonnummer speichern können. Zeile **02** zeigt einen rohen Zeiger, welcher auf eine Struktur vom Typ `adresse` verweisen kann und womit dann auch dynamische Speicher für `adressen`-Elemente zur Laufzeit angefordert werden könnte. Natürlich können Sie hierbei auch ein rohes C-Style-Array anlegen, wie dies in Zeile **03** gezeigt wird, wo ein rohes C-Style-Array mit Platz für zehn Adressensätze vom Typ `adressen` angelegt, aber noch nicht mit gültigen Werten initialisiert wurde. Eine fixe Größe für Adressen ist in der Regel Unsinn und eine dynamische Speicherreservierung über rohe Zeiger zu umständlich. Daher empfiehlt sich hierfür auch in der Praxis, die fertigen Sachen der Standardcontainer zu verwenden, wie Sie es hier in Zeile **04** mit `vector` sehen. Ich würde Ihnen allerdings für die Verwaltung von Adressensätzen zu einer verketteten Liste raten, wofür mit `std::list` ebenfalls ein Standardcontainer vorhanden ist.

Zugriff auf die einzelnen Elemente einer Strukturvariablen

Um von außen auf die einzelnen Strukturmitglieder zuzugreifen, benötigen Sie den Punktoperator. Hierbei schreiben Sie zunächst den Bezeichner für die Struktur, den Sie beim Anlegen verwendet haben. Dann folgt der Punktoperator und zum Schluss das entsprechende Strukturmitglied, auf das Sie zugreifen wollen. Ansonsten funktioniert dann alles, wie Sie es schon von den Basisdatentypen, rohen Arrays oder von `std::vector` her kennen. Hier die Syntax:

Bezeichner.Strukturmitglied

Hierzu ein paar Beispiele, welche Ihnen den Zugriff auf einzelne Strukturmitglieder von außen demonstrieren sollen:

```
00 // kapitel04/struct01.cpp
...
01 struct adressen {
02     std::string nachname;
03     unsigned int plz;
04     std::string ort;
05     unsigned int tel;
06 };

07 int main() {
08     adressen eineAdresse;
09     eineAdresse.nachname = "Mustermann";
10     eineAdresse.plz = 89000;
11     getline(std::cin, eineAdresse.ort);
12     std::cin >> eineAdresse.tel;
```

```

13     std::cout << eineAdresse.nachname << std::endl;
14     std::cout << eineAdresse.plz << std::endl;
15     std::cout << eineAdresse.ort << std::endl;
16     std::cout << eineAdresse.tel << std::endl;
...
17 }

```

Hier sehen Sie einige Beispiele, wie Sie einem Strukturmitglied der Strukturvariablen `eineAdresse` (Zeile 08) Werte übergeben können. In den Zeilen 09 und 10 bekommen die Strukturmitglieder ihren Wert über den Zuweisungsoperator. Natürlich funktioniert das Ganze wie gewohnt auch interaktiv mit `getline()` (Zeile 11) und `std::cin` (Zeile 12). Ebenso trivial funktioniert hier auch die Ausgabe mit `std::cout`, wie Sie dies in den Zeilen 13 bis 16 sehen können.

Es ist auch möglich, eine Struktur direkt zu initialisieren. Hierfür müssen Sie lediglich die einzelnen Werte in der richtigen Reihenfolge, getrennt von einem Komma, und zwischen geschweiften Klammern zuweisen:

```
adressen eineAdresse = {"Mustermann", 89000, "München", 882211};
```

Zugriff auf die einzelnen Elemente in einem rohen Strukturvariablen-Array

Ähnlich einfach funktioniert dies mit rohen Struktur-Arrays, nur dass Sie hier noch den `[]`-Operator mit dem richtigen Index hinter den Bezeichner stellen müssen:

```

00 // kapitel04/struct02.cpp
...
01 adressen rawArrayAdressen[10];
02 rawArrayAdressen[0].nachname = "Mustermann";
03 rawArrayAdressen[0].plz = 89000;
04 std::getline(std::cin, rawArrayAdressen[0].ort);
05 std::cin >> rawArrayAdressen[0].tel;
...

```

Wichtig ist hierbei, dass Sie den Indizierungsoperator `[]` an der richtigen Stelle platzieren und **nicht** aus Versehen bspw. `rawArrayAdressen.nachname[0]` schreiben. Ansonsten gilt auch hier alles, was Sie schon von den rohen Arrays her kennen. Auch eine direkte Initialisierung lässt sich mit rohen Arrays einer Struktur realisieren:

```

adressen eineAdresse[5] = {
    { "Mustermann", 89000, "München", 882211 },
    { "Musterfrau", 89000, "München", 112288 }
};

```

Hier wurden in einer Initialisierungsliste die ersten beiden (von fünf) Strukturvariablen direkt mit einem Wert initialisiert. Die restlichen drei Strukturvariablen im Array enthalten noch ungültige Werte, mal abgesehen von den `std::string`-Elementen, die ja standardmäßig mit einem leeren `""` initialisiert werden.

Wie allerdings schon des Öfteren erwähnt, sind die rohen alten Sachen eher eine schlechte Wahl und nicht sehr C++-like. In diesem Beispiel empfehle ich Ihnen wieder, Standardcontainer wie `std::vector` oder in diesem Fall besser noch `std::list` zu verwenden. Ein Beispiel dazu finden Sie ein paar Seiten später im Unterabschnitt »Dynamische Datenstrukturen – verkettete Listen«.

Zugriff auf die einzelnen Elemente über einen Standardcontainer

Am idealsten für die Verwendung von Strukturen mit einem Adressensatz dürften wohl Standardcontainer sein. Hierbei sind Sie an keine fixe Größe gebunden und können den Adressensatz beliebig erweitern, ohne komplexen und fehleranfälligen Code schreiben zu müssen. Hier ein Codeausschnitt, wie Sie den Standardcontainer `std::vector` mit der Strukturvariablen `adressen` verwenden können:

```
00 // kapitel04/struct03.cpp
01 #include <vector>
...
02 std::vector<adressen> vecAdressen(10);

03 vecAdressen.at(0).nachname = "Mustermann";
04 vecAdressen.at(0).plz = 89000;
05 getline(std::cin, vecAdressen.at(0).ort);
06 std::cin >> vecAdressen.at(0).tel;
...
07 std::cout << vecAdressen.at(0).nachname << std::endl;
...
08 adressen eineAdresse;
09 eineAdresse.nachname = "Musterfrau";
10 eineAdresse.plz = 89111;
11 eineAdresse.ort = "Starnberg";
12 eineAdresse.tel = 123321;

13 vecAdressen.push_back(eineAdresse);
```

In Zeile 02 legen wir einen `std::vector` für zehn Strukturen vom Typ `adressen` an. Hier müssen Sie ja nicht unbedingt angeben, wie viele Elemente Sie hinzufügen wollen. In den Zeilen 03 bis 06 sehen Sie dann, wie Sie für das erste Element in `vector<adressen>` die Daten der einzelnen Strukturmitglieder hinzufügen können. Im

Grunde alles wie schon bei einem rohen Array, nur dass wir hier die Elementfunktion `at()` statt des Indizierungsoperators verwenden (was Sie hier auch tun könnten).

Selbiges gilt auch für die Ausgabe, was in Zeile 07 nur mit dem Nachnamen abgedruckt wurde. In den Zeilen 08 bis 12 weisen Sie einer Strukturvariablen und deren Mitgliedern einzelne Werte zu und fügen diese Strukturvariable mit `push_back()` am Ende von `vecAdressen` hinzu (Zeile 13) – womit das neu hinzugefügte Element jetzt das elfte wäre und die Größe von `vecAdressen` dynamisch um ein Element auf 11 erweitert wurde.

Auch eine direkte Initialisierung lässt sich dank der vereinheitlichten Initialisierungssyntax von C++11 jetzt auch mit `std::vector` realisieren:

```
// nur mit C++11 möglich:
std::vector<adressen> vecAdressen {
    {"Mustermann", 88000, "Musterdorf", 123456},
    {"Musterfrau", 66011, "Musterort", 654321}
};
```

Roher Strukturzeiger

Natürlich können Sie auch einen rohen (Struktur-)Zeiger verwenden, um auf die einzelnen Mitglieder einer Strukturvariablen zuzugreifen. Sie sehen hier zwar kurz die Verwendung solcher Strukturzeiger, aber in der allgemeinen Praxis gibt es genügend bessere Alternativen dazu. Zur Verwendung eines rohen Strukturzeigers können Sie zwar ebenfalls noch den Punktoperator verwenden, aber der Zugriff ist dann mit `(*struktur_zeiger).struktur_mitglied` ziemlich umständlich. Daher finden Sie zur Vereinfachung einen Pfeiloperator `->`, womit der Zugriff über `struktur_zeiger->struktur_mitglied` wesentlich lesbarer wird. Hierzu ein einfaches Listing, welches die Verwendung des Strukturzeigers in der Praxis demonstriert:

```
00 // kapitel04/struct04.cpp
...
01 adressen myadr;
02 adressen *addrPtr;

03 addrPtr = &myadr;
04 addrPtr->nachname = "Mustermann";
05 addrPtr->plz = 88001;
06 addrPtr->ort = "Musterort";
07 addrPtr->tel = 123456;

08 std::cout << addrPtr->nachname << std::endl;
...
```


In Zeile **02** erhält der Zeiger `addrPtr` die Adresse von `myadr`. In den Zeilen **04** bis **07** initialisieren wir dann indirekt über den Zeiger `addrPtr` die Werte der Strukturvariablen `myadr`. Der lesende Zugriff funktioniert genauso, wie Sie in Zeile **08** mit der Ausgabe über `std::cout` sehen können.

Dieser indirekte Zugriff über einen Strukturzeiger kann bspw. verwendet werden, um den Inhalt einer Strukturvariablen an eine Funktion zu übergeben und dort zu bearbeiten. Statt einer kompletten Kopie der Strukturvariablen wird an die Funktion nur die Anfangsadresse davon übergeben. In C++ verwendet man allerdings die speziell für solche Zwecke eingeführten Referenzen (siehe Abschnitt 4.4, »Referenzen«).

Ein weiteres Anwendungsgebiet ist auch wieder die dynamische Speicherreservierung von weiteren Strukturvariablen zur Laufzeit (hierfür heißt die bessere Alternative wieder Standardcontainer), und dann wären da noch komplexe Datenstrukturen wie verkettete Listen, bei denen ein Strukturzeiger auf das nächste oder vorherige Element verweist. Aber auch hier sollten Sie wieder auf die Standardcontainer zurückgreifen, weil dort verkettete Listen bereits mit `std::forward_list` (C++11) implementiert sind.

Strukturen vergleichen

Ein direktes Vergleichen zweier Strukturen mit `==` oder `!=` ist nicht möglich, da der Compiler nicht weiß, wie er vergleichen soll. Aus Optimierungsgründen findet man hier häufig leere Füllbytes. Wenn Sie zwei Strukturen miteinander vergleichen wollen, müssen Sie die entsprechenden Operatoren überladen. Mehr dazu erfahren Sie in Kapitel 8, »Operatoren überladen«.

Dynamische Datenstrukturen – verkettete Liste

An dieser Stelle soll auch noch kurz das Thema verkettete Listen angesprochen werden. Die verketteten Listen werden (neben binären Bäumen) oft als Grundlagen von Lesungen in der Informatik verwendet. Ich will Sie an dieser Stelle auch kurz darüber informieren, wie solche Listen implementiert werden. Die gute Nachricht gleich vorweg: In C++ müssen Sie solche rohen Listen nicht mehr von Hand implementieren. Hier stehen Ihnen die Standardcontainer `std::list` oder `std::forward_list` (neu seit C++11) zur Verfügung – und wenn Sie wirklich unbedingt eine verkettete Liste erstellen müssen, dann sollten Sie es C++-typisch mit Klassen (`class`) realisieren.

Verkettete Listen werden mit gewöhnlichen Strukturen definiert. Nur enthalten diese Strukturen neben den Daten zusätzlich noch Zeiger vom selben Typ wie die Struktur selbst. Ein Codeausschnitt dazu:

```
01 struct adressen {
02     std::string nachname;
03     unsigned int plz;
```

```

04  std::string ort;
05  unsigned int tel;
06  struct adressen *next;
07  };
// Zeiger auf das erste Element (Aufhänger)
08  adressen *anfang = 0L;

```

In Zeile **06** der Struktur finden Sie jetzt einen Zeiger, welcher eine Adresse vom selben Typ wie die Struktur speichern kann. Damit wird die Struktur verkettet. Sie hängen quasi eine Struktur an die andere. Der Zeiger `next` verweist immer auf das nächste Element der Struktur und dient auch dazu, die verkettete Liste zu durchlaufen. Zusätzlich benötigt man einen Anfang oder auch Aufhänger für die verkettete Liste, was hier mit Zeile **08** implementiert ist, welche dann immer auf das erste Element in der verketteten Liste zeigt. Ebenso wäre es sinnvoll, einen Zeiger auf das letzte Element der Liste zu verwenden – oder einfach die Liste mit dem Nullzeiger abzuschließen (Abbildung 4.3). Jedes Element in einer Liste mit einem Zeiger auf den Nachfolger wird als Knoten (engl. *node*) bezeichnet.

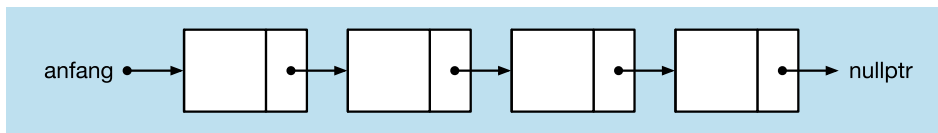


Abbildung 4.3 Die Datenstruktur einer einfach verketteten Liste

Doppelt verkettete Liste

Neben der einfach verketteten Liste können Sie auch noch eine doppelt verkettete Liste erstellen. Hierzu müssen Sie lediglich in der Struktur noch einen weiteren Zeiger verwenden, welcher immer auf das Vorgängerelement verweist. Mit doppelt verketteten Listen können Sie quasi eine Liste von vorne nach hinten und von hinten nach vorne durchlaufen.

Allerdings bedeutet ein weiterer Zeiger auch mehr Verwaltungsaufwand, um die Liste konsistent zu halten, weil jetzt neben einer Adresse auf das Nachfolgerelement auch noch auf eine Adresse zum Vorgängerelement geachtet werden muss.

In der Praxis werden solche Listen als dynamisches Modell erstellt, wo für jedes neue Element, welches Sie in der Liste hinzufügen, auch Speicher zur Laufzeit vom Heap angefordert wird. Und beim Entfernen eines Elements müssen Sie den Speicher für das Element auch wieder freigeben. Besonders wichtig beim Hinzufügen und Entfernen von Elementen ist es, dass der Zeiger auf das nächste Element, hierbei auf ein gültiges Element, verweist – oder einfacher, richtig miteinander verkettet wird/bleibt. Verlieren Sie den »Faden« der verketteten Liste, indem ein `next`-Zeiger auf

einen ungültigen Speicherbereich verweist, ist die Liste abgerissen, und die Daten sind verloren.

Keine Sorge, ich werde Ihnen jetzt nicht den C-Style-Weg zeigen, eine verkettete Liste zu schreiben. Stattdessen finden Sie jetzt ein Beispiel, eine solche Liste C++-like mit dem Standardcontainer `std::list` zu schreiben. Hier ein ausnahmsweise etwas umfangreicheres Beispiel dazu, in dem die einzelnen Elemente lexikografisch sortiert eingefügt werden.

```

00 // kapitel04/struct05.cpp
01 #include <list> // für den Standardcontainer list
02 #include <string>
...
03 struct adressen {
04     std::string nachname;
05     unsigned int plz;
06     std::string ort;
07     unsigned int tel;
08 };
...
09 std::list<adressen> myAdressen;
10 adressen tmp;
11 int wahl;
12 do {
13     std::cout << "-1- Adresse hinzufügen\n";
14     std::cout << "-2- Adressen ausgeben\n";
15     std::cout << "-3- Beenden\n";
16     std::cout << "Ihre Wahl : ";
17     std::cin >> wahl;
18     switch( wahl ) {
19         case 1:
20             std::cout << "Nachname: "; std::cin >> tmp.nachname;
21             std::cout << "Postleitzahl : "; std::cin >> tmp.plz;
22             std::cout << "Ort          : "; std::cin >> tmp.ort;
23             std::cout << "Telefonnr.   : "; std::cin >> tmp.tel;
24             if(myAdressen.empty()) {
25                 myAdressen.push_back(tmp);
26             }
27             else {
28                 if(tmp.nachname > myAdressen.back().nachname) {
29                     myAdressen.push_back(tmp);
30                     break;
31                 }

```

```

32 // C++11: auto für std::list<adressen>::iterator verwendet
33     for(auto it=myAdressen.begin();
           it!=myAdressen.end();++it){
34         if(tmp.nachname <= it->nachname ) {
35             myAdressen.insert(it, tmp);
36             break;
37         }
38     }
39 }
40 break;
41 case 2:
42 // C++11: auto für std::list<adressen>::iterator verwendet
43     for (auto it=myAdressen.begin();
           it!=myAdressen.end();++it) {
44         std::cout << it->nachname << std::endl;
45     }
46     break;
47 } // Ende switch
48 } while(wahl != 3);

```

Hilfe: Eingaben werden einfach übersprungen

Wenn bei Ihnen die Eingabe von `cin` »übersprungen« und einfach mit der übernächsten Eingabe fortgefahren wird, dann liegt dies am übrig gebliebenen  im Tastaturpuffer. Hier können Sie einen Trick ausprobieren, indem Sie hinter dem Einlesen von `cin` den Befehl

```
cin.ignore();
```

hinschreiben. Allerdings funktioniert dies so auch nicht mit jedem Compiler. Manche Compiler akzeptieren dann auch ein

```
cin.get();
```

statt `ignore()`, um ein einzelnes Zeichen (hier das ) aus dem Tastenpuffer zu ziehen.

In Zeile 09 legen Sie eine leere Liste `myAdresse` für Strukturen vom Typ `adressen` an. Der weitere Programmablauf wurde in eine `do-while`-Schleife gepackt. Zunächst folgt die Abfrage, was Sie tun wollen (Adressen hinzufügen, ausgeben oder Programm beenden). Wenn Sie der Liste ein neues Element hinzufügen wollen, geben Sie zunächst die einzelnen Daten für die Struktur in den Zeilen 20 bis 23 ein. Hierfür verwenden wir eine temporäre Strukturvariable. In der Praxis sollten Sie die Eingaben noch prüfen.

Der erste Fall, den es zu überprüfen gilt, ist, ob die Liste leer ist. Ist dies der Fall, können Sie mit einem einfachen `push_back()` die eben eingelesene Strukturvariable am Ende der Liste hinzufügen, wie dies in den Zeilen 24 bis 26 geschieht.

Ist die Liste nicht leer, prüfen wir in der `else`-Verzweigung zunächst, ob der Nachname unseres neuen Elements lexikografisch größer ist als das letzte Element in der Liste, auf welches Sie mit der Elementfunktion `back()` zugreifen können. `std::string` hat die Vergleichsoperatoren von Haus aus implementiert, und daher funktioniert der logische Vergleich von `std::string > std::string` in Zeile 28 sehr gut. Ist also der Nachname des neuen Elements größer als das letzte Element, können wir dies ebenfalls am Ende der Liste mit `push_back()` anhängen.

Die letzte Möglichkeit ist jetzt, dass Sie das neue Element irgendwo dazwischen einfügen müssen. Hierzu verwenden wir einfach eine `for`-Schleife (Zeilen 33 bis 37), durchlaufen Element für Element der Liste und prüfen auch hier lexikografisch, ob unser neues Element kleiner als oder gleich (Zeile 34) dem Element der Liste ist, auf das unser Iterator gerade verweist. Trifft diese Bedingung zu, haben Sie die Position gefunden, an der unser neues Element mit der Elementfunktion `insert()` (Zeile 35) eingefügt werden kann.

Die Ausgabe aller Elemente wurde in den Zeilen 42 bis 45 in eine `for`-Schleife gepackt, wo wieder ein Iterator alle Element durchläuft und im abgedruckten Beispiel nur den Nachnamen ausgibt, zur Demonstration, dass die Liste sauber danach sortiert wurde.

Gerade für Einsteiger ist ein solches Beispiel auf den ersten Blick etwas abstrakt, weil man hier viele Vorgänge nicht direkt »sieht«. So geschieht bspw. die Speicherreservierung beim Hinzufügen neuer Elemente mit `push_back()` oder `insert()` ganz *automatisch* hinter den Kulissen. Auch beim Anfügen oder Einfügen kann man die Verkettung der Liste kaum kaputt machen. Dank der Iteratoren wird die komplexe Zeigerbehandlung versteckt und eine einfach zu verwendende Schnittstelle angeboten. Und wenn Sie eine weitere Funktion zum Entfernen von Elementen mit bspw. der Elementfunktion `remove()` verwenden wollen, müssten Sie sich auch nicht um Speicherlecks Ihre Gedanken machen, weil auch hier alles hinter den Kulissen automatisch abläuft.

Deklaration und Definition

Zum Schluss müssen nochmals kurz ein paar Formalitäten zur Deklaration und Definition von Strukturen und Strukturvariablen geklärt werden. Eine reine Deklaration einer Struktur erfolgt ganz einfach über das Schlüsselwort `struct`:

```
// Deklaration einer Struktur
struct eineStruktur;
```

Bei einer Definition einer Struktur müssen alle Mitglieder der Struktur zwischen den geschweiften Klammern deklariert werden:

```
// Definition einer Struktur
struct eineStruktur {
    // Hier werden die Mitglieder der Struktur deklariert
};
```

In C++11 könnten Sie die Mitglieder der Struktur auch gleich bei der Definition der Struktur initialisieren. In C++03 war dies noch nicht direkt möglich. Wenn Sie eine Strukturvariable deklarieren wollen, müssen Sie die Struktur zuvor definiert haben, damit der Compiler weiß, wie viel Speicher er dafür benötigt. Eine Strukturvariable können Sie dann wie folgt definieren:

```
// Strukturvariablen deklarieren
eineStruktur einBezeichner;
```

Es ist auch möglich, gleichzeitig eine Struktur zu definieren und dazu gleich die Strukturvariable zu deklarieren:

```
// Definition einer Struktur und Deklaration einer Strukturvariablen
struct eineStruktur {
    // Hier werden die Mitglieder der Struktur deklariert
} einBezeichner;
```

4.6 Unions

Ähnlich wie eine Struktur ist eine Union auch ein Verbund verschiedener Datentypen. Abgesehen von den Schlüsselwörtern `union` und `struct` besteht auf den ersten Blick kein Unterschied zwischen Unions und Strukturen. Auch die Anwendung und der Zugriff auf die Union-Mitglieder erfolgt wie bei den Strukturen. Allerdings beim Speicherplatz gehen beide dann doch auseinander. Unions reservieren im Grunde nur Speicher für das größte Element (Mitglied) im Verbund. Das bedeutet auch, dass die einzelnen Mitglieder bei einer Union überlappend angeordnet werden – was gleichzeitig auch weniger Speicherbedarf bedeutet.

Sind Unions böse ...?

In gewisser Hinsicht wurden und werden Unions immer noch vermieden in C++, weil es in der gewöhnlichen Anwendungsprogrammierung kaum Gründe gibt, diese anstelle von `struct` oder `class` zu verwenden. Wenn Sie bei einschlägigen Seiten, wie bspw. <http://stackoverflow.com>, suchen, werden Sie das eine oder andere sinnvolle Beispiel (bspw. Discriminated Unions) für Unions finden. Es wird aber eben genauso viele Personen geben, die davon abraten.

Da C++ im Augenblick auch in der Embedded-Welt anzutreffen ist und mancherorts dort C ablöst, kommt es dort auch wieder auf den Speicher an. Auch wer auf einer niedrigen Ebene mit einzelnen Bytes arbeiten muss, dem können Unions recht hilfreich sein. Und dass mit dem C++11-Standard die Unions ein wenig aufgebohrt wurden, zeigt, dass diese nicht ganz tot sind. Ich werde hier allerdings nicht weiter auf die Pros und vielen Contras eingehen, sondern ganz nüchtern die Unions beschreiben. Sie wissen dann zumindest, was Unions sind, und können im extremen Fall darauf zugreifen.

In der Praxis bedeutet diese Überlappung dann allerdings auch, dass Sie nur ein Mitglied einer Union verwenden können, weil jedes Mitglied dieselbe Anfangsadresse besitzt. Ein einfaches Beispiel dazu:

```
00 // kapitel04/union01.cpp
...
01 struct RGB {
02     unsigned short red, green, blue;
03 };

04 union Color {
05     RGB rgb;
06     unsigned int hexRGB;
07 };
...
08 Color color01, color02;
09 color01.rgb.red = 0;
10 color01.rgb.green = 255;
11 color01.rgb.blue = 255;
12 color02.hexRGB = 0x00ffff;
```

In diesem Beispiel wird in den Zeilen **01** bis **03** zunächst eine Struktur definiert, wo Farbwerte für Rot, Grün und Blau gespeichert werden können. In den Zeilen **04** bis **07** definieren wir dann eine Union mit der Struktur `RGB` und dem Wert `unsigned int` für den Farbwert als Hexadezimalzahl. In der Praxis können Sie jetzt nur einen der beiden Speicherbereiche der Union verwenden, also entweder `rgb` als Struktur oder `hexRGB` als `unsigned int`.

Leider ist es bei den Unions trotzdem möglich, auf andere Mitglieder zuzugreifen. So können Sie bspw. an `color02` trotzdem noch `color.rgb.red=255` oder ähnlich zuweisen und den ursprünglichen Wert `color.hexRGB` überschreiben. Dass dies funktioniert, liegt daran, dass die Unions nur rohe Daten speichern. Und wenn Sie den Begriff »roh« in C++ hören, können Sie sich denken, dass es sich hierbei mehr um einen C-Style handeln muss. Sie programmieren damit quasi auf einer sehr niedrigen Ebene.

Somit benötigen Sie Unions in C++ in den meisten Fällen nicht wirklich. Der einzige wirkliche Vorteil »könnte« sein, wenn Sie Unions bei maschinennaher Programmierung verwenden und extrem an Speicher sparen müssen – allerdings »könnte« man dies auch C++-like mithilfe abgeleiteter Klassen machen.

Wenn Sie Unions mit eigenen Klassen verwenden, kann es zu Problemen beim Erzeugen und Zerstören von Unions kommen, weil der Compiler beim Freigeben nicht weiß, was er freigeben soll – und hier ist es dann wirklich »böse«, die Unions einzusetzen.

Unbeschränkte Unions (C++11)

Da Unions ziemlich C-Style-like sind, war es nicht verwunderlich, dass mit C++11 die Unions erweitert und mehr C++-like gemacht wurden. Mit C++11 können jetzt auch Mitglieder enthalten sein, deren Typen einen nicht trivialen Konstruktor enthalten. So ist es jetzt bspw. auch möglich, `std::string` in Unions zu verwenden. Ebenso erlaubt C++11 jetzt auch klassenübliche Dinge, etwa Konstruktoren und Destruktoren zu definieren.

Aber um ein Klassenobjekt (wie bspw. `std::string` oder auch eine eigene Klasse) in einer Union verwenden zu können, sind ein paar Vorkehrungen zu treffen. Als **Einsteiger** können Sie diesen Abschnitt **überspringen**, weil hier auf klassentypische Themen vorgegriffen wird und weil Unions, auch wenn diese in C++11 klassenähnlicher geworden sind, **nicht** die erste Wahl sind, wenn es um die objektorientierte Programmierung geht.

Auch wenn Sie ein Klassenobjekt als Mitglied für eine C++11-Union verwenden können, lässt sich nicht einfach Folgendes schreiben:

```
01 union Color {
02     unsigned int hexRGB;
03     std::string stringRGB; // C++11
04 };
```

Die Klasse `std::string` definiert in C++11 alle sechs nötigen Funktionalitäten (Standardkonstruktor, Kopierkonstruktor, Zuweisungsoperator, Move-Konstruktor, Move-Zuweisungsoperator und den Destruktor), und daher sind all diese Funktionalitäten bei `Color` als `deleted` markiert. Das bedeutet, Sie können keine Instanz von `Color` anlegen, solange Sie nicht die nötigen Funktionalitäten auch für `Color` explizit selbst implementieren. Im einfachsten Fall würden der Konstruktor und Destruktor ausreichen:

```
00 // kapitel04/union02.cpp
...
01 union Color {
02     unsigned int hexRGB;
03     std::string stringRGB;
```



```

04  Color(){};    // Standardkonstruktor
05  ~Color(){};   // Destruktor
06  };
...
07  Color color01;
08  std::string farbe;
09  std::cout << "Farbe eingeben: ";
10  std::cin >> farbe;

11  new ( &color01.stringRGB) std::string(farbe);
12  std::cout << color01.stringRGB << std::endl;

```

In diesem Fall wurde in Zeile 11 auch noch das sogenannte *Placement new* verwendet, womit nicht direkt neuer Speicher reserviert wird, sondern in bereits reservierten Speicher (hier `farbe`) an die Adresse von `&color.stringRGB` übergeben wird. Und wenn hier die Frage kommt, ob es auch ein Placement `delete` gibt, so lautet die Antwort: Nein! Dieses müssen Sie selbst implementieren – aber das geht hier jetzt hier zu weit.

4.7 Aufzählungstypen (C++11-Version)

Mit C++11 hat endlich auch C++ einen für die Sprache typsichereren `enum`-Aufzählungstyp bekommen. Ein einfaches Beispiel hierzu:

```

01  enum class RGB {red, green, blue};
02  enum class RGBA {red, green, blue, alpha};

```

Das alte C-Style-`enum` hätte hier schon bei der Übersetzung einen Fehler gemeldet, weil hier mit `red`, `green`, `blue` zwei gleiche Bezeichner eingeführt wurden. Das alte `enum` war noch »unscoped«. Mit dem neuen C++11-`enum` ist dies jetzt kein Problem mehr, da dies jetzt »scoped« ist und die korrekten Bezeichner hier `RGB::red`, `RGB::green`, `RGB::blue`, `RGBA::red`, `RGBA::green` usw. lauten.

Mit den `enum` in den Zeilen 01 und 02 haben Sie jeweils einen neuen Datentyp mit dem Namen `RGB` bzw. `RGBA` eingeführt. Die einzelnen Aufzählungen werden zwischen den geschweiften Klammern, getrennt durch ein Komma, aufgelistet. Die Enumeration wird mit einem Semikolon beendet. Die erste Aufzählungskonstante beginnt mit 0 (hier jeweils `red`), wenn nichts anderes angegeben wurde. Wenn auch nicht anders vorgegeben, wird die jeweils nächste Aufzählung immer um den Wert 1 erhöht. Somit wären bei `RGB` und `RGBA` die Werte `red=0`, `green=1`, `blue=2` und bei `RGBA` `alpha=3`.

Die Werte der Aufzählungen können aber auch angepasst werden, z.B.:

```

01  enum class RGB {red=1, green, blue};

```

C-Style-enum vs. C++11-enum

An dieser Stelle soll gleich angemerkt werden, dass hier das neue C++11-enum und nicht das alte C-Style-enum beschrieben wird. Wenn Ihr Compiler kein C++11 kann, dann müssen Sie das Schlüsselwort `class` bzw. `struct` weglassen.

Das Problem beim alten C-Style-enum ist ganz einfach:

- ▶ Es wurde implizit immer zu `int` konvertiert. Oder genauer, der Compiler hat für Sie entschieden, welches `int` er als `enum` verwendet. Mit dem neuen C++11-enum können Sie den grundlegenden Typ selbst anlegen.
- ▶ Das alte `enum` konnte für allerlei Vergleiche verwendet werden. So konnten Sie Vergleiche mit unterschiedlichen Integer- oder gar anderen `enum`-Typen durchführen. Wozu »Äpfel« definieren, um diese dann mit »Birnen« zu vergleichen? Oder einfacher, wozu »Wochentage« mit »Monaten« vergleichen? Die neuen `enums` sind jetzt streng typisiert. Wenn Sie unbedingt unterschiedliche Typen benötigen, können Sie immer noch eine explizite Umwandlung durchführen.
- ▶ Mit den alten `enum` wurden die Bezeichner in die Umgebung eingeführt. Sie konnten quasi ohne den Bereichsoperator auf die Bezeichner in `enum` zugreifen. Wenn Sie einen Bezeichner verwendeten, der bereits in einem anderen `enum` als Bezeichner verwendet wurde, ließ sich das Programm nicht mehr übersetzen. Mit dem neuen C++11-enum müssen Sie jetzt mit dem `enum`-Typ und Zugriffsoperator auf den Bezeichner zugreifen.

Jetzt beginnt die Aufzählung bei `red` mit 1, `green` ist dann 2 und `blue` wäre somit 3. Natürlich spricht auch nichts dagegen, für jede Aufzählungskonstante einen gesonderten Wert zu vergeben:

```
01 enum class RGB{red=0xff0000, green=0x00ff00, blue=0x0000ff};
```

Theoretisch können Sie auch zwei verschiedene Aufzählungskonstanten mit demselben Wert versehen. Nur können Sie logischerweise keine zwei Konstanten mit demselben Bezeichner verwenden.

4.7.1 Zugriff auf die Bezeichner

Der Zugriff auf die einzelnen Bezeichner muss bei dem neuen C++11-enum über den Zugriffsoperator `::` erfolgen. Ein Zugriff auf den Bezeichner `red` von `RGB` muss daher über `RGB::red` erfolgen. Wenn Ihr Compiler noch kein C++11 kann, müssen Sie den Zugriff über `RGB::` weglassen. Hierzu einige mögliche und nicht mögliche Beispiele:

```
00 // kapitel04/enum.cpp
...
01 enum class RGB {red, green, blue};
```

```

02 enum class RGBA {red, green, blue, alpha};
...
03 RGB rgb01{RGB::red}, rgb02 = RGB::blue;
04 RGBA rgba01{RGBA::green};

05 if( rgb01 == RGB::red ) {
06     // Ja, es ist RGB::red
07 }

08 switch( rgba01 ) {
09     case RGBA::red : ; //...
10     case RGBA::green: ;
11     case RGBA::blue : ;
12     case RGBA::alpha: ;// ...
13 }

14 int mycolor01 = rgb02; // Fehler in C++11
15 int mycolor02 =static_cast<int>(rgb02);

```

Wie im Beispiel zu sehen, sind die Aufzählungstypen, richtig angewendet, ein gutes Hilfsmittel, um die Programmierung zu erleichtern. Anstatt einfach `int rgb01=0` (für red) zu schreiben, lässt sich eine Zeile wie `RGB rgb01=RGB::red` wesentlich besser lesen und handhaben. Gerade aus `if`-Überprüfungen (siehe Zeilen 05 bis 07) oder Fallunterscheidungen (Zeilen 08 bis 13) geht dies deutlich hervor.

Und wie schon erwähnt, ist das neue C++11-enum sehr streng typisiert, und daher können Sie nicht einfach einen enum-Wert einem Integer-Wert zuweisen, wie dies in Zeile 14 versucht wurde – obwohl enum ohne weitere Angaben vom Typ `int` ist. Dies funktioniert jetzt nur noch mit einer expliziten Umwandlung, wie in Zeile 15 mit `static_cast<>` demonstriert wird.

4.7.2 Typ für enum festlegen

Der Standardtyp der Aufzählungskonstanten von enum ist `int`. In C++11 können Sie jetzt aber diesen Typ durch einen anderen integralen Typ überschreiben. Müssen Sie bspw. Speicherplatz sparen, können Sie `char` für die Breitenangabe der Aufzählungskonstanten verwenden:

```
enum class RGB : char {red, green, blue};
```

<cstdint> (C++11)

Leider hatte C++ nie bestimmte Integer-Typen mit einer fixen Breite. Wenn Sie bspw. ein 32-Bit-Integer verwenden wollen, werden Sie wohl auf `int` zurückgreifen. Leider

gibt es keine Garantie, dass `int` auf allen Systemen auch 32 Bit breit ist. Mit C++11 wurde daher der aus der Programmiersprache C (genauer dem C99-Standard) bekannte Header `<stdint.h>` als `<cstdint>` übernommen. Dieser Header enthält Typen mit fixer Breite wie `std::int8_t` (8-Bit-Integer), `std::int16_t` (16-Bit-Integer), `std::int32_t` (32-Bit-Integer) usw. Auch die unsigned-Versionen davon sind mit `std::uint8_t`, `std::uint16_t` usw. enthalten. Hiermit können Sie dann auch quasi Aufzählungstypen mit Typen verwenden, welche eine fixe Breite haben:

```
01 #include <cstdint>
02 enum class RGB : std::int8_t {red, green, blue};
```

Damit verwenden Sie quasi mit `std::int8_t` ein 8 Bit breites Integer, das auf allen Compilern und Architekturen dieselbe Breite hat.

4.8 Synonym-Technik

In C++ gibt es zwei Möglichkeiten, um einen alternativen Namen (ein Synonym oder auch einen Aliasnamen) aus einem vorhandenen Typ zu machen, zum einen mit `typedef` und zum anderen mit Alias-Templates (neu in C++11). Wohlgedenkt, damit erstellen Sie keinen neuen Typ, sondern nur einen neuen Namen. Dies kann recht hilfreich sein, um komplizierte oder lange Typnamen als verständlichere oder kurze Namen anzubieten. Eine solche Synonym-Technik ist somit ein weiteres Hilfsmittel, um den Code besser lesbar und gegebenenfalls einfacher zu machen. Betrachten Sie folgenden Codeausschnitt:

```
info_t get_text( ) const;
```

Als Rückgabewert finden Sie hier einen selbst definierten Typ `info_t`, welcher deutlich macht, dass diese Funktion eine Information zurückgibt. Es ist hier eigentlich zunächst egal, von welchem Typ `info_t` ist, so lässt es sich schon mal besser lesen, als würde hier einfach nur der Basisdatentyp stehen.

4.8.1 typedef

Mit dem Schlüsselwort `typedef` können Sie wie folgt ein Synonym aus einem vorhandenen Typ erstellen:

```
typedef Typ Synonym;
```

Die Standardbibliothek macht bspw. Gebrauch von `typedef` für `std::string`, was nur ein Aliasname für die Template-Klasse `basic_string<char>` ist. Hierzu ebenfalls einige Beispiele:

```

00 // kapitel04/typedef.cpp
01 #include <map>      // für std::map
02 #include <string>   // für std::string

03 typedef double km_t;
04 typedef double miles_t;

05 typedef struct Knoten* KnotenPtr_t;
06 struct Knoten {
07     int daten;
08     KnotenPtr_t next;
09 };

10 typedef std::map<std::string, int> AArray_t;
...
11 km_t km_distance01 = 123.33;
12 miles_t m_distance01 = 234.99;
13 AArray_t plz;
14 plz["Frankfurt"]=60311;

```

In den Zeilen **03** und **04** wurde jeweils ein Synonym für `double` mit `km_t` und `miles_t` erstellt. In den Zeilen **11** und **12** sehen Sie die beiden Synonyme für `double` im Einsatz. In Zeile **05** wurde ein `typedef` verwendet, um einen rohen Zeiger zu verstecken. Statt `Knoten*` kann hier jetzt auch das Synonym `KnotenPtr_t` verwendet werden, wie Sie dies in Zeile **08** sehen können. Mehr C++11-like ist dann das `typedef` in Zeile **10**, wo das Template `std::map<std::string, int>`, welches Sie aus Abschnitt 4.1.4, »Assoziatives Array«, kennen, vereinfacht wurde und jetzt ohne `std::map<std::string, int>` verwendet werden kann, wie die Zeilen **13** und **14** zeigen.

Solche Synonyme wurden in der Vergangenheit häufig auch für die Portierung von Programmen verwendet, um bspw. Typen mit einer bestimmten Breite anzubieten. So wurde bspw. gerne mit

```
typedef short INT16_t;
```

ein 16-Bit-Typ erstellt. Auf anderen (häufig älteren) 16-Bit-Systemen, wo ein `int` allerdings auch tatsächlich 16 Bit hat, musste dann lediglich diese Zeile geändert werden in:

```
typedef int INT16_t;
```

Allerdings bietet C++11 für solche Sachen bspw. eine Überprüfung zur Übersetzungszeit mit `static_assert()` (siehe Abschnitt 2.6.4, »Sicherheit beim Kompilieren mit `static_assert` (C++11)«) oder die in C++11 eingeführte C99-Headerdatei `<cstdint>` an.

4.8.2 Alias-Templates (C++11)

Etwas moderner und mehr C++-like ist das neue Alias-Template, welches als schöneres `typedef` verwendet werden kann und wenn möglich auch verwendet werden sollte. Mit dem neuen Alias-Template ist es jetzt u.a. möglich, mit `using` etwas einfachere Synonyme zu erstellen. Die Alias-Templates können aber noch etwas mehr, doch die Templates sind hier noch nicht das Thema. Bezogen auf die Beispiele aus Listing *typedef.cpp* sieht die modernere Version mit `using` wie folgt aus:

```
00 // kapitel04/aliastemplates.cpp
01 using km_t = double;
02 using miles_t = double;

03 using KnotenPtr_t = struct Knoten*;
04 struct Knoten {
05     int daten;
06     KnotenPtr_t next;
07 };

08 using AArray_t = std::map<std::string, int>;
...
```

Persönlich empfinde ich die Erstellung von Synonymen mithilfe von `using` und dem Zuweisungsoperator zugänglicher und verständlicher. Daher ist hier auch meine Empfehlung, diese, falls möglich (da C++11), in der Praxis zu verwenden.

4.9 Fazit

An dieser Stelle angekommen, haben Sie eine Menge an Informationen zu fortgeschrittenen Typen erhalten. Ich bin hier schonungslos auch auf C-Style-Dinge eingegangen und habe Ihnen immer die besseren Alternativen dazu genannt. Es folgt stets der Hinweis, diese C-Style-Dinge nicht zu verwenden und stattdessen auf echtes C++ zurückzugreifen. Es ist zwar so, dass C++ von C abstammt, aber die Verwendung von echten C++-Dingen ist einfach sicherer und komfortabler zu verwenden. Dass viele ältere C-Dinge in diesem Abschnitt trotzdem erwähnt und beschrieben wurden, liegt daran, dass immer noch sehr viel alter C++-Code gemischt mit C-Style vorhanden ist, und man daher solche Dinge auch als C++-Programmierer im Repertoire haben sollte. Beispiele der veralteten C-Style- gegenüber den neueren C++-Varianten finden Sie in der Übersicht nochmal in Tabelle 4.4.

C-Style	C++-Alternative
rohe C-Style-Arrays	Standardcontainer
rohe C-Style-Strings	<code>std::string</code>
rohe Zeiger	Standardcontainer, Referenzen, Smart Pointer
Strukturen und Funktionen	Klassen mit Elementfunktionen
rohe Datenstrukturen	Standardcontainer wie <code>std::list</code> oder <code>std::forward_list</code> (C++11)

Tabelle 4.4 Diverse C-Style-Techniken und deren C++-Alternativen

Kapitel 5

Funktionen

Als Funktionen werden Unterprogramme (auch als Prozedur oder Routine bekannt) bezeichnet, welche eine bestimmte Aufgabe in eventuell mehrere sinnvolle und überschaubare Teilaufgaben unterteilen.

Um ein Programm einfachere bis komplexere Aufgaben lösen zu lassen, kann man dieses in mehrere Teilaufgaben – die Funktionen – zerlegen. Einer solchen Funktion können optional auch Daten übergeben werden. Ebenso kann eine Funktion (optional) Daten an den Aufrufer dieser Funktion zurückgeben, damit der Aufrufer entsprechend mit dem zurückgegebenen Wert reagieren oder weiterarbeiten kann (bspw. diese Daten an eine weitere Funktion weiterreichen kann). Wenn Sie eine Funktion erst einmal definiert haben, können Sie diese anschließend beliebig oft aufrufen. Immer wiederkehrende Befehle können Sie praktisch in eine Funktion packen und somit mit nur einem »Befehl« (dem Funktionsaufruf) ausführen, so dass Sie diese wiederkehrenden Befehle nicht erneut schreiben müssen. Außerdem sorgen Funktionen für eine bessere Übersicht im Quellcode, wenn sie in logische Einheiten aufgeteilt sind.

5.1 Funktionen definieren

Bevor Sie eigene Funktionen schreiben können, will ich Ihnen zunächst das komplette Grundgerüst von solchen, mit allen Bestandteilen, auflisten. Hier die grundlegende Syntax einer Funktion als Pseudocode:

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

Eine grundlegende Funktionsdefinition enthält somit folgende Bestandteile:

- **Spezifizierer:** Hiermit können Sie optional ein Speicherlassenattribut verwenden. Die einzelnen Spezifizierer werden in Abschnitt 6.2, »Speicherlassenattribut«, umfassender beschrieben.
- **Rückgabetyf:** Damit legen Sie fest, welchen Typ die Funktion zurückgibt. Hierbei können beliebige Datentypen verwendet werden. Für eine Funktion ohne Rückgabewert wird hier statt eines Typs das Schlüsselwort `void` (= leerer Datentyp) verwendet.

- **Funktionsname:** Hier müssen Sie einen eindeutigen Funktionsnamen verwenden, mit dem Sie diese Funktion von anderen Stellen im Programm aufrufen können. Für einen gültigen Namen bzw. Bezeichner gelten dieselben Regeln wie schon bei den Variablennamen (siehe Abschnitt 2.1.1, »Erlaubte Bezeichner für den Zugriff auf Variablen«). Außerdem sollten Sie vermeiden, bereits verwendete Funktionen der Laufzeitbibliothek zu verwenden.
- **Parameter:** Zwischen den runden Klammern der Funktion können Sie die Parameter schreiben. Die Parameter sind optional, aber die runden Klammern nicht. Diese müssen bei einer Funktion immer vorhanden sein. Die einzelnen Parameter werden mit ihrem Datentyp und dem Bezeichner spezifiziert. Wenn Sie mehrere Parameter verwenden, müssen diese durch ein Komma voneinander getrennt werden. Für Funktionen ohne Parameter können Sie die Liste zwischen den runden Klammern einfach leer lassen. In C++ ist (im Gegensatz zu C) eine leere Argumentliste gleichwertig zu `void`, weshalb Sie darauf in diesem Fall auch verzichten können.
- **Anweisungsblock mit Anweisungen:** Wie Sie dies bereits von der `main`-Funktion her kennen, enthält eine Funktionsdefinition einen Anweisungsblock, in dem Sie die einzelnen Anweisungen für die Funktion und lokale Deklarationen zwischen den geschweiften Klammern zusammenfassen.

Neue Rückgabesyntax von C++11

In C++11 wurde eine neue Schreibweise für Funktionen eingeführt, bei der der Rückgabebetyp ans Ende des Funktionskopfes gestellt wird. Mithilfe der neuen Syntax und `auto` können Sie lange und komplexe Rückgabewerte erheblich vereinfachen. Mehr zu dieser neuen Syntax finden Sie in Abschnitt 5.9, »Die neue Funktionssyntax (C++11)«.

Eine einfache Funktion ohne Rückgabewert und ohne Parameter kann somit folgendermaßen aussehen:

```
void simple_function() {
    std::cout << "Funktion bei der Ausführung\n";
}
```

Funktionen innerhalb einer Funktionsdefinition zu definieren geht nicht

An dieser Stelle sollte vielleicht noch kurz darauf hingewiesen werden, dass es nicht möglich ist, innerhalb einer Funktionsdefinition zwischen dem Anweisungsblock `{...}` eine weitere Funktion zu definieren. Folgendes geht also nicht:

```
void simple_function01() {
    ...
    void simple_function02() { // Fehler !!! Nicht erlaubt in C++
    }
```

```

    ...
}
}

```

Es ist aber möglich, sogenannte *Funktoren* lokal zu definieren – also eine lokale Klasse, die den `operator()` definiert. Sie kann somit ein vollwertiger Ersatz für eine lokale Funktion sein. Seit C++11 kann man auch eine Lambda-Funktion hierzu verwenden, das heißt eine anonyme Funktion. Mehr zu diesen beiden fortgeschrittenen Themen finden Sie in den Abschnitten 8.3.16, »Funktionsoperator() – Funktionsobjekte«, 12.5.5, »Funktionsobjekte«, 5.8.2, »Lambda-Funktionen (C++11)«, und 12.5.6, »Lambda-Funktion (C++11)«.

5.2 Funktionen aufrufen

Mit den Kenntnissen zu den grundlegenden Elementen einer Funktion können Sie nun selbst ein einfaches Beispiel erstellen, welches eine Funktion aufruft:

```

00 // kapitel05/function01.cpp
01 #include <iostream>
02 void simple_function() {
03     std::cout << "Die Funktion bei der Ausführung\n";
04 }
05 int main() {
06     simple_function();
07     std::cout << "Nach dem Funktionsaufruf\n";
08 }

```

Wenn Sie das Programm starten, wird zunächst immer die `main`-Funktion ausgeführt. In Zeile **06** wird dann mit dem Funktionsnamen (hier `simple_function()`) die Funktion aufgerufen, welche hier in den Zeilen **02** bis **04** definiert wurde. Auch wenn die Funktion keine Parameter erwartet, müssen Sie die runden Klammern beim Funktionsaufruf verwenden. In der Funktion wird im Grunde nur eine einfache Ausgabe über `std::cout` in Zeile **03** gemacht. Nach der Ausführung der Funktion wird in der `main`-Funktion hinter dem Funktionsaufruf in Zeile **07** fortgefahren – was in diesem Beispiel eine weitere Ausgabe über `std::cout` auf dem Bildschirm ist.

5.3 Funktionen deklarieren (Vorausdeklaration)

Wie Sie es im Minimalbeispiel in Listing *function01.cpp* eben gesehen haben, werden Sie selten eine Funktion direkt in derselben Quelldatei wie den Aufrufer definieren und aufrufen. Wenn Sie bereits einige Quellcodes studiert haben oder in den Header-

dateien der Standardbibliothek gesehen haben, werden Sie festgestellt haben, dass hier meistens nur der Funktionskopf ohne den Anweisungsblock und die Anweisungen deklariert ist. Solch eine Funktionsdeklaration ohne den Code der eigentlichen Funktion wird als *Funktionsprototyp* bezeichnet. Hier die grundlegende Syntax eines Funktionsprototyps als Pseudocode:

```
Rückgabetyf Funktionsname( [Parameter] );
```

Ein **Funktionsprototyp** zeichnet sich dadurch aus, dass er die Funktion nur deklariert und nicht definiert. Für eine Deklaration reicht lediglich der Funktionskopf aus, welcher neben dem Funktionsnamen den Typ des Rückgabewertes und die Anzahl und Typen der Parameter enthält. Statt des Anweisungsblocks wird dieser Funktionsprototyp mit einem Semikolon abgeschlossen. Eine solche Deklaration wird häufig auch als **Vorausdeklaration** (engl. *forward declaration*) bezeichnet. Neben diesem Begriff findet man häufig auch die falsche Übersetzung mit Vorwärtsdeklaration. Eine einfache Vorausdeklaration könnte demnach wie folgt aussehen:

```
// Vorausdeklaration
void simple_function();
```

Mit einem solchen Funktionsprototyp teilen Sie dem Compiler die Schnittstelle einer Funktion mit, bevor diese verwendet wird, damit dieser bei der Übersetzung überprüfen kann (genauer, eine Syntaxüberprüfung vornehmen kann), ob der Aufruf korrekt erfolgt ist. Die Implementation bzw. genauer die Definition der Funktion (mitsamt Anweisungsblock) kann jetzt entweder zu einem späteren Zeitpunkt oder, wie es meistens der Fall ist, in einem anderen Programmmodul erfolgen. Ein Beispiel:

```
00 // kapitel05/function02.cpp
01 #include <iostream>
02 // Vorausdeklaration (Funktionsprototyp)
03 void simple_function();
04 int main() {
05     simple_function();
06 }
07 // Funktionsdefinition
08 void simple_function() {
09     std::cout << "Funktion bei der Ausführung\n";
10 }
```

In Zeile **03** wird in diesem Beispiel eine einfache Vorausdeklaration der Funktion `simple_function()` gemacht. Ohne diesen Funktionsprototyp in Zeile **03** würde sich der Compiler in Zeile **05** beschweren, dass ihm zu diesem Zeitpunkt noch keine Funktion `simple_function()` bekannt ist, welche erst hinter dem Aufruf in den Zeilen **07** bis **10** definiert wurde.

Auch wenn es bei den einfachen Beispielen im Buch so verwendet wird, ist es in der Praxis eher unüblich, wie im Beispiel in Listing *function02.cpp* die Definition hinter bzw. die Vorwärtsdeklaration vor der `main`-Funktion zu schreiben. Hier packt man gewöhnlich die Definition in eine extra Quelldatei, und der Funktionsprototyp kommt in eine Headerdatei. Als Programmierer stehen einem oft nur die Funktionsprototypen zur Verfügung. Die eigentliche Definition bleibt häufig verborgen und wird oft nur als Objektcode oder Bibliotheksdatei mitgeliefert – so wie dies bspw. bei Ihrer C++-Standardbibliothek auch der Fall ist. Als Programmierer müssen Sie hierbei häufig nur noch die Headerdatei einbinden und bei der Übersetzung dem Linker den Pfad zur Objektdatei oder Bibliotheksdatei mitteilen.

Code organisieren

Um hier jetzt nicht zu weit vom Thema abzuschweifen, möchte ich Sie auf den Abschnitt 6.6, »Modulare Programmierung – Code organisieren«, verweisen.

5.4 Exkurs: Gültigkeitsbereiche

An dieser Stelle wird jetzt ein kleiner Exkurs nötig, um Sie für die verschiedenen Gültigkeitsbereiche (Abbildung 5.1) zu sensibilisieren.

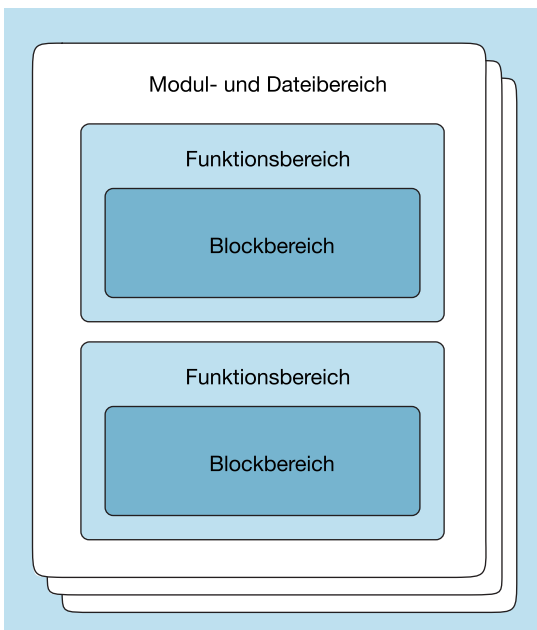


Abbildung 5.1 Übersicht zu den unterschiedlichen Gültigkeitsbereichen ohne Klassen, die erst später im Buch behandelt werden

5.4.1 Globaler Gültigkeitsbereich

Alle Daten, die Sie nicht innerhalb eines Anweisungsblocks `{...}` definieren, sind globale Daten und gültig ab der Position, an der diese definiert werden, bis zum Ende des Moduls bzw. der Datei. Im Gegensatz zu lokalen Variablen werden globale Variablen beim Programmstart automatisch mit 0 initialisiert – das gilt allerdings nur für die Typen ohne Konstruktoren, bei denen stattdessen der Standardkonstruktor aufgerufen wird (worüber Sie sich an dieser Stelle noch keine Gedanken machen müssen). Folgendes Beispiel soll die globalen Daten kurz demonstrieren:

```
00 // kapitel05/function03.cpp
01 #include <iostream>
02 // Globale Variable
03 int glob_var01;
04 // Vorausdeklaration
05 void func();

06 int main() {
07     int loc_var01 = glob_var01;
08     int loc_var02 = glob_var02; // Fehler !!!
09 }

10 // Globale Variable
11 int glob_var02;
12 // Funktionsdefinition
13 void func() {
14     int loc_var01 = glob_var02;
15 }
```

In Zeile **03** wird die globale Variable `glob_var01` definiert, welche übrigens automatisch mit 0 initialisiert wird. Eine Zuweisung wie in Zeile **07** ist daher möglich, weil diese globale Variable hier in der gesamten Datei gültig ist. Die Zuweisung in Zeile **08** von der globalen Variablen `glob_var02` hingegen führt zu einer Fehlermeldung des Compilers, weil diese globale Variable dem Compiler an dieser Stelle noch nicht bekannt ist und erst in Zeile **11** definiert wird. Diesen Fehler könnten Sie beheben, indem Sie die globale Variable aus Zeile **11** gleich hinter Zeile **03** verschieben. Die Zuweisung innerhalb der Funktion `func()` in Zeile **14** hingegen ist wieder in Ordnung, weil die globale Variable `glob_var02` hier bereits bekannt ist. Die Verwendung des gleichnamigen Bezeichners `loc_var01` in den Zeilen **07** und **14** führt auch zu keinen Problemen, weil beide lokal innerhalb eines Anweisungsblocks definiert wurden und daher nur innerhalb des Anweisungsblocks gültig sind.

Verzichten Sie wenn möglich auf globale Daten

Es ist zwar erlaubt, globale Daten zu verwenden, aber in der Praxis werden diese nicht unbedingt empfohlen, weil sie auch oft unerwartete Seiteneffekte mit sich bringen. So wird gerne eine globale Variable mit der Funktion `abc()` verwendet und dort bspw. eine Funktion `xyz()` aufgerufen, welche ebenfalls auf diese globale Variable zugreift und sie verändert. So produziert man oft unbeabsichtigte und unvorhersehbare Seiteneffekte. Auch in umfangreicheren Programmen, wo man gerne gängige Bezeichner wie `wert`, `var`, `ivar`, `val`, `i` usw. verwendet, werden versehentlich gerne mal zwei Variablen mit demselben Bezeichner verwendet.

Wenn Sie trotzdem nicht darum herum kommen und globale Variablen verwenden wollen, so sollten Sie diese möglichst am Modulanfang (ganz oben in der Quelldatei) definieren.

5.4.2 Lokaler Gültigkeitsbereich

Einen lokalen Gültigkeitsbereich haben die Daten, welche innerhalb einer Funktion oder innerhalb eines Anweisungsblocks definiert wurden. Und wenn solche Daten innerhalb einer Funktion oder innerhalb eines Anweisungsblocks definiert wurden, sind sie auch nur innerhalb dieses Bereichs gültig. Daher ist es möglich, dass Sie bspw. in mehreren Funktionen einen Bezeichner mit demselben Namen verwenden können. Auch lokale Daten können innerhalb einer Funktion oder eines Anweisungsblocks an einer beliebigen Stelle definiert werden und sind gültig ab der Position der Definition bis zum Ende der Funktion bzw. zum Ende des Anweisungsblocks, worin die Daten definiert wurden. Im Grunde sind die lokalen Daten von Funktionen und Anweisungsblöcken fast dasselbe – nur können Sie zusätzlich innerhalb einer Funktion noch weitere Blockbereiche mit neuen (oder gleichnamigen) Daten anlegen.

Wird ein Anweisungsblock ausgeführt, werden lokale Daten auf einem Stack (= Stapelspeicher; ein spezieller Bereich eines Programms für temporäre Daten) abgelegt. Es ist ganz wichtig, zu wissen, dass diese Daten nur während der Funktionsausführung gültig sind. Wenn diese Funktion mit der Ausführung fertig ist, werden diese Daten vom Stack wieder entfernt. Dasselbe gilt auch für Daten innerhalb eines Anweisungsblocks.

Hierzu ein Listing, welches die lokalen Daten innerhalb einer Funktion und innerhalb von Blöcken demonstrieren soll:

```
00 // kapitel05/function04.cpp
01 #include <iostream>
02 // Globale Variable
03 int var=1;
```

```

04 // Vorausdeklaration
05 void func1();
06 void func2();

07 int main() {
08     int var=2; // verdeckt das globale var
09     {
10         int var=3; // überdeckt lokales var
11         var=4;
12         ::var=5; // Zugriff auf globales var
13     }
14 }

15 void func1() {
16     int var=6; // verdeckt globales var
17     {
18         var=7; // Zugriff auf funktionslokales var
19     }
20 }

21 void func2() {
22     var=8; // Zugriff auf globales var
23 }

```

Zunächst wurde in diesem Beispiel in Zeile **03** eine globale Variable `var` definiert. In der `main`-Funktion wird in Zeile **08** eine weitere lokale Variable `var` definiert, welche jetzt das globale `var` aus Zeile **03** verdeckt. Innerhalb der `main`-Funktion sehen Sie zwischen den Zeilen **09** bis **13** einen neuen Blockbereich, in dem in Zeile **10** eine weitere Variable `var` definiert wird. Diese Variable ist jetzt nur in diesem Blockbereich gültig und überdeckt das lokale `var` aus Zeile **08**. Die Zuweisung in Zeile **11** bezieht sich daher auch auf die Blockvariable `var` in Zeile **10**.

In Zeile **12** wird der Gültigkeitsoperator `::` (engl. *scope resolution operator*) vor der Variable `var` verwendet. Wird dieser Operator vor dem Variablennamen gesetzt, so kann damit auf globale Daten zugegriffen werden. Wohlgemerkt, damit ist es nur möglich, auf die globale Variable `var` in Zeile **03** zuzugreifen und nicht auf lokale übergeordnete Daten wie im Beispiel in Zeile **08**!

In Zeile **15** bis **20** finden Sie wieder eine Funktionsdefinition, in der in Zeile **16** eine weitere funktionslokale Variable `var` definiert wurde, welche nur innerhalb dieser Funktion gültig ist und die globale Variable aus Zeile **03** wieder überdeckt. In Zeile **18** wird dann in einem Blockbereich der funktionslokalen Variablen aus Zeile **16** ein neuer Wert zugewiesen. Hier sehen Sie schon, dass immer nach der lokalsten Variablen im über-

geordneten Bereich gesucht wird. In der zweiten Funktionsdefinition in den Zeilen 21 bis 23 wird in Zeile 22 `var` ein neuer Wert zugewiesen. Da es in diesem Fall keine lokale Variable gibt, gilt diese Zuweisung dem übergeordneten globalen `var` aus Zeile 03.

Ausnahme: Variablen mit `static` kennzeichnen

Es gibt noch eine Ausnahme, um eine lokale Variable innerhalb einer Funktion oder eines Blocks dauerhaft mit einem festen Speicherplatz zu versehen. Hierfür müssen Sie lediglich das Schlüsselwort `static` verwenden. Mit `static` gekennzeichnete Variablen erhalten einen festen Speicherplatz im Programm und werden im Fall der Verwendung innerhalb von Funktionen oder Blöcken nicht mehr nur auf das temporäre Stack gelegt und behalten somit auch beim Verlassen der Funktion ihren Wert. Mehr über dieses Speicherklassenattribut erfahren Sie in Abschnitt 6.2.2, »Das Schlüsselwort `static`«.

5.5 Funktionsparameter

Wie Sie bereits eingangs in Abschnitt 5.1, »Funktionen definieren«, erfahren haben, können Sie Funktionen auch Daten übergeben. Dabei handelt es sich quasi um den Datentransfer in die Funktion hinein. In C++ sind zwei Varianten sehr gängig, um einer Funktion die Argumente zu übergeben: *Call by Value* und *Call by Reference*. Natürlich ist es nach wie vor auch möglich, einer Funktion rohe Zeiger als Argumente zu übergeben. Dies kann nützlich sein, wenn Sie eine alte Schnittstelle vor sich haben und bspw. rohe C-Style-Arrays an einer Funktion übergeben müssen. Trotzdem gilt auch hier, dass die Verwendung von rohen Zeigern nicht sehr C++-like sind und daher nur verwendet werden sollte, wenn es wirklich sein muss.

Bei der Parametrisierung von Funktionen wollen wir zunächst die Deklaration des Funktionsprototyps betrachten. Bei der Deklaration müssen Sie mindestens den Datentyp des Parameters angeben, damit der Compiler eine Typprüfung beim späteren Aufruf der Funktion durchführen kann. Mehrere Parameter werden durch ein Komma getrennt. Die Anzahl der Parameter, die Sie verwenden können, ist nicht beschränkt. Hierzu zwei einfache Funktionsprototypen:

```
void area_of_a_disk(double);
void capacity(double, double);
```

Wie Sie hier sehen können, kann bei der Deklaration auf die Bezeichner bei den Parametern verzichtet werden. In der Praxis würde ich Ihnen allerdings davon abraten, weil Sie mithilfe eines Bezeichners den Sinn der Parameter quasi noch hinzuschreiben können, sofern Sie keine sinnfreien Bezeichner wie `i`, `x`, `y`, oder `var` verwenden. Also machen Sie besser folgende Funktionsdeklarationen:


```
void area_of_a_disk(double radius);
void capacity(double radius, double high);
```

Anhand der verwendeten Bezeichner bei den Funktionsprototypen lässt sich jetzt auch schon eher errahnen, was diese Funktionen machen werden. Auf keinen Fall sollten Sie auf einen Bezeichner verzichten, auch wenn Sie Funktionen schreiben, bei denen Sie nur die Schnittstelle weitergeben und die eigentliche Implementation lediglich als Objekt- bzw. Bibliotheksdatei mitliefern.

Unnötig zu erwähnen, aber es sollte zur Sicherheit noch gesagt werden, dass die Reihenfolge und Anzahl der Datentypen und Parameter bei der Deklaration mit der späteren Funktionsdefinition übereinstimmen muss. Nicht übereinstimmen müssen hingegen die verwendeten Bezeichner der Parameter. Wenn Sie in der Deklaration `radius` geschrieben haben und in der Definition später `r`, so ist dies kein Fehler. Innerhalb der Funktion müssen Sie dann logischerweise den Bezeichner verwenden, den Sie im Funktionskopf der Definition (hier also `r`) geschrieben haben.

5.5.1 Funktionsparameter als Kopie (Call by Value)

Bei dieser Möglichkeit werden die Werte als Kopie an die Funktion übergeben. Sie können diesen Wert zwar innerhalb der Funktion ändern, aber der Originalwert des Aufrufers wird nicht angerührt. Sehen Sie hierzu folgendes Beispiel an:

```
00 // kapitel05/function05.cpp
01 #include <iostream>

02 void area_of_a_disk(double radius);
03 void capacity(double radius, double high);

04 int main() {
05     double radius=0.0, high=0.0;
06     std::cout << "Radius des Kreises: ";
07     std::cin >> radius;
08     std::cout << "Höhe des Zylinders: ";
09     std::cin >> high;
10     area_of_a_disk(radius);
11     capacity(radius, high);
12     std::cout << radius << std::endl; // wie in Zeile 07
13 }

14 void area_of_a_disk(double r) {
15     const double PI = 3.141592653;
16     double A = PI * r * r;
```

```

17  std::cout << "Fläche des Kreises: " << A << std::endl;
18  }

19  void capacity(double radius, double high) {
20      std::cout << "Volumen des Zylinders: "
                << 3.141592653 * radius * radius * high
                << std::endl;
21      radius = 0.0; // kein Effekt in der main()-Funktion
22  }

```

In den Zeilen **02** und **03** finden Sie die Vorausdeklarationen der Funktionen, welche wir für das Programm verwenden. Mit der Funktion in Zeile **02** soll die Fläche eines Kreises anhand des Radius und mit der Funktion (Zeile **03**) soll das Volumen eines Zylinders anhand des Radius und der Höhe berechnet werden. In der `main`-Funktion, werden Sie in den Zeilen **06** bis **09** aufgefordert, die Werte für Radius und Höhe für unsere Berechnungen einzugeben.

In Zeile **10** wird jetzt die Funktion `area_of_a_disk()` mit dem eingegebenen Wert für `radius` als Parameter aufgerufen und an die Funktion übergeben. Die Definition der Funktion finden Sie in den Zeilen **14** bis **18** vor. Der übergebene Parameter aus Zeile **10** wird dabei kopiert und als Kopie an die Funktion an `double r` übergeben. Würden Sie innerhalb der Funktion der Variablen `r` einen Wert zuweisen, so hätte diese keine Auswirkungen auf die Variable `radius` in der `main`-Funktion. Im Funktionskopf der Definition (Zeile **14**) wurde außerdem zur Demonstration ein anderer Bezeichner verwendet, als in der Deklaration (Zeile **02**) angegeben wurde. Innerhalb der Funktion wurden jetzt zwei weitere Variablen (genauer eine Variable/eine Konstante) definiert, welche nur innerhalb der Funktion gültig sind. In der Funktion wird dann die Berechnung durchgeführt und auch gleich ausgegeben. Wenn die Funktion fertig ist, werden alle Variablen auf dem Stack (Stapelspeicher für temporäre Daten), welcher für jeden Funktionsaufruf angelegt wird, wieder gelöscht, und das Programm fährt hinter dem Funktionsaufruf fort.

Vereinfacht ist der Stack (auch Stapelspeicher) ein Speicherbereich für lokale Variablen des aktuellen Gültigkeitsbereichs (wie bspw. eine Funktion oder einen Anweisungsblock). Wird dieser Gültigkeitsbereich verlassen, werden diese Objekte auch wieder automatisch zerstört. Der Stack ist ein stark begrenzter Bereich und sollte vorwiegend nur für lokale und temporäre Daten verwendet werden.

Zurück in der `main`-Funktion wird in Zeile **11** die Funktion `capacity()` mit den Variablen `radius` und `high` aufgerufen. Beide Variablen werden wieder als Kopie auf dem Stack angelegt und in der Funktion `capacity()` unabhängig von den gleichnamigen Bezeichnern der `main`-Funktion verwendet. In der Funktion `capacity()` (Zeile **19** bis **22**) wurde diesmal auf weitere temporäre Variablen verzichtet und das Ergebnis der Berechnung gleich an `std::cout` für die Ausgabe weitergereicht.

Zur Demonstration, dass es sich bei der Übergabe um eine Kopie handelt, wurde in Zeile 21 der Wert von `radius` auf 0.0 gesetzt. Diese Zeile hat keinerlei Effekt auf den Wert des Aufrufers der `main`-Funktion, wie die Ausgabe in Zeile 12 demonstrieren soll. Die Ausgabe in Zeile 12 entspricht nach wie vor dem Wert, den Sie in Zeile 07 eingegeben haben.

Auf Kopien (Call by Value) verzichten?! (Single-Threading)

Wie Sie in der Hinweisbox eben erfahren haben, werden die lokalen Variablen temporär auf einem Stack abgelegt. Bei einfachen Basisdatentypen wie `char`, `int`, `double` usw. ist die Verwendung *by Value* als Parameter noch akzeptabel, aber bei größeren Objekten (wie umfangreichen Klassen) sollten Sie allerdings, wenn möglich und Ihre Anwendung eine einfache Single-Thread-Anwendung ist, auf die Übergabe als Kopie verzichten und stattdessen Referenzen (*Call by Reference*) verwenden. Modernere Compiler beschränken diesen Speicherbereich des Stacks oftmals auf eine bestimmte Menge. Wenn Sie mehr als die erlaubte Menge an Daten auf dem Stack ablegen wollen, müssen Sie gegebenenfalls die maximale Größe anpassen.

Auf Kopien (Call by Value) verzichten?! (Multithreading)

An dieser Stelle will ich Sie trotzdem noch ein wenig sensibilisieren für die Aussage in der Hinweisbox zuvor. Noch vor C++11 wurde immerzu geraten, auf *Call by Value* zu verzichten und stattdessen *Call by Reference* zu verwenden. Das war allerdings bevor mit C++11 das Multithreading dazu gekommen ist. Was in der Single-Threading-Welt kein Problem ist, wird beim Multithreading gefährlich. Auf das Thema wird in Kapitel 15, »Multithreading (C++11)«, noch eingegangen, aber daher sollten Sie die Empfehlung »im Zweifelsfall Call by Reference« künftig zweimal überlegen – sowohl in die eine, als auch in die andere Richtung.

5.5.2 Funktionsparameter als Referenz (Call by Reference)

Neben der Übergabe als Wert (*by Value*) gibt es noch die Übergabe als Referenz (*by Reference*) an Funktionen. Referenzen haben Sie ja bereits kurz in Abschnitt 4.4, »Referenzen«, kennengelernt. Die Übergabe als Referenz wird meistens verwendet, wenn das übergebene Objekt in der Funktion geändert werden soll oder das Objekt einfach zu groß ist und einen erheblichen Zeit- und Speicheraufwand benötigen würde.

Im Gegensatz zur Übergabe als Wert, wo Sie mit einer Kopie des Originals in der Funktion arbeiten, wird mit der Übergabe als Referenz nicht das ganze Objekt, sondern nur die Speicheradresse des Arguments übergeben, weshalb Änderungen an der Variablen in der Funktion auch nach dem Funktionsaufruf bestehen bleiben. Daher

wird für solche Referenzen kein Speicher auf dem Stack benötigt. Für die Verwendung von Call by Reference müssen Sie lediglich den Operator & verwenden. Bezogen auf unser Beispiel aus Listing *funktion05.cpp* brauchen Sie daher lediglich die Funktionsprototypen und die Funktionsköpfe der Definition entsprechend mit dem kaufmännischen Und anzupassen:

```
00 // kapitel05/function06.cpp
...
02 void area_of_a_disk(const double& radius);
03 void capacity(const double& radius, const double& high);
...
14 void area_of_a_disk(const double& r) {
15     const double PI = 3.141592653;
16     double A = PI * r * r;
17     std::cout << "Fläche des Kreises: " << A << std::endl;
18 }

19 void capacity(const double& radius, const double& high) {
20     std::cout << "Volumen des Zylinders: "
                << 3.141592653 * radius * radius * high
                << std::endl;
21     // radius = 0.0; // Jetzt Fehler, da const
22 }
```

Wollen Sie sicherstellen, dass die übergebenen Werte in der Funktion nicht (versehentlich) verändert werden können, was jetzt mit Zeile 21 der Fall gewesen wäre, können Sie, wie im Codeausschnitt von Listing *funktion06.cpp* zu sehen ist, die Referenzparameter mit `const` deklarieren, um den Inhalt des Speichers vor Änderungen zu schützen.

Ein weiterer Vorteil von konstanten Referenzen in diesem Beispiel ist es, dass Sie hiermit auch ein Literal wie `area_of_a_disk(10.12)` oder `capacity(10.12, high)` für den Funktionsaufruf verwenden können. Erst mit einer `const`-Referenz ist es möglich, dass der Compiler eine temporäre Variable anlegt. Ergo bedeutet dies auch, dass nur dann nicht konstante Referenzen verwendet werden können, wenn das übergebene Argument auch eine Speicheradresse hat.

Wohin mit dem &

Die Position von & ist übrigens recht frei zwischen dem Typ und dem Bezeichner zu verwenden. Hierbei ist es egal, ob Sie `double& r`, `double &r`, `double & r` oder `double& r` verwenden. Alles ist möglich.

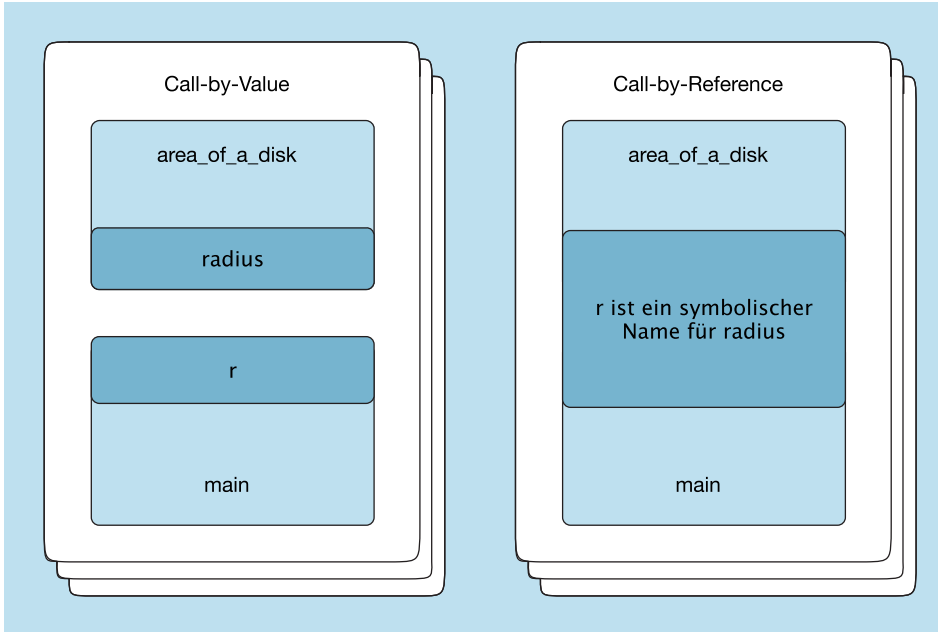


Abbildung 5.2 Unterschied zwischen dem Call-by-Value- und Call-by-Reference-Verfahren

Beim Call-by-Value-Verfahren verwendet die Funktion »area_of_a_disk« eine eigene Kopie von »radius« mit dem Bezeichner »r«. Beim Call-by-Reference-Verfahren hingegen ist »r« nur ein symbolischer Name für »radius« und entspricht somit derselben Speicheradresse (Abbildung 5.2).

5.5.3 Funktionsparameter als rohe Zeiger

Es wurde bereits erwähnt, dass Sie auch rohe Zeiger als Funktionsparameter verwenden können. Allerdings ist die Verwendung recht umständlich, und es wurden in der Vergangenheit oft Fehler damit gemacht. Dafür wurden u.a. ja die Referenzen in C++ eingeführt. Da auch bei der Übergabe von Zeigern die Adresse übergeben wird, arbeiten Sie in der Funktion mit den Originalwerten des Aufrufers und nicht mit einer Kopie. Änderungen innerhalb der Funktion wirken sich also auch hier auf Funktionen aus. Bezogen auf unser Beispiel aus Listing *funktion06.cpp* stellt folgendes Beispiel mit rohen Zeigern ein Äquivalent dar:

```
00 // kapitel05/function07.cpp
01 #include <iostream>

02 void area_of_a_disk(double* radius);
03 void capacity(double* radius, double* high);
```

```

04 int main() {
...
10     area_of_a_disk(&radius);
11     capacity(&radius, &high);
12 }

13 void area_of_a_disk(double* r) {
14     const double PI = 3.141592653;
15     double A = PI * *r * *r;
16     std::cout << "Fläche des Kreises: " << A << std::endl;
17 }

18 void capacity(double* radius, double* high) {
19     std::cout << "Volumen des Zylinders: "
                << 3.141592653 * *radius * *radius * *high
                << std::endl;
20 }

```

Im Gegensatz zu den Referenzen müssen Sie bei den rohen Zeigern selbstverständlich bei der Deklaration und Definition das Sternchen `*` anstelle des kaufmännischen `&` verwenden. Beim Aufruf müssen Sie aufpassen, dass Sie die Adresse und nicht den Wert an die Funktion übergeben. Daher müssen Sie hier den Adressoperator in den Zeilen **10** und **11** für `radius` und `high` mit angeben. In den Funktionen selbst müssen Sie dann den Indirektionsoperator vor jede Zeigervariable stellen, um auch mit dem Inhalt arbeiten zu können (siehe Zeilen **15** und **19**). Insgesamt also ziemlich umständlich, wenn man dies mit dem Beispiel in Listing *funktion06.cpp* und den Referenzen vergleicht.

Des Weiteren lässt sich mit den Zeigern auch ein Aufruf mit einem Literal wie `area_of_a_disk(10.12)` oder `capacity(10.12, high)` im Gegensatz zu den Referenzen nicht mehr verwenden.

C-Style-Arrays an Funktionen

Da es doch immer noch alte Programme und Schnittstellen gibt, soll hier noch kurz erwähnt werden, wie Sie ein C-Style-Array als Parameter an eine Funktion übergeben können. Dass hier kein ganzes Array übergeben werden kann, sollte einleuchten. Muss ja auch gar nicht sein. Da ja garantiert wird, dass Arrays immer in einem zusammenhängenden aufeinanderfolgenden Speicherbereich (ohne Lücken) vorliegen, reicht es auch aus, nur die Anfangsadresse an die Funktion zu übergeben. Neben der Anfangsadresse müssen Sie auch noch die Anzahl der Elemente des Arrays mit übergeben. Folgendes Minimalbeispiel soll Ihnen zeigen, wie Sie ein C-Style-Array an eine Funktion übergeben können:

```

00 // kapitel05/function08.cpp
01 #include <iostream>

02 void printArray(const int* array, const int nvals);

03 int main() {
04     int myarray[]={1,2,3,5,8,13,21};
05     printArray(myarray, sizeof(myarray)/sizeof(myarray[0]));
06 }

07 void printArray(const int* array, const int nvals) {
08     std::cout << "Werte im Array: ";
09     for(int i=0; i < nvals; ++i) {
10         std::cout << array[i] << " ";
11     }
12     std::cout << std::endl;
13 }

```

In Zeile **02** deklarieren wir den Funktionsprototyp. Das Array in Zeile **04** rufen wir in Zeile **05** mitsamt der Anzahl der Elemente auf. Warum Sie hier keinen Adressoperator angeben müssen, wurde bereits im Abschnitt »Rohe Zeiger und rohe Arrays« in Abschnitt 4.3 beschrieben. Dasselbe gilt auch für die Verwendung von `array[i]` innerhalb der Funktion in Zeile **10**, wo es genauso `*(array+i)` lauten könnte. Wenn Sie sich nicht mehr daran erinnern oder den Abschnitt übersprungen haben, müssen Sie jetzt nicht zurückblättern, sondern sich einfach vor Augen halten, dass Sie als C++-Programmierer solch einen Code eher nicht mehr bei neuen Projekten verwenden werden, weil Sie mit `std::vector` oder `std::array` viel bessere C++-Alternativen haben.

Verzichten Sie auf rohe C-Style-Arrays

Wenn Sie die Fäden zur Entwicklung eines neuen Projekts in der Hand halten, dann sollten Sie auf rohe C-Style-Arrays (welche häufig unweigerlich auch die Verwendung von rohen Zeigern beinhalten) verzichten und stattdessen auf die besseren C++-Alternativen zurückgreifen.

5.5.4 Strukturen und Klassen als Funktionsparameter

Alles Beschriebene zu Funktionsparametern können Sie auch für Strukturen und Klassen übernehmen. Auch hier können Sie eine Übergabe als Kopie, als Referenz oder als rohen Zeiger deklarieren und definieren. Wobei eine Kopie bei umfangreichen Daten, wie es Strukturen und Klassen meistens auch sind, die teuerste Wahl ist, weil damit der Stack mit Kopien belastet wird.

Zeiger sind auch hier nicht so empfehlenswert, weil diese die eigentliche Semantik verdecken. Bei einem Zeiger weiß man bspw. nicht sofort, wem dieser gehört (wer ist bspw. verantwortlich, den Speicher wieder freizugeben etc.). Dann kommen noch die umständliche Verwendung der Adressenübergabe und die Verwendung des Indirektionsoperators innerhalb der Funktion dazu, wenn man auf die Werte zugreifen will. Sicherlich gibt es auch Fälle, in denen man auf Zeiger zurückgreifen kann. Trotzdem sollten Sie auch hier vorher immer überlegen, ob Sie ein Argument als Referenz oder Wert übergeben sollten, und erst dann, ob es als Zeiger einen Sinn macht. Man kann also Zeiger nicht kategorisch komplett ausschließen.

Eine sehr gute Wahl sind auch hier wieder die Referenzen, sofern Sie keine Multithreading-Anwendung (siehe Kapitel 15, »Multithreading (C++11)«) schreiben. Hier kommt man bspw. gar nicht erst auf die Idee, ein &-Argument mit `delete` freizugeben. Auch kann man einem &-Argument keinen Null-Zeiger (`nullptr`) übergeben, ganz im Gegensatz zum *-Argument. Letztendlich hängt es auch vom Anwendungsfall ab, ob Sie Zeiger oder Referenzen verwenden können oder sollten.

Zwar wurden die Klassen in diesem Buch noch nicht behandelt (erst in Kapitel 7, »Grundlagen zu den Klassen«), aber da Sie bereits Kenntnisse über fertige Standardklassen wie `std::string` und `std::vector` haben, können wir diese in einem Beispiel hier zur Demonstration verwenden:

```
00 // kapitel05/function09.cpp
01 #include <iostream>
02 #include <vector>
03 #include <string>

04 struct Bookmark_t {
05     std::string titel;
06     int seite;
07 };

08 void addData( std::vector<Bookmark_t>& vec,
               const std::string& t, const int& s );

09 int main() {
10     std::vector<Bookmark_t> myvec;
11     addData(myvec, "Standardcontainer", 356);
12     addData(myvec, "Smart-Pointer", 754);
13     for(size_t it=0; it<myvec.size(); ++it) {
14         std::cout << myvec.at(it).titel << "; "
                   << myvec.at(it).seite << std::endl;
15     }
16 }
```



```

17 void addData( std::vector<Bookmark_t>& vec,
               const std::string& t, const int& s ) {
18     Bookmark_t temp = {t,s}; // Temporäres Objekt erstellen
19     vec.push_back(temp);      // Am Ende hinzufügen
20 }

```

Das Beispiel dient nur der Demonstration. Hier wurde der Standardcontainer `std::vector` verwendet, um Daten von der Struktur `Bookmark_t` in einem Array zu speichern. Die Übergabe des Originalcontainers wurde hier mit Referenzen realisiert. Ebenso wurden die weiteren Parameter für die Daten der Struktur als Referenz realisiert, wodurch eine Übergabe von Literalen wie in den Zeilen 11 und 12 auch möglich wird. Diese beiden Werte hätten Sie auch als Kopie (*by Value*) realisieren können. Mit Zeigerversionen hätten Sie allerdings Probleme mit den Literalen bekommen.

std::vector und std::string

Die Containerklasse `std::vector` wurde in Abschnitt 4.1.1, »Standardcontainer `std::vector`«, und die Klasse `std::string` in Abschnitt 4.2.1, »Standardcontainer `std::string`«, beschrieben.

5.5.5 Schreibschutz für die Referenzparameter mit const

`const` wurde jetzt bereits mehrmals in den Beispielen zuvor verwendet und erwähnt und soll daher hier nur nochmals kurz erfasst werden. Verwenden Sie für Referenzparameter, welche nicht geändert werden sollen, immer das Schlüsselwort `const`. Damit stellen Sie sicher, dass diese Werte nicht versehentlich in der Funktion geändert werden können. Tun Sie es versehentlich doch, bekommen Sie eine Fehlermeldung bei der Übersetzung zurück.

Ein weiterer Vorteil von konstanten Referenzen als Funktionsparameter ist, dass Sie damit auch ein Literal (welches ja keine direkte Speicheradresse hat) oder Zwischenergebnis als Argument an die Funktion übergeben können, weil der Compiler dann eine temporäre Variable anlegen kann.

5.5.6 Default-Parameter (Standardparameter)

Bei der Deklaration des Funktionsprototyps können Sie auch Funktionsparameter mit einem Wert vorbelegen. Solche vorbelegten Werte werden als Default-Parameter (bzw. Standardparameter) bezeichnet. Default-Parameter können verwendet werden, damit beim Funktionsaufruf bspw. nicht alle Parameter explizit angegeben werden müssen. Betrachten Sie hierzu folgendes bereits bekanntes Beispiel:

```

00 // kapitel05/function10.cpp
01 #include <iostream>

02 void capacity(double radius=0.0, double high=0.0);

03 int main() {
04     double radius=0.0, high=0.0;
05     std::cout << "Radius des Kreises: ";
06     std::cin >> radius;
07     std::cout << "Höhe des Kreises  : ";
08     std::cin >> high;
09     capacity(radius);          // Kreisfläche
10     capacity(radius, high);    // Zylindervolumen
11     capacity();                // Keine Berechnung
12 }

13 void capacity(double radius, double high) {
14     const double PI = 3.141592653;
15     if(high == 0 && radius == 0) {
16         std::cout << "Beide Parameter sind 0.0!\n";
17     }
18     else if(high == 0) {
19         std::cout << "Fläche des Kreises: "
20                 << PI * radius * radius << std::endl;
21     }
22     else {
23         std::cout << "Volumen des Kreises: "
24                 << PI * radius * radius * high << std::endl;
25     }
26 }

```

Durch die Default-Parameter in Zeile 02 können Sie diese Funktion jetzt auf drei verschiedene Arten aufrufen, wie dies den Zeilen 09, 10 und 11 auch sehr schön demonstriert wird. Natürlich müssen Sie die Default-Werte nicht mit 0.0 belegen. Hier könnten Sie auch einen beliebigen für den Typ gültigen Wert verwenden. Durch die Verwendung von Default-Parametern wurde hier auch auf die Funktion `area_of_a_disk()` zur Kreisflächenberechnung verzichtet und alles in die Funktion `capacity()` gepackt. Abhängig davon, mit wie vielen Argumenten die Funktion aufgerufen wird, wird jetzt die passende Berechnung für eine Kreisfläche oder einen Zylinder durchgeführt.

Wird die Funktion ohne Argument aufgerufen (wie dies in Zeile 11 der Fall ist), sind dank der Default-Parameter beide Werte des Funktionsparameters 0.0, weshalb in der Funktion die `if`-Bedingung aus Zeile 15 gleich `true` ist und eine entsprechende Meldung auf dem Bildschirm ausgegeben wird.

Übergeben Sie der Funktion ein Argument beim Aufruf, wie in Zeile 09 geschehen, wird die Kreisfläche berechnet, weil hierbei nur der Wert von `radius` in der Funktion initialisiert wird und `high` dank des Default-Parameters gleich 0 ist. In der Funktion gibt hierbei die Bedingung aus Zeile 18 `true` zurück, und die Kreisfläche wird daher in Zeile 19 berechnet.

Vergleich auf 0

An dieser Stelle muss angemerkt werden, dass geschummelt wurde. Ein Vergleich von `double` auf 0.0, wie in den Zeilen 15 und 18, sollte man ja unterlassen, wie bereits an einer anderen Stelle im Buch vermerkt wurde, weil Fließkommazahlen eben nicht genau sind und es auch bei 0.0 zu Rundungsfehlern kommen kann. Daher müssen Sie eine Funktion verwenden, um zu prüfen, ob die Zahlen ungefähr gleich sind. Allerdings geht das Thema hier über den Rahmen des Buches hinaus und ich empfehle Ihnen, sich hierfür mit der IEEE 754-Norm zu befassen. Bezogen auf unser Beispiel könnten Sie eine solche Funktion wie folgt implementieren:

```
00 // kapitel05/function10_2.cpp
#include <cmath>
...
bool dequals( const double& a, const double& b ){
    return !(
        std::fabs(a-b)>std::numeric_limits<double>::epsilon()
    );
}
```

Und die dritte Möglichkeit ist, die Funktion ganz normal mit zwei Argumenten aufzurufen, wodurch in der Funktion das erste Argument für den `radius` und das zweite Argument für `high` verwendet wird. In der Funktion wird hierfür dann der `else`-Zweig (Zeilen 21 bis 23) aktiv, um das Volumen eines Zylinders zu berechnen.

Sie müssen nicht alle Parameter mit Default-Werten versehen. Sie müssen allerdings die Default-Parameter bei der Deklaration von rechts nach links zuordnen. Wenn hierbei ein Parameter keinen Default-Wert besitzt, kann der nächste Parameter links davon auch keinen Default-Wert mehr haben. Bezogen auf unser Beispiel wäre bspw. folgende Deklaration mit einem Default-Parameter noch möglich:

```
void capacity(double radius, double high=0.0);
```

Natürlich ändert sich hiermit auch, dass ein leerer Funktionsaufruf von `capacity()` ohne Argumente durch diese Deklaration nicht mehr möglich ist. Jetzt muss die Funktion mit mindestens einem Argument aufgerufen werden. Dieses eine Argument würde dann im Funktionskopf für `radius` verwendet und `high` hätte dann den Default-Wert 0.0. Der Aufruf mit zwei Argumenten geht sinngemäß nach wie vor.

Folgende Deklaration mit einem Default-Parameter ist aber nicht möglich, weil bei der Deklaration des Funktionsprototyps die Zuordnung der Default-Werte von rechts nach links erfolgen muss:

```
void capacity(double radius=0.0, double high); // Fehler !!!!
```

Beispiele von Funktionsprototypen mit Default-Werten

Die folgenden Beispiele von Funktionsprototypen mit Default-Werten wären somit in C++ erlaubt:

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

Die folgenden Funktionsprototypen hingegen sind alle falsch, weil mindestens einer der vorangegangenen Parameter auf der rechten Seite keinen Default-Wert enthält:

```
//!!! Alle drei Beispiele sind falsch !!!
void MyFunc ( int par1 = 11, int par2, int par3 );
void MyFunc ( int par1 = 11, int par2, int par3 = 33 );
void MyFunc ( int par1, int par2 = 22, int par3 );
```

5.6 Rückgabewert aus einer Funktion

Nachdem Sie die Schnittstelle für Daten an die Funktionen kennen, benötigen Sie jetzt noch Kenntnisse darüber, wie Sie etwas aus einer Funktion zurückgeben können. Der Rückgabewert kann auch hier von beliebigem Typ (mit Ausnahme eines C-Style-Arrays) sein.

Zurückgeben können Sie in der Regel nur einen Wert bzw. einen Typ. Dieser Wert wird dann als Rückgabewert oder Returnwert bezeichnet. Damit der Compiler weiß, welcher Wert zurückgegeben wird, müssen Sie diesen auch hier beim Funktionsprototyp entsprechend angeben. In der Funktion selbst geben Sie diesen Wert dann mit der Anweisung `return` zurück. Der Typ, den Sie mit `return` zurückgeben, muss mit dem Rückgabewert im Funktionsprototyp übereinstimmen oder zumindest in den Rückgabebetyp konvertierbar sein.

Der Aufrufer der Funktion mit Rückgabewert sollte natürlich auf die Rückgabe irgendwie reagieren. Wenn Sie bspw. etwas berechnen, dann sollten Sie den zurückgegebenen Wert bspw. mit dem Zuweisungsoperator `=` an einen passenden Typ zuweisen. Ist der Rückgabewert hingegen ein boolescher Typ, kann die Funktion auch innerhalb einer `if`-Anweisung stehen.

Verwenden wir hierzu folgendes einfache Beispiel zur Demonstration:

```

00 // kapitel05/function11.cpp
01 #include <iostream>
02 bool input(double& in);
03 double area_of_a_disk(const double& radius);

04 int main() {
05     double radius=0.0, A=0.0;
06     std::cout << "Radius des Kreises: ";
07     if( input(radius) ) {
08         A = area_of_a_disk(radius); // Kreisfläche
09         std::cout << "Kreisfläche: " << A << std::endl;
10     }
11 }

12 bool input(double& in) {
13     if( std::cin >> in ) {
14         return true;
15     }
16     else {
17         in = 0.0;
18         std::cout << "Fehler bei der Eingabe!\n";
19         return false;
20     }
21 }

22 double area_of_a_disk(const double& r) {
23     return 3.141592653 * r * r;
24 }

```

An den Funktionsprototypen in den Zeilen **02** und **03** können Sie schon erkennen, welche Typen die beiden Funktionen zurückliefern. Die Funktion `input()` liefert ein `bool` zurück, und daran sollte man sofort erkennen, dass diese Funktion etwas überprüft. In unserem Fall verwenden wir die Funktion `input()`, um auf die korrekte Eingabe von `std::cin` zu prüfen. In Zeile **07** wird diese Funktion innerhalb der `if`-Anweisung ausgeführt. Liefert also unsere Funktion `input()` gleich `true` zurück, werden die Anweisungen in den Zeilen **08** und **09** ausgeführt.

Die Funktion `input()` wurde in den Zeilen **12** bis **20** definiert und bekommt auch gleich eine Referenz als Parameter, wodurch sich Änderungen auf dem Parameter auch auf das Original in der `main`-Funktion auswirken, was wir hier ja bewusst haben wollen, weil hier in Zeile **13** die Eingabe über `std::cin` an den symbolischen Namen `in`

erfolgt, was wiederum nur ein anderer Name für die Speicheradresse `radius` (Zeile 05) ist. Je nachdem, ob die Eingabe der Funktion jetzt korrekt war, wird `true` oder `false` aus der Funktion zurückgeliefert.

Mit Referenzparameter mehr als einen Wert zurückliefern

Wenn Sie die Funktion `input()` (Zeile 12 bis 20) genauer betrachten, werden Sie feststellen, dass diese Funktion eigentlich zwei statt einen Wert aus der Funktion zurückgibt. Neben dem booleschen Typ, der unmittelbar mit `return` zurückgegeben wird, verwenden wir hier den Trick mit Referenzparameter, um mehr als nur einen Wert aus der Funktion zurückzubekommen. Sie könnten quasi mit mehreren Referenzparametern noch mehr Werte aus der Funktion zurückliefern. Hier also ein weiterer Vorteil, die Referenzen als Parameter einer Funktion zu verwenden.

Gefällt Ihnen dieser Ansatz nicht oder können Sie keine Referenzen als Argument einer Funktion verwenden, verwenden Sie ein `return` von `std::make_tuple(true, in)` aus der Headerdatei `<tuple>`, und speichern Sie das Ergebnis in einer `auto`-Variablen ab. Mehr dazu in Abschnitt 12.6.2, »Tupel, das bessere `pair` (C++11)«.

Wenn die Eingabe über `input()` erfolgreich war und `true` zurückgegeben hat, wird in Zeile 08 die Funktion zur Kreisflächenberechnung durchgeführt. Als Parameter erhält diese Funktion den zuvor erfolgreich eingelesenen Wert für `radius`. Den Rückgabewert, welcher ja als `double` deklariert wurde, weisen wir hier der Variablen `A` zu. In der Funktionsdefinition (Zeile 22 bis 24) wird der übergebene Wert in einer einfachen Kreisflächenberechnung in Zeile 23 direkt über `return` an den Aufrufer zurückgegeben, so dass die Variable `A` in Zeile 08 die aus `radius` berechnete Kreisfläche enthält und in Zeile 09 ausgeben kann.

Im Beispiel hätten Sie auch auf die Zwischenspeicherung in `A` (Zeile 08) verzichten können und die Ausgabe gleich über `std::cout` wie folgt auf dem Bildschirm ausgeben können:

```
std::cout << "Kreisfläche: " << area_of_a_disk(radius);
```

Die `return`-Anweisung

Mit einer `return`-Anweisung wird eine Funktion sofort beendet und kehrt zum Aufrufer der Funktion zurück. Verwenden Sie die `return`-Anweisung in der `main`-Funktion, wird das Programm beendet. `return`-Anweisungen können Sie auf unterschiedliche Arten benutzen. Hier die Syntax dazu:

```
return;           // Beendet Funktion ohne Rückgabewert
return ausdruck;  // Gibt ausdruck an den Aufrufer zurück
return (ausdruck); // Gibt ausdruck an den Aufrufer zurück
```

Unterscheidet sich der Typ von `ausdruck` vom erwarteten Wert des Typs, der einer Variablen zugewiesen wird, findet eine implizite Typumwandlung statt, wie Sie sie aus Abschnitt 2.9.1, »Automatische (implizite) Typumwandlung«, kennen.

Weiteres zu `return`

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

5.6.1 Referenz als Rückgabewert

Auch Referenzen lassen sich als Rückgabewert einer Funktion verwenden. Allerdings müssen Sie hier sehr sorgfältig darauf achten, was Sie da zurückgeben. Wie bereits erwähnt, werden bei einem Funktionsaufruf alle lokalen Daten auf einen Stapelspeicher (Stack) gelegt und wieder zerstört, wenn die Funktion beendet wird. Was hier dann zum Problem werden kann, soll das folgende Beispiel zeigen:

```
double& area_of_a_disk(double r) {
    double Area = r * r * 3.141592653;
    return Area // Refernz von lokalem Stack wird zurückgegeben
}
...
double& A = area_of_a_disk(radius);
// Hier wurde der lokale Stack bereits wieder freigegeben !!!
```

Das Problem an diesem Beispiel ist, dass die Funktion jetzt eine Referenz auf einer lokalen (!) Variablen zurückgibt. Die Variable `Area` ist nur zur Laufzeit der Funktion `area_of_a_disk()` gültig, wenn diese auf dem lokalen Stapelspeicher (Stack) liegt. Der zurückgegebene Wert ist somit ungültig, auch wenn vielleicht zunächst ein korrekter Wert zurückgegeben wird. Früher oder später wird dieser Speicherbereich allerdings überschrieben. Im konkreten Fall wäre die beste Möglichkeit, eine Kopie zurückzugeben:

```
double area_of_a_disk(double r);
```

Referenzen bei Operatorüberladung

Sie kennen jetzt die Tücken, wenn Sie Referenzen als Rückgabewert von gewöhnlichen Funktionen verwenden. Allerdings werden Sie auch später bei der Operatorüberladung (Kapitel 8, »Operatoren überladen«) Beispiele sehen, bei denen Sie die Referenzen als Rückgabewert sehr zu schätzen wissen werden.

5.6.2 Rohe Zeiger als Rückgabewert

Wie Referenzen lassen sich selbstverständlich auch rohe Zeiger aus Funktionen zurückgeben. Allerdings auch mit denselben Tücken, dass lokale Daten nicht aus Funktionen zurückgegeben werden dürfen. Die Anfangsadresse in Form von Zeigern zurückzugeben, mag sich auf den ersten Blick vielleicht ganz praktisch anhören, aber es ist trotzdem, wie alles mit Verwendung von rohen Zeigern, eher weniger in modernem C++ vertreten. C++11 bietet hier bspw. mit der neuen Move-Semantik eine wesentlich bessere Möglichkeit zur Rückgabe von umfangreichen Daten an. Im nächsten Abschnitt lernen Sie einige Tücken kennen, welche teilweise unmittelbar in Verbindung mit rohen Zeigern als Rückgabewert stehen.

5.6.3 Dinge, die man besser nicht zurückgibt

Zwar wurde es bereits erwähnt, dass es zu undefinierten Ergebnissen kommt, wenn Sie lokale Daten aus einer Funktion zurückgeben. Trotzdem will ich Sie hier nochmals etwas für das Thema sensibilisieren, weil damit in der Vergangenheit doch relativ viel Unfug getrieben wurde.

Mit der Rückgabe von lokalen Werten aus Funktionen verhält es sich wie mit russischem Roulette – für Programmierer also ein unerwünschtes Übel. Gerade Einsteiger wissen nicht immer 100 %ig, was alles an einer Funktion »lokal« ist. Im Grunde ist alles in einer Funktion »lokal«, was auf dem Stapelspeicher (Stack) liegt. Neben in Funktionen definierten Variablen sind dies auch die Parameter, wenn Sie diese *by Value*, also als Kopie, übergeben. Die lokalen Daten selbst sind allerdings zunächst kein Problem. Probleme treten im Grunde erst auf, wenn man die Speicheradressen dieser lokalen Daten an den Aufrufer zurückgibt. Nach dem Funktionsaufruf gibt es diese Adressen nämlich nicht mehr, und der Aufrufer verwendet dann quasi undefinierte Werte, mit denen vielleicht noch weitergearbeitet werden kann. Aber sobald diese Speicheradresse erneut verwendet wird, geht die tickende Zeitbombe hoch, wie folgendes Beispiel demonstriert:

```
...
01 std::vector<int>* n_init(const int &n, const int& val) {
02     std::vector<int> lokal_vec(n);
03     for(size_t i=0; i<lokal_vec.size(); ++i) {
04         lokal_vec.at(i) = val;
05     }
06     return (&lokal_vec);
07 }
...
08 std::vector<int>* daten = n_init(10, 123);
```


Die Funktion `n_init()` sollte eigentlich eine bestimmte Anzahl an Elementen vom Typ `vector<int>` anlegen, mit dem Wert `val` initialisieren und wieder zurückgeben. Das Dilemma geht aber in der Funktion in Zeile 02 los, wo wir ein lokales `int`-Array mit `n`-Elementen mithilfe von `vector` anlegen. Sie wissen ja jetzt, dass diese Werte bzw. Speicheradressen nur zur Laufzeit der Funktion auf dem Stapelspeicher existieren. An sich ist das ja hier noch kein Fehler. Anschließend werden alle Werte mit `val` belegt (Zeilen 03 bis 05). Jetzt wird in Zeile 06 die Anfangsadresse des `vector<int>`-Arrays dem Aufrufer zurückgegeben. Der Zeiger `daten` in Zeile 08 verweist jetzt auf die Anfangsadresse des Speicherbereichs, welche ja offiziell jetzt gar nicht mehr existiert, weil mit Beendigung der Funktion auch `lokal_vec` aus Zeile 02 auf dem Stapelspeicher zerstört wurde.

Zugegeben, ein solche Beispiel lässt echte C++-Puristen erschauern, aber uns geht es ja hier um Dinge, wovon man besser die Finger lässt. Das Beispiel zeigt sehr schön das Problem, wenn man bei Funktionen mit der Rückgabe von lokalen Speicheradressen hantiert. Bezogen auf dieses Beispiel gäbe es jetzt zwei Möglichkeiten, welche auch nicht unbedingt »schön« sind, aber zumindest den lokalen Gültigkeitsbereich »umgehen«. Neben der wohl schlechtesten Möglichkeit, eine globale Variable zu verwenden, gäbe es noch folgende:

- Sie fordern mit `new` vom Heap einen Speicher zur Laufzeit an (siehe Abschnitt »Speicherplatz zur Laufzeit anfordern (C++03)« in Abschnitt 4.3). Auch Speicheradressen vom Heap sind zur kompletten Laufzeit des Programms vorhanden. Allerdings steht man dann vor dem Problem der Freigabe des Speichers mit `delete`. Alternativ könnten Sie hier auch die neuen `std::unique_ptr` (siehe Abschnitt 16.3.3, »Unique Pointer«) aus C++11 verwenden.
- Sie deklarieren den lokalen Typ aus dem Listing in Zeile 02 mit dem Schlüsselwort `static`, womit der Typ ebenfalls nicht mehr im Stapelspeicher (= Stack), sondern im immer vorhandenen Datensegment gespeichert wird (siehe Abschnitt 6.2.2, »Das Schlüsselwort `static`«).

Der folgende Codeausschnitt demonstriert Ihnen die Verwendung von `new`:

```
01  std::vector<int>* n_init(const int &n, const int& val) {
02      std::vector<int>* lokal_vec=new std::vector<int>(n);
03      for(size_t i=0; i<lokal_vec->size(); ++i) {
04          lokal_vec->at(i) = val;
05      }
06      return (lokal_vec);
07  }
...
08  std::vector<int>* daten = n_init(10, 123);
```

In Zeile **02** wurde jetzt zur Laufzeit Speicher vom Heap angefordert und die Anfangsadresse in Zeile **06** zurückgegeben. Wie bereits eben erwähnt, ist dies so programmtechnisch in Ordnung, weil die Speicheradresse auch nach dem Funktionsaufruf noch vorhanden ist. Allerdings birgt dieses Beispiel dann wieder ein anderes Problem, und zwar: Wo wird der Speicher wieder freigegeben (Stichwort Memory Leak)? So geschrieben, werden Sie quasi gezwungen, den Speicher an einer anderen Stelle im Programm wieder freizugeben. Die Vergangenheit hat gezeigt, dass so etwas gerne vernachlässigt wurde. Außerdem ist dieser rohe Programmierstil wiederum weniger C++-like!

Rückgabe mit der neuen Move-Semantik (C++11)

Neben den oben vorgestellten Möglichkeiten gibt es gerade für Standardcontainer eine noch viel bessere Möglichkeit mit der neuen Move-Semantik seit C++11, mit der der Rückgabewert »verschoben« wird. Das Tolle an der Sache ist, dass Sie oftmals gar nichts am Code ändern müssen, sondern das Programm einfach nur neu mit einem C++11-Compiler übersetzen müssen. Folgender Codeausschnitt soll Ihnen zeigen, was sich dahinter verbirgt:

```
01 std::vector<int> createVec(int n) {
02     std::vector<int> ivec(n);
...
03     return ivec;// hier kommt die Move-Semantik zum Einsatz
04 }
...
05 std::vector<int> data = createVec(10);
```

Dank der Tatsache, dass der Funktionsaufruf in Zeile **05** ein *RValue* ist und dass seit C++11 in allen Standardcontainern die Move-Semantik implementiert wurde, wird in Zeile **03** keine Kopie mehr (wie in C++03) zurückgegeben, sondern besagte Move-Semantik (Verschiebe-Technik) kommt zum Einsatz. Hierfür werden allerdings noch tiefgreifende Kenntnisse benötigt. In Abschnitt 16.1, »Move-Semantik und Perfect Forwarding«, wird diese neue Technik umfassend behandelt.

5.6.4 Mehrere Werte zurückgeben – `std::pair<>`

Sie haben bereits gesehen, wie Sie mithilfe eines Tricks von Referenzen als Funktionsparametern mehrere Werte aus einer Funktion »zurückgeben« können (siehe Abschnitt 5.6.1, »Referenz als Rückgabewert«). Diese Werte wurden nicht direkt mit `return` zurückgegeben, sondern wir nutzen es lediglich aus, dass eine Änderung von Referenzen innerhalb der Funktion (da ja eine Speicheradresse übergeben wurde) auch noch außerhalb der Funktion gültig war.

Über eine Struktur oder gar einer Klasse können Sie ebenfalls mehrere Daten zusammenpacken und so mehrere Werte aus einer Funktion zurückgeben.

In C++ gibt es auch noch mit `std::pair<>` ein spezielles **Hilfsmittel**, um zwei Typen aus einer Funktion direkt über die `return`-Anweisung zurückzugeben. `std::pair` ist in der Headerdatei `<utility>` definiert, welche Sie auch mit angeben müssen, wenn Sie `std::pair<>` verwenden. Die Verwendung von `std::pair<>` ist denkbar einfach, was das folgende Beispiel demonstrieren soll:

```
00 // kapitel05/function12.cpp
01 #include <iostream>
02 #include <string>
03 #include <utility> // für std::pair<>

04 std::pair<std::string, int> simple_pair();

05 int main() {
06     std::pair<std::string, int> daten = simple_pair();
07     std::cout << "Monat      : " << daten.first << std::endl;
08     std::cout << "Temperatur : " << daten.second
09               << std::endl;
10 }

10 std::pair<std::string, int> simple_pair() {
11     std::string monat;
12     int temperatur;
13     std::cout << "Monat : "; std::cin >> monat;
14     std::cout << "Temperatur : "; std::cin >> temperatur;
15     return make_pair(monat, temperatur);
16 }
```

An der Deklaration der Funktion in Zeile **04** können Sie schon anhand des Rückgabewertes erkennen, dass diese Funktion einen `std::string` und ein `int` zurückliefert. Das ist im Grunde alles, was Sie tun müssen. Sie müssen lediglich zwischen den spitzen Klammern die zwei Typen eingeben (getrennt durch ein Komma), welche zurückgegeben werden sollen. In Zeile **06** rufen wir die Funktion auf und weisen den Rückgabewert ebenfalls einem `std::pair<std::string, int>` zu. In der Funktionsdefinition (Zeile **10** bis **16**) werden Sie aufgefordert, jeweils einen `std::string` und ein `int` einzugeben (hier als Wetterdaten). In Zeile **15** verwenden wir die Funktion `make_pair()`, um aus den beiden eingegebenen Daten ein `std::pair<std::string, int>` zu machen und auch gleich zurückzugeben. Natürlich hätten Sie auch eine lokale Variable wie

```
std::pair<std::string, int> daten(monat, temperatur);
return daten;
```

verwenden und zurückgeben können, aber mit `make_pair()` geht das gleich in einem Rutsch an den Aufrufer zurück. Auf die einzelnen Werte eines `std::pair<>` greifen Sie jetzt mit `first` (für den ersten Wert) und `second` (für den zweiten Wert) zu, wie Sie dies auch in den Zeilen 07 bis 08 sehr schön sehen können.

std::tuple, das bessere std::pair (C++11)

Neu in C++11 wurde `std::tuple` eingeführt, es kann im Gegensatz zu `std::pair` beliebig viele unterschiedliche Typen als Elemente speichern. Mehr zu `std::tuple` finden Sie in Abschnitt 12.6.2, »Tupel, das bessere pair (C++11)«.

5.7 Funktionen überladen

Dem Thema der Überladung werden Sie noch des Öfteren in diesem Buch begegnen. Den Anfang machen wir mit der Funktionsüberladung. Von einer Überladung (engl. *overloading*) spricht man, wenn mehrere Versionen mit demselben Bezeichner gleichzeitig vorhanden und sichtbar sind und die Auswahl des richtigen Bezeichners anhand der Signatur erfolgt. Die Entscheidung, welchen Bezeichner der Compiler wählt, hängt dann von der Signatur der Funktion ab, welche beim Aufruf verwendet wird. Der Compiler vergleicht die Signatur des Aufrufes mit der Signatur der überladenen Funktionen und wählt die passende zur Übersetzungszeit aus. Die passende Signatur erkennt der Compiler anhand des Funktionsnamens und der Typen der Parameter. Das Überladen von Funktionen wird auch als *Funktionspolymorphie* (oder nach abstrakter: *Ad-hoc-Polymorphie*) bezeichnet.

Betrachten Sie bspw. folgende Deklarationen von Funktionsprototypen (die Definition lassen wir jetzt noch außen vor):

```
int simple_func ( int ivar );
int simple_func ( int ivar1, int ivar2 );
double simple_func ( double dvar );
double simple_func ( double dvar1, double dvar2 );
```

Die Technik, welche der Compiler jetzt verwendet, um die richtige Funktion auszuwählen, ist folgendermaßen:

1. Der Compiler findet eine Funktion, die perfekt auf eine Signatur passt, und muss somit keinerlei Anpassungen durchführen.
2. Der Compiler sucht nach einer Funktion, mit der er eine einfache integrale Promotion durchführen kann (bspw. von `bool` oder `char` nach `int`).
3. Als Nächstes versucht der Compiler, eine Standard-Typumwandlung durchzuführen, um eine vorhandene Funktion zu finden.

4. Zum Schluss sucht der Compiler nach einer benutzerdefinierten Typumwandlung für den Funktionsaufruf.

Das folgende Beispiel soll noch eine kleine Erweiterung zu Listing *function12.cpp* sein, wo Sie jetzt die höhere Temperatur von zwei übergebenen Werten zurückbekommen. Die Funktion `simple_pair()` wird hier nicht abgedruckt und kann aus dem Beispiel in Listing *function12.cpp* entnommen werden. Hierbei können Sie die Funktion einmal mit `std::pair<std::string, int>` und einmal nur mit `int` als Parameter aufrufen. Hier das Beispiel:

```
00 // kapitel05/function13.cpp
01 #include <iostream>
02 #include <string>
03 #include <utility>

04 std::pair<std::string, int> simple_pair();
05 std::pair<std::string, int> maxTemp(
    std::pair<std::string, int> d1,
    std::pair<std::string, int> d2 );
06 int maxTemp(int temp1, int temp2);

07 int main() {
08     std::pair<std::string, int> daten1 = simple_pair();
09     std::pair<std::string, int> daten2 = simple_pair();

10     std::pair<std::string, int> maxT =
        maxTemp(daten1, daten2);
11     std::cout << "Der wärmere Monat: " << maxT.first
        << std::endl;
12     std::cout << "Die wärmere Temperatur : "
        << maxTemp(daten1.second, daten2.second)
        << std::endl;
13 }
...
...
14 std::pair<std::string, int> maxTemp(
    const std::pair<std::string, int> d1,
    const std::pair<std::string, int> d2 ) {
15     if(d1.second > d2.second) {
16         return d1;
17     }
18     return d2;
19 }
```

```

20 int maxTemp(int temp1, int temp2) {
21     return temp1 > temp2 ?temp1 :temp2;
22 }

```

Bei der Deklaration der Funktionsprototypen in den Zeilen 05 und 06 finden Sie zwei Funktionen `maxTemp()` mit unterschiedlichen Parametern und somit unterschiedlichen Signaturen, eine klassische Überladung. In Zeile 10 wird die Überladung mit `std::pair<std::string, int>` als Parameter verwendet. Der Compiler kümmert sich anhand der Signatur um den richtigen Funktionsaufruf. Die Funktion wird in den Zeilen 14 bis 19 definiert und vergleicht die Temperaturdaten zweier `std::pair<std::string, int>`-Werte und gibt den Typ zurück, dessen Wert höher ist. Der Wert befindet sich an zweiter Stelle im `std::pair<std::string, int>` und wird mit `second` angesprochen.

Die zweite überladene Funktion `maxTemp()` rufen wir in Zeile 12 auf. Auch hier erkennt der Compiler anhand des Aufrufes, dass hier die Signatur mit `maxTemp(int, int)` verwendet werden muss. Mit `daten1.second` bzw. `daten2.second` übergeben wir der Funktion ja direkt die `int`-Werte. Auch hier liefert die Funktion, welche in Zeile 20 bis 22 definiert ist, den größeren der beiden Werte als Integer zurück.

Selbstverständlich können Sie noch weitere Überladungen schreiben, bei denen bspw. die Parametertypen gemixt werden, wie etwa folgende Funktionsprototypen demonstrieren sollen:

```

int maxTemp( const std::pair<std::string, int> d1,
             const int d2 );
int maxTemp( const int d1,
             const std::pair<std::string, int> d2 );

```

Die Funktionsdefinition müssen Sie natürlich auch entsprechend schreiben.

5.8 Spezielle Funktionen

An dieser Stelle sollen noch einige spezielle Dinge behandelt werden, welche mit Funktionen zu tun haben oder sich einfach nirgends so recht reinschieben lassen.

5.8.1 Inline-Funktionen

Bei jeder aufgerufenen Funktion wird ein Stack-Rahmen angelegt, in dem die Funktion ihre Daten ablegt. Darin finden Sie auch die Rücksprungadresse gespeichert, damit die Funktion am Ende auch wieder weiß, von wo Sie aufgerufen wurde, und damit Sie auch wieder dorthin zurückgelangt. Zusätzlich wird der Stack-Rahmen auch dafür verwendet, lokale Daten, lokale Parameter und den Rückgabebetyp abzulegen. Ein ziemlicher Aufwand für einen einfachen Funktionsaufruf.

Für solche Zwecke können Sie eine Funktion mit dem Schlüsselwort `inline` kennzeichnen:

```
inline int simple_func(double a, double b, double c);
```

Mit dem Schlüsselwort `inline` vor der Funktion schlagen Sie dem Compiler vor, dass er den Code der Funktion nicht als aufrufbare Funktion integrieren soll, sondern dass der Code an Ort und Stelle des Funktionsaufrufes eingesetzt werden soll.

Nur ein Vorschlag für den Compiler!

Die Rede ist hier von »soll«, weil es eben nur ein Vorschlag für den Compiler ist. Ob der Compiler die Funktion an Ort und Stelle integriert oder nicht, entscheidet dieser letztendlich wieder selbst.

5.8.2 Lambda-Funktionen (C++11)

Die Lambda-Funktionen wurden mit dem Hintergedanken eingeführt, kleinere anonyme Funktionen an Ort und Stelle einzubauen. Etwas Ähnliches bewirkt zwar auch eine Funktion mit dem Schlüsselwort `inline`, aber hier entscheidet letztendlich der Compiler, ob er die Funktion tatsächlich an Ort und Stelle integriert.

In der Praxis sind solche Lambda-Funktionen sehr hilfreich, wenn Sie Algorithmen der Standardbibliothek oder Multithreads verwenden. Daher wird nochmals umfassend auf diese Lambda-Funktionen in Abschnitt 12.5.6, »Lambda-Funktion (C++11)«, eingegangen. Trotzdem hier schon mal ein kurzes Beispiel, welches eine solche Lambda-Funktion im Einsatz zeigen soll:

```
00 // kapitel05/function14.cpp
01 #include <iostream>
02 #include <algorithm> // for_each
03 #include <vector>

04 int main() {
05     std::vector<int> vec{12, 34, 13, 45, 38, 27, 11};
06     int count{0};
07     for_each(begin(vec), end(vec), [&count] (int val) {
08         if( (val%2) ) ++count;
09     });
10     std::cout << "vec enthält " << count
11               << " gerade Zahlen\n";
11 }
```

Die Lambda-Funktion wird hier fett gedruckt. In diesem Beispiel wurde außerdem mit `count` eine lokale Variable dem Lambda sichtbar gemacht. Hier wird eine Referenz verwendet, weil sonst eine Kopie übergeben wird, welche außerhalb der Lambda-Funktion ihre Gültigkeit verliert. Das Beispiel macht im Grunde nichts anderes, als für jedes Element (= `for_each`) vom Anfang (= `begin(vec)`) bis zum Ende (= `end(vec)`) die Lambda-Funktion aufzurufen. Der aktuelle Wert wird dabei an den Parameter `val` übergeben. In der Lambda-Funktion wird überprüft, ob der Wert gerade oder ungerade ist. Ist er gerade, inkrementieren wir den Wert von `count`, welcher die Anzahl der geraden Zahlen mitzählt, die in `vec` enthalten sind.

Die Syntax der Lambda-Funktion erscheint auf den ersten Blick etwas kryptisch, aber wenn man sich daran gewöhnt hat, will man diese nicht mehr missen:

```
[capture] (parameter) -> return-value { anweisungen; }
```

Erklärung verschafft Ihnen Tabelle 5.1.

Bereich	Bedeutung
[]	Innerhalb der eckigen Klammern wird die Bindung einer Variablen im lokalen Bereich angegeben. Mit [=] werden Werte kopiert, mit [&] referenziert und mit [] erfolgt überhaupt keine Bindung.
()	Darin können Sie die optionalen Argumente schreiben, welche anschließend im Anweisungsblock verwendet werden.
->	Optional kann so ein Rückgabetyt angegeben werden.
{ }	Zwischen den geschweiften Klammern werden wie gewöhnlich die Anweisungen zu einem Block zusammengefasst.

Tabelle 5.1 Die Bestandteile einer Lambda-Funktion

Zum Weiterlesen

Etwas intensiver wird die Lambda-Funktion in Abschnitt 12.5.6, »Lambda-Funktion (C++11)«, behandelt.

5.8.3 **main()-Funktion**

Jedes ausführbare Programm muss eine Funktion mit dem Namen `main()` besitzen. Auch wenn es klar sein sollte, es darf nur eine Funktion mit dem Namen `main()` geben. Nach dem Standard erwartet die `main`-Funktion einen Rückgabewert vom Typ `int`. Ein korrektes Grundgerüst wäre somit:


```
int main() {
    return 0; // muss nicht explizit verwendet werden
}
```

Impliziter Rückgabewert aus main()

Die Rückgabe an die main-Funktion muss nicht unbedingt explizit über `return 0` erfolgen und kann auch weggelassen werden. Wenn das Blockende von `main()` erreicht wird, wird in diesem Fall implizit 0 als Rückgabewert verwendet.

Welchen Rückgabewert verwenden?

Der Rückgabewert bei der Beendigung eines Programms ist abhängig von der Umgebung des Betriebssystems. Meistens bedeutet ein Rückgabewert von 0, dass das Programm ordentlich beendet wurde. Ein anderer Wert als 0 wird meistens als Fehler gedeutet. Um sich nicht mit systemabhängigen Dingen herumschlagen zu müssen, könnten Sie auch die Headerdatei `<cstdlib>` hinzufügen und statt 0 den Wert `EXIT_SUCCESS` für eine ordentliche Beendigung oder `EXIT_FAILURE` für eine fehlerhafte Beendigung des Programms verwenden.

Kommandozeilenargumente an das Programm übergeben

Es ist auch eine `main`-Funktion mit zwei Parametern erlaubt, wobei Sie dann theoretisch Argumente von der Kommandozeile aus an das Programm übergeben können. Ein einfaches Beispiel hierzu:

```
00 // kapitel05/function15.cpp
01 #include <iostream>

02 int main(int argc, char* argv[]) {
03     for(int i = 0; i < argc; ++i) {
04         std::cout << i+1 << ".Argument: " << argv[i]
05             << std::endl;
06     }
07 }
```

Im ersten Argument der `main`-Funktion finden Sie in `argc` die Anzahl der in der Kommandozeile übergebenen Argumente. Der Wert hierbei ist immer mindestens 1, weil das erste Argument (`argv[0]`) immer der C-Style-String des Programmaufrufes ist. Bei `char* argv[]` handelt es sich um einen Vektor mit Zeiger auf C-Style-Strings. Die weiteren Argumente, welche Sie, durch ein Leerzeichen getrennt, in der Kommandozeile übergeben haben, befinden sich dann in `argv[1]`, `argv[2]` usw. – je nachdem, wie viele Argumente Sie übergeben haben. Starten Sie das Programm bspw. in der Kommandozeile so

J.Wolf \$ `./function15 C++ von A bis Z`

dann sieht die Ausgabe des Programms wie folgt aus:

```
1.Argument: ./function15
2.Argument: C++
3.Argument: von
4.Argument: A
5.Argument: bis
6.Argument: Z
```

5.8.4 Funktions- und Programmende

An dieser Stelle sollen noch einige Möglichkeiten aufgezählt werden, mit denen Sie eine Funktion oder gar das komplette Programm beenden können:

- ▶ Mit der `return`-Funktion wird eine Funktion beendet. Innerhalb der `main`-Funktion bedeutet ein `return` gleichzeitig auch das Ende des Programms. Innerhalb einer Funktion wird mit `return` zur Adresse des Aufrufers zurückgesprungen.
- ▶ Verwenden Sie die Funktion `void exit(int val);` aus der Headerdatei `<cstdlib>`, wird das Programm mit dem übergebenen Wert `val` an jeder beliebigen Stelle im Programm beendet. Allerdings garantiert dieses Ende kein »Saubermachen« von lokalen Daten mehr. Hier können Sie auch jederzeit wieder `EXIT_FAILURE` bzw. `EXIT_SUCCESS` für `val` verwenden.
- ▶ Sehr ruppig können Sie ein Programm mit der Funktion `void abort();` ebenfalls aus dem Header `<cstdlib>` beenden. Ein »Saubermachen« findet hier dann gar nicht mehr statt. Hier kann allerdings nicht genau vorausgesagt werden, wie das System nach dem Ende des Programms reagiert, weil dies systemabhängig ist.
- ▶ Ein Programm kann auch mit Ausnahmen (selbst ausgelöst oder unerwartete) beendet werden. Ausnahmen können allerdings abgefangen und behandelt werden. Wie dies geht, erfahren Sie in Kapitel 10, »Ausnahmebehandlung (Exceptions)«, des Buches.
- ▶ Dann gibt es noch die unerwarteten Fehler, die vom Programmierer oder Anwender produziert wurden, welche das Programm (unerwartet) beenden können. Gängige Fehler sind hierbei Speicherschutzverletzungen, welche vom Programmierer behoben werden müssen.

5.9 Die neue Funktionssyntax (C++11)

Neben einer neuen Funktionssyntax wurden in C++11 auch noch einige neue Dinge eingeführt, welche sich u.a. auch hervorragend (aber nicht nur) mit Funktionen ver-

wenden lassen. In diesem Abschnitt werden die meisten Dinge nur kurz abgehandelt, weil diese im Zusammenhang mit einfachen Funktionen doch eher trivial sind und erst später bei Elementfunktionen (Methoden) von Klassen sinnvoller sind.

Zum Weiterlesen

Einen sinnvolleren Einsatz in der Praxis, wie Ihnen die neuen Techniken wirklich das Leben vereinfachen können und auch das »Warum«, können Sie in Abschnitt 7.7.11, »Die neue Funktionssyntax – decltype und auto (C++11)«, nachlesen.

5.9.1 Die neue Rückgabesyntax

Als Autor ist man immer gewillt, nur Fakten zu liefern und Dinge nicht zu hinterfragen. Aber mein erster Eindruck zur Einführung der neuen Funktionssyntax war zunächst schon, warum man nach gut vier Jahrzehnten (seit Entstehen von C eingeschlossen) eine neue Funktionssyntax eingeführt hat, welche ja nicht abwärtskompatibel ist. Am Ende aber, wenn man es als Gesamtpaket betrachtet und die neue Syntax bei den Methoden von Klassen verwendet (siehe Abschnitt 7.7.11, »Die neue Funktionssyntax – decltype und auto (C++11)«), sehe ich sie schon als Gewinn, weil sich damit der Code mindestens besser lesen lässt. An manchen Stellen lassen sich auch komplexere Sachen einfacher realisieren.

Geändert hat sich zunächst einmal nur, dass der Rückgabewert an das Ende des Funktionskopfes verschoben wurde und zwischen der Liste der Parameter und dem Rückgabewert durch einen Pfeil getrennt ist. Außerdem muss hierbei das neue Schlüsselwort `auto` vor den Funktionsnamen gestellt werden. Folgende Deklarationen zeigen einmal die alte und einmal die neue Syntax:

```
// Alte Syntax:
double addition( double val1, int val2 );
// Neue Syntax:
auto addition( double val1, int val2 ) -> double;
```

Das Schlüsselwort `auto` am Anfang ist unbedingt notwendig, um Mehrdeutigkeiten auszuschließen. Bei der Funktionsdefinition muss natürlich dieselbe Syntax verwendet werden:

```
00 // kapitel05/function16.cpp
01 #include <iostream>

02 auto addition( double val1, int val2 ) -> double;

03 int main() {
```

```

04     std::cout << addition(123.123, 234) << std::endl;
05 }

06 auto addition( double val1, int val2 ) -> double {
07     return val1+val2;
08 }

```

In diesem Beispiel ist die neue Funktionssyntax doch recht trivial, und eine wirkliche Vereinfachung lässt sich da kaum erkennen. Aber die Übersichtlichkeit wird erhöht, wenn viele Funktionsnamen, und dann auch noch mit dem Bereichsoperator, geschrieben stehen:

```

// Alte Syntax:
std::vector<double> get_val();
std::pair<std::string, int> simple_pair();
std::vector<std::string> get_simple_text();
std::string real_line();

// Neue Syntax:
auto get_val() -> std::vector<double>;
auto simple_pair() -> std::pair<std::string, int>;
auto get_simple_text() -> std::vector<std::string>;
auto real_line() -> std::string;

```

Hier finde ich, lässt sich der Rückgabotyp durch die neue Funktionssyntax doch schon besser und schneller herauslesen als noch mit der alten Syntax. Und dabei sind diese Beispiele noch relativ einfach gehalten.

5.9.2 decltype

Das Beispiel aus Listing *funktion16.cpp* soll jetzt nochmals herangezogen werden, um auch das neue `decltype` etwas zu beschreiben. Mithilfe von `decltype` können Sie Typinformationen aus einem Ausdruck herausziehen. Betrachten Sie hierfür das folgende Beispiel:

```

RETVALUE addition( unsigned int val1, long val2 ) {
    return val1+val2;
}

```

Welchen Typ würden Sie jetzt für **RETVALUE** verwenden? Schließlich hängt die tatsächliche Größe von `long` und `unsigned int` vom System und der Implementierung ab. Mit dem neuen `decltype` machen Sie es sich jetzt einfach und lassen den Compiler den Typ ermitteln. In der Praxis (inklusive der neuen Funktionssyntax) sieht dies dann wie folgt aus:

```

auto addition( unsigned int val1, long val2 ) ->
    decltype(val1+val2) {
    return val1+val2;
}

```

Zugegeben, dieses Beispiel ist ebenfalls noch recht trivial und einfach. Aber spätestens beim Einsatz von Templates können Sie praktisch ein Funktion-Template schreiben, welches Sie mit fast jedem beliebigen Typ aufrufen können. Eine Kostprobe gefällig?

```

template<typename T1=int, typename T2=int>
auto addition( T1 val1, T2 val2 ) -> decltype(val1+val2) {
    return val1+val2;
}

```

Dieses Template können Sie jetzt mit beliebigen Typen (bspw. `addition(double, int)`, `addition(int, double)`, `addition(short, float)` usw.) aufrufen. Mithilfe des Schlüsselwortes `auto`, der neuen Rückgabesyntax und `decltype` brauchen Sie sich nicht um den Rückgabotyp zu kümmern. Bei gemischten Typen wie `addition(short, double)` wird der größere der Typen verwendet. Im Beispiel wäre das `double`. Sofern der `+`-Operator für den Typ vorhanden ist, funktioniert das mit diesen Typen auch. Selbst ein Aufruf von `addieren(string("wort1"),string("wort2"))` würde funktionieren und diese beiden Strings als einen ganzen String wieder zurückgeben.

Das Ganze funktioniert auch auf fremde Schnittstellen, bei denen Sie vielleicht nicht wissen, was für ein Typ zurückgegeben wird, oder bei denen Sie einfach keine Lust haben, nachzusehen. Auch hier können Sie mithilfe von `decltype` die Arbeit an den Compiler delegieren:

```

00 // kapitel05/function17.cpp
01 #include <iostream>
02 #include <ctime>

03 auto getTstamp( ) -> decltype(time(nullptr)){
04     return time(nullptr);
05 }

06 int main() {
07     decltype(time(nullptr)) tstamp = getTstamp();
08     std::cout << tstamp << std::endl;
09 }

```

Hier behelfen wir uns gleich zweimal mit `decltype`, einmal als Funktionsrückgabewert in Zeile 03 und einmal, um den Typ der Variablen `tstamp` (Zeile 07) ermitteln zu

lassen. Hier wurde die Uralt-Funktion `time()` aus der Headerdatei `<ctime>` verwendet, deren Rückgabetyt mir jetzt nicht einfallen wollte (es war `clock_t`, aber wer hat schon alle Typen im Kopf). Den `decltype` in Zeile 07 hätten wir uns aber auch sparen und stattdessen einfach `auto` vor `tstamp` setzen können:

```
07  auto tstamp = getTstamp();
```

Automatische Breite auswählen lassen

Dem Ganzen lässt sich noch eines draufsetzen. Mit `decltype` können Sie sich vom Compiler passend zu einem Literal den kleinstmöglichen Integer-Typ für den Parameter und den Rückgabewert wählen lassen. Hört sich schlimmer an, als es ist. Betrachten Sie folgendes Beispiel:

```
00  // kapitel05/function18.cpp
01  #include <iostream>

02  auto dekrement(
    decltype(987654321098765432) ivar ) -> decltype(ivar) {
03      return ivar-1;
04  }

05  auto inkrement( decltype(987654321) ivar ) ->
    decltype(ivar) {
06      return ivar+1;
07  }

08  int main() {
09      std::cout << dekrement(123456789012) << std::endl;
10      std::cout << inkrement(123456789012) // Fehler
        << std::endl;
11  }
```

Mit der Angabe in Zeile 02 mit `decltype(987654321098765432) ivar` sucht der Compiler für Sie den kleinstmöglichen passenden Integer-Typ und bildet daraus auch gleich den Rückgabewert. Selbiges geschieht auch in Zeile 05, nur benötigen wir hier nicht einen so großen Integer-Typ. Den Beweis sollen die Zeilen 09 und 10 liefern. Während der Funktionsaufruf in Zeile 09 noch problemlos funktioniert, meldet der Compiler **höchstwahrscheinlich** in Zeile 10 eine Überlauf-Warnung. Kein Wunder, denn das übergebene Argument in Zeile 10 ist größer, als es mit dem Literal `decltype(987654321) ivar` angegeben wurde, und somit passt der Aufrufparameter nicht mehr in den `int`-Typ hinein, der für `decltype()` vom Compiler ausgewählt wurde.

5.10 Ausblick

In diesem Kapitel haben Sie viel über Funktionen und darum herum erfahren. Das Wissen wird Ihnen helfen, wenn Sie später Methoden (auch Elementfunktionen oder Memberfunktionen genannt) schreiben. Die komplette Semantik lässt sich in ähnlicher Weise bei Methoden wiederverwenden. Methoden sind, vereinfacht ausgedrückt, auch wieder nur Funktionen – aber genauer sind es eben Funktionen von Klassen, die das Verhalten eines Objekts charakterisieren. Mehr dazu erfahren Sie, wenn es um die objektorientierte Programmierung mit Klassen geht (siehe Kapitel 7, »Grundlagen zu den Klassen«). Speziell die Methoden werden in Abschnitt 7.7, »Methoden – die Funktionen der Klasse«, beschrieben.

Kapitel 6

Modularisierung

In diesem Kapitel werden wichtige Schlüsselwörter für Typen, Namensbereiche, die Präprozessor-Direktiven und das Organisieren von Quelldateien behandelt.

Auf den folgenden Seiten werden verschiedene Themen abgehandelt, die sich recht gut in Verbindung mit der Organisation (Modularisierung) von mehreren Quelldateien beschreiben lassen. Hierbei werden Sie die Namensräume kennenlernen. Bisher haben Sie in Ihren Beispielen `std::` geschrieben, ohne dass dies etwas genauer erklärt wurde. Ebenso haben Sie verschiedene Schlüsselwörter für Speicherklassenattribute oder Typqualifikatoren verwendet, ohne Genaueres darüber zu erfahren. Daher werden diese Schlüsselwörter ebenfalls in diesem Kapitel behandelt. Zusätzlich werden hier auch noch die Präprozessoranweisungen erläutert (bspw. `#include` usw.). Am Ende möchte ich Ihnen noch ein paar Ratschläge mitgeben, wie Sie Ihren Code organisieren können.

6.1 Namensräume

In diesem Abschnitt werden Sie eigene Namensräume erstellen und verwenden. Am Ende soll noch kurz erläutert werden, warum in den Programmen bisher immer `std::` verwendet wurde und welche Alternativen es dafür gibt.

Mit einem Namensraum (engl. *Namespace*) erzeugen Sie einen eigenen Gültigkeitsbereich, worin Sie beliebige Bezeichner wie Klassen, Variablen, Funktionen, Typen oder gegebenenfalls sogar weitere Namensräume deklarieren können. Besonders bei umfangreichen Projekten mit mehreren Personen und mehreren Modulen (Quelldateien) und Klassenbibliotheken wird zur Verwendung von Namensräumen geraten, um eventuelle Konflikte mit gleichnamigen Bezeichnern zu vermeiden.

Um den Sinn und Zweck von Namensbereichen zu verstehen, finden Sie hier ein völlig offensichtliches Beispiel dazu. Betrachten Sie folgenden einfachen Code:

```
00 // kapitel06/namespace01.cpp
01 #include <iostream>
02 using namespace std;

03 void cout( int data );
```



```

04 int main() {
05     cout << "Das gibt Probleme\n";
06 }

07 void cout( int data ) {
08     // Code für sort();
09 }

```

Dieses kleine Beispiel enthält zwei unschöne Dinge, weshalb sich der Code nicht übersetzen lässt. In Zeile **03** deklarieren und in den Zeilen **07** bis **09** definieren wir eine Funktion `cout()`, was zu Problemen mit dem Standard-Stream `cout` führt. Dass die Funktion `cout()` zu einer Fehlermeldung bei der Übersetzung führt, ist allerdings das globale Einbinden vom Namensraum `std` in Zeile **02** Schuld, wodurch es jetzt zwei Bezeichner `cout` im Gültigkeitsbereich gibt. Das Problem könnten Sie jetzt hier beheben, indem Sie Zeile **02** entfernen und in Zeile **05** wieder wie gehabt über `std::cout` auf den Stream zugreifen würden.

Oder aber Sie könnten das Problem beheben, indem Sie die Funktion `cout()` in einen eigenen Namensraum verpacken:

```

00 // kapitel06/namespace02.cpp
01 #include <iostream>
02 using namespace std; // Nicht gut! Siehe Abschnitt 6.1.4

03 namespace mySpace { // Neuer Namensraum - Anfang
04     void cout( int data ) {
05         // Anweisungen
06     }
07 } // Neuer Namensraum - Ende

08 int main() {
09     cout << "Jetzt gibt es keine Probleme\n";
10     mySpace::cout(1234);
11 }

```

Jetzt wurde die Funktion `cout()` zwischen den Zeilen **03** bis **07** in einen Namensraum `mySpace` gepackt. So viel zur Theorie. Mehr dazu erfahren Sie in den folgenden Abschnitten. Ich wollte Sie hier nur schon mal für das Thema Namensräume sensibilisieren. Auch wenn das Problem hier mit `cout` ziemlich offensichtlich war, sollte das Beispiel zeigen, zu welchen Problemen es bei zwei gleichen Bezeichnern im selben Gültigkeitsbereich kommen kann. In solch trivialen Beispielen hat man das Problem vielleicht im Überblick, aber bei mehreren tausend Zeilen Code und ein paar Dutzend Modulen und vielen verschiedenen Entwicklern kann es da schon mal zu Überschneidungen kommen.

Der Namensraum std

Dem Namensraum `std` ist ein extra Abschnitt gewidmet (Abschnitt 6.1.4, »Der Namensraum `std`«).

6.1.1 Einen neuen Namensraum erstellen

Um zu veranschaulichen, wie Sie einen neuen Namensraum erstellen, soll ein möglichst einfaches Fallbeispiel erhalten, in dem es zu Konflikten kommt. Betrachten Sie die beiden Codeausschnitte von *datei1.cpp* und *eineHeaderdatei.h*. Hier kommt es ganz offensichtlich zu Konflikten:

```
// kapitel06/0/datei1.cpp
#include <iostream>
#include <string>
#include "eineHeaderdatei.h"

std::string nname;
std::string vname;
unsigned short alter;
struct Datensatz { /*...*/};
void function() { /*...*/}

int main() {
    // ...
}

// kapitel06/0/eineHeaderdatei.h
#include <string>

std::string nname;
std::string vname;
unsigned short alter;
struct Datensatz { /*...*/};
void function() { /*...*/}
class aClass { /*...*/};

// ...
```

Was sofort auffallen sollte, ist, dass es hier gleichnamige Bezeichner im globalen Gültigkeitsbereich gibt, weil die Headerdatei *eineHeaderdatei.h* auch noch in *datei1.cpp* mit eingebunden wurde. Eine einfache Lösung bei solchen Minibeispielen könnte es sein, einfach andere Bezeichner zu verwenden. Aber bei ein paar tausend Zeilen bedeutet dies dann schon einen ziemlichen Aufwand. Allerdings liefert C++ für solche Zwecke ja eigene Namensräume.

Einen neuen Namensraum können Sie mit dem Schlüsselwort `namespace` wie folgt anlegen:

```
namespace NAME_FUER_NAMENSBEREICH
{
    // Deklarationen und Defintionen
}
```

Bezogen auf unsere Beispiele stehen Ihnen jetzt zwei Möglichkeiten zur Verfügung.

Möglichkeit 1

Der Header wird in den neuen Namensbereich gesetzt:

<pre>// kapitel06/1/datei1.cpp #include <iostream> #include <string> namespace myspace { #include "eineHeaderdatei.h" } std::string nname; std::string vname; unsigned short alter; struct Datensatz { /*...*/}; void function() { /*...*/} int main() { // ... }</pre>	<pre>// kapitel06/1/eineHeaderdatei.h #include <string> std::string nname; std::string vname; unsigned short alter; struct Datensatz { /*...*/}; void function() { /*...*/} class aClass { /*...*/}; // ...</pre>

Möglichkeit 2

Wir erstellen einen neuen Namensraum in der Quelldatei.

<pre>// kapitel06/2/datei1.cpp #include <iostream> #include <string> #include "eineHeaderdatei.h" std::string nname; std::string vname; unsigned short alter; struct Datensatz { /*...*/}; void function() { /*...*/} int main() { // ... }</pre>	<pre>// kapitel06/2/eineHeaderdatei.h #include <string> namespace myspace { std::string nname; std::string vname; unsigned short alter; struct Datensatz { /*...*/}; void function() { /*...*/} class aClass { /*...*/}; } // ...</pre>

Es spricht auch nichts dagegen, viele weitere Namensbereiche zu verwenden. Die Anzahl der Namensbereiche ist nicht beschränkt. Im Beispiel wäre es praktisch recht sinnvoll, die globalen Dinge von *datei1.cpp* auch noch in einen Namensraum zu packen:

```
// kapitel06/3/datei1.cpp
#include <iostream>
#include <string>
#include "eineHeaderdatei.h"

namespace otherspace {
    std::string nname;
    std::string vname;
    unsigned short alter;
    struct Datensatz { /*...*/};
    void funktion() {/*...*/}
}

int main() {
    // ...
}
```

```
// kapitel06/3/eineHeaderdatei.h
#include <string>

namespace myspace {
    std::string nname;
    std::string vname;
    unsigned short alter;
    struct Datensatz { /*...*/};
    void funktion() {/*...*/}
    class aClass { /*...*/};
}

// ...
```

Die Anzahl der Namensbereiche ist nicht beschränkt.

Deklarationen und Definitionen trennen

In der Praxis wird auch hier gewöhnlich die Deklaration von Funktionen, Strukturen, Klassen usw. von der Definition getrennt. Die Definition erstellt man gewöhnlich in einer anderen Quelldatei. Auch hierbei bieten sich zwei verschiedene Möglichkeiten an, die Deklaration und Definition eines Namensbereichs zu trennen.

Möglichkeit 1: Deklarationen und Definitionen voneinander trennen

Deklarationen	Definitionen
#include <string>	...
...	// Definition von Datensatz
namespace myspace {	struct myspace:: Datensatz {
std::string nname;	// Typen ...
std::string vname;	};
unsigned short alter;	// Definition von funktion()
struct Datensatz;	void myspace:: funktion() {
void funktion();	// Code für funktion
class aClass;	}
}	// Definition von aClass
	class myspace:: aClass {
// ...	// Code für Klasse
	};
	// ...

Möglichkeit 2: Deklarationen und Definitionen voneinander trennen, indem einfach erneut der Namensraum geöffnet wird

Deklarationen

```
#include <string>
...
namespace myspace {
    std::string nname;
    std::string vname;
    unsigned short alter;
    struct Datensatz;
    void funktion();
    class aClass;
}

// ...
```

Definitionen

```
...
// Namensbereich erneut öffnen
namespace myspace {
    // Definition von Datensatz
    struct Datensatz {
        // Typen für Datensatz
    };
    // Definition von funktion()
    void funktion() {
        // Code für funktion()
    }
    // Definition von aClass
    class aClass {
        // Code für aClass
    };
}

// ...
```

6.1.2 Namensraum verwenden

Namensräume sind eigene Gültigkeitsbereiche, wie Sie diese in Abschnitt 5.4, »Exkurs: Gültigkeitsbereiche«, kennengelernt haben. Es gibt jetzt drei Möglichkeiten, wie Sie nun auf einen Namensbereich zugreifen können. Als Beispiel soll folgender, sehr einfach gehaltener Code verwendet werden.

```
// kapitel06/datei1-0.cpp
#include <iostream>
#include <string>
#include "eineHeaderdatei.h"

std::string nname;
std::string vname;
unsigned short alter;
struct Datensatz { /*...*/};
void funktion() { /*...*/}

int main() {
    // ...
}
```

```
// kapitel06/eineHeaderdatei.h
#include <string>

namespace myspace {
    std::string nname;
    std::string vname;
    unsigned short alter;
    struct Datensatz { /*...*/};
    void funktion() { /*...*/}
    class aClass { /*...*/};
}

// ...
```

Mithilfe des Quellcodes sollen drei Möglichkeiten demonstriert werden, um auf einen Namensraum zuzugreifen.

Möglichkeit 1: Mit Bereichsoperator (Scope)

Der Zugriff mithilfe des Scope-Operators `::` (oder auch Bereichsoperators) dürfte Ihnen ja relativ geläufig sein, weil Sie diesen Zugriff bereits bei sämtlichen Beispielen mit `std::` verwendet haben. Um einzelne Bezeichner aus einem Namensraum verwenden zu können, müssen Sie den Namensbereich, gefolgt vom Scope-Operator, und dann den Bezeichner angeben:

Namensbereich::**Bezeichner**

Bezogen auf unser Beispiel könnte dies jetzt wie folgt aussehen:

```
00 // kapitel06/datei-1-1.cpp
...
01 #include "eineHeaderdatei.h"
...
02 int main() {
03     myspace::nname = "Mustermann";
04     myspace::vname = "Max";
05     nname = "Musterfrau";
06     vname = "Maxima";
07     std::cout << myspace::nname << std::endl;
08     std::cout << nname << std::endl;
09     myspace::aClass testClass;
...
10 }
```

Um auf Namensbereiche zuzugreifen, müssen Sie dem Projekt natürlich entsprechende Dateien hinzufügen. Im Beispiel wurde alles recht einfach in eine Headerdatei verpackt, welche wir hier auch in Zeile **01** hinzugefügt haben. In den Zeilen **03** und **04** erfolgt ein Zugriff auf die Variablen `nname` und `vname` aus dem Namensbereich `myspace`, welche in der Headerdatei *eineHeaderdatei.h* deklariert und definiert wurden. Der Zugriff in den Zeilen **05** und **06** auf die gleichnamigen Bezeichner `nname` und `vname` erfolgt dann auf die globalen Variablen aus der Quelldatei *datei1.cpp*. In Zeile **09** wird ein Objekt vom Typ `aClass` aus dem Namensraum `myspace` definiert.

Möglichkeit 2: Einzelne Bezeichner importieren

Die zweite Möglichkeit ist es, nur einzelne Bezeichner aus einem Namensraum mit dem Schlüsselwort `using` zu importieren, das wird auch als `using`-Deklaration bezeichnet:

```
using Namensraum::Bezeichner;
```

In der Praxis in unserem Quellcode lässt sich dies bspw. wie folgt realisieren:

```
00 // kapitel06/datei-1-2.cpp
...
01 #include "eineHeaderdatei.h"
...
02 int main() {
03     using myspace::nname;
04     using myspace::vname;
05     using myspace::aClass;
06     nname = "Mustermann";
07     vname = "Max";
08     ::nname = "Musterfrau";
09     ::vname = "Maxima";
10     aClass testClass;
11     std::cout << nname << std::endl;
12     std::cout << ::nname << std::endl;
13 }
```

Mithilfe der drei `using`-Deklarationen in den Zeilen 03 bis 05 innerhalb der `main`-Funktion stehen die Bezeichner `nname`, `vname` und `aClass` innerhalb der `main`-Funktion lokal zu Verfügung und können ohne Scope-Operator verwendet werden. Da es allerdings globale Funktionen mit denselben Bezeichnern gibt, würde es zu einem Konflikt kommen. Hier können wir uns in den Zeilen 08 und 09 nochmals mit dem alleinigen Scope-Operator (`::Bezeichner`) behelfen, um explizit auf die globalen Bezeichner zuzugreifen.

Globale `using`-Deklaration

Natürlich ist es auch möglich, die `using`-Deklaration global außerhalb einer Funktion zu machen. Allerdings würde es in diesem Fall dann tatsächlich zu Problemen mit den gleichnamigen globalen Bezeichnern kommen. Hier könnten Sie sich dann auch nicht mehr mit einem alleinigen Scope-Operator (`::Bezeichner`) retten. Ich denke, spätestens jetzt dürfte es Ihnen dämmern, dass es mit globalen Variablen irgendwann immer zu Problemen kommt und warum es besser ist, wenn möglich auf globale Sachen zu verzichten. Daher wäre es hier auch ideal, die globalen Bezeichner wiederum in einen Namensbereich zu packen.

Möglichkeit 3: Alles importieren

Die dritte Möglichkeit wäre es, den kompletten Namensraum zur Verfügung zu stellen. Damit wäre es möglich, ohne Umwege direkt den Bezeichner zu verwenden – also explizite Anwendung über den Scope-Operator:

```
using namespace Namensraum;
```

Bezogen auf unser Beispiel wird es schwer, den Namensraum komplett mit `using namespace myspace;` zur Verfügung zu stellen, weil das zwangsläufig zu Konflikten mit den globalen Bezeichnern führen würde. Hier kommen Sie daher nicht mehr um den Scope-Operator herum, um Mehrdeutigkeiten zu vermeiden:

```
00 // kapitel06/datei-1-3.cpp
...
01 #include "eineHeaderdatei.h"
...
02 int main() {
03     using namespace myspace;
04     myspace::nname = "Mustermann";
05     myspace::vname = "Max";
06     ::nname = "Musterfrau";
07     ::vname = "Maxima";
08     aClass testClass;

09     std::cout << myspace::nname << std::endl;
10     std::cout << ::nname << std::endl;
11 }
```

In Zeile 03 wird der komplette Namensraum von `myspace` importiert und innerhalb der `main`-Funktion zur Verfügung gestellt. Damit es hierbei nicht zu Namenskonflikten mit den gleichnamigen globalen Bezeichnern kommt, mussten in den Zeilen 04 und 05 trotzdem die Zugriffsoperatoren verwendet werden. Nur die Klasse in Zeile 08 konnte problemlos ohne einen Scope-Zugriff verwendet werden, weil es da keinen gleichnamigen globalen Bezeichner gibt.

Namensbereich ist ein echter Gültigkeitsbereich

An dieser Stelle muss nochmals deutlich gesagt werden, dass ein Namensbereich ein echter Gültigkeitsbereich ist. Die Gültigkeit einer `using`-Deklaration oder eines kompletten `using`-Namensraumes beginnt und endet immer innerhalb eines Anweisungsblocks ab `{` bis `}`. Deshalb ist es von enormer Bedeutung, wo Sie die `using`-Anweisungen platzieren, um einen eindeutigen Namen innerhalb eines Gültigkeitsbereichs zu erhalten. Wenn Sie `using`-Anweisungen außerhalb einer Funktion global platzieren, werden alle Bezeichner eines Namensraumes im kompletten Programm sichtbar. Hierbei sollten Sie sich schon Ihre Gedanken, machen! Warum zunächst etwas in einen Namensraum stecken, um es dann wieder öffentlich zur Verfügung zu stellen?

Zugriff auf verschachtelte Namensräume

Vollständigkeitshalber soll hier auch noch erwähnt werden, dass es durchaus möglich ist, Namensbereiche zu verschachteln, wie bspw.:

```
namespace outerSpace {
    // ...
    namespace innerSpace {
        void funktion();
    }
}
```

Der Zugriff auf `funktion()`, welche sich innerhalb vom Namensbereich `innerSpace` befindet, der sich wiederum innerhalb von `outerSpace` befindet, muss dann vollständig qualifiziert über `outerSpace::innerSpace::funktion()` erfolgen.

6.1.3 Aliases für Namensräume

Zu lange und umständliche Namen eines Namensraumes können Sie auch mit einem anderen Namen (um-)benennen oder verbergen. Ein einfaches Beispiel dazu:

```
01 namespace my_own_namespace {
02     unsigned int ival;
03     // ...
04 }

05 namespace mon = my_own_namespace;
...
06 using namespace mon;
...
07 mon::ival = 1234;
08 std::cout << my_own_namespace::ival << std::endl;
...
```

In den Zeilen **01** bis **04** wird ein Namensbereich mit dem Namen `my_own_namespace` erstellt. In Zeile **05** erstellen wir mit `mon` einen Aliasnamen für `my_own_namespace`. Sie können jetzt quasi `mon` und `my_own_namespace` verwenden. In Zeile **06** würden Sie alle Bezeichner aus `mon`, alias `my_own_namespace`, importieren. Der Zugriff auf die einzelnen Bezeichner des Namensbereichs kann jederzeit über `mon` (siehe Zeile **07**), aber auch über das ursprüngliche `my_own_namespace` (siehe Zeile **08**) erfolgen.

6.1.4 Der Namensraum `std`

An dieser Stelle muss noch auf den Namensraum `std` eingegangen werden, welchen wir in diesem Buch in fast jedem Beispiel mit `std::Bezeichner` verwendet haben.

Sehr häufig anzutreffen ist immer noch die Form, wo alles global importiert und zur Verfügung gestellt wird. In der Praxis sollten Sie sich aber bereits vorher darüber Gedanken machen, ob Sie den kompletten Inhalt von `std` verfügbar machen wollen, weil in der echten Praxis die Beispiele eher nicht so trivial wie in Büchern oder Tutorials sind. Gerne werden in eigenen Modulen die Bezeichner wie `count`, `sort`, `find`, `swap` oder ähnlich verwendet, und schnell hat man einen Konflikt, weil der Namensraum `std` ebenfalls solche Bezeichner enthält. Zugegeben, wenn man nur die Standardbibliothek verwendet, hat man alles noch im Griff. Aber spätestens wenn Sie eine von den unzähligen Bibliotheken verwenden, welche von anderen Programmierern oder bei einem Projekt zur Verfügung gestellt werden, kann (meistens wird) es immer wieder zu Namenskonflikten kommen.

Wie Sie bereits erfahren haben, stehen in der Praxis drei verschiedene Möglichkeiten zur Verfügung, die Bezeichner aus anderen Namensbereichen zu verwenden. Hier nochmals die drei Möglichkeiten in Bezug auf `std`:

1. Sie verwenden den Namensraum mit dem Bezeichner direkt im Quellcode. Statt bspw. `cout` für die Ausgabe verwenden Sie `std::cout`. Der Vorteil ist hierbei, dass es niemals mehr zu einem Konflikt gleicher Bezeichner kommen kann. In der Praxis sieht dies dann wie folgt aus:

```
std::cout << "Namensbereich:." << std::endl;
```

2. Sie verwenden eine `using`-Deklaration, womit Sie nur einzelne Bezeichner eines Namensraumes öffentlich machen. Schreiben Sie diese `using`-Deklaration dann auch noch schön brav in einen Anweisungsblock, dann sind diese Bezeichner nur in diesem Anweisungsblock gültig. In der Praxis sieht dies wie folgt aus:

```
{
    using std::cout;
    using std::endl;
    cout << "Namensbereich:." << endl;
    ...
}
cout << "Namensbereich:." << endl; // Fehler!!!
```

Durch die die beiden `using`-Direktiven in diesem Beispiel sind `cout` und `endl` innerhalb des Anweisungsblocks sehr komfortabel ohne den Namensbereich und Zugriffsoperator erreichbar.

3. Sie verwenden die bereits bekannte `using`-Direktive mit `namespace std` für den gesamten Namensraum `std`. Setzen Sie diese Direktive an einem globalen Bereich, sind alle Bezeichner überall verfügbar, und die Gefahr von Namenskonflikten ist am größten. Sie können dies aber auch ein wenig eingrenzen, indem Sie die `using`-Direktive innerhalb eines Anweisungsblocks schreiben. Dann sind die Bezeichner nur innerhalb des Anweisungsblocks sichtbar, wie folgendes Beispiel zeigen soll:

```

{
    using namespace std;
    cout << "Namensbereich::" << endl; // OK
}
cout << "Namensbereich::" << endl; // Fehler!!!

```

Innerhalb des Anweisungsblocks sind alle Bezeichner des Namensbereichs `std` bei diesem Beispiel verwendbar. Aber außerhalb des Anweisungsblocks sind die Bezeichner wieder unbekannt, weil der Namensraum `std` hier nicht mehr gültig ist.

Was ist jetzt die beste Lösung?

Das ist auch eine Frage des persönlichen Stils. Auf keinen Fall empfehlen kann ich eine `using`-Deklaration oder den kompletten Namensbereich mit `using global` zu stellen. Es macht einfach keinen Sinn, erst alles in einen Namensbereich zu stecken, um anschließend alles wieder global zur Verfügung zu stellen. Sicherer ist es da schon, einzelne Bezeichner oder gar den kompletten Namensraum innerhalb eines Anweisungsblocks zu importieren. Oder aber Sie verwenden gleich den Namensraum mit Bezeichner direkt im Code (`Namensraum::Bezeichner`). Dies bedeutet natürlich das geringste Namenskonfliktrisiko, aber die meiste »Schreibarbeit«.

Für manche wiederkehrenden Aufgaben hat sich die Form `using std::BEZEICHNER` innerhalb von Funktionen durchgesetzt, so z.B., wenn man `swap()` für eigene Objekte implementieren möchte – ein `using std::swap;` zu Beginn der eigenen Funktion macht den Bezeichner standardmäßig innerhalb der eigenen Funktion bekannt.

6.2 Speicherklassenattribute

Mit den Schlüsselwörtern `extern` und `static` legen Sie fest, wo die Typen im Speicher abgelegt werden und/oder wie lange diese dort gültig sind. Hier gibt es noch weitere Schlüsselwörter wie `mutable` und `thread_local`, deren Bedeutungen aber erst im entsprechenden Kapitel erläutert werden.

6.2.1 Das Schlüsselwort `extern`

Mit dem Schlüsselwort `extern` weiß der Compiler, dass der Linker das Speicherobjekt in einer anderen Übersetzungseinheit (Quell- oder Headerdatei) auflösen muss. Wenn Sie bspw. eine globale Variable `ival` in der Quelldatei `extern01.cpp` definieren und auch noch in der Datei `extern02.cpp` verwenden wollen, können Sie diese Variablen in `extern02.cpp` nicht erneut definieren, weil sich der Compiler sonst beschweren würde. Mithilfe des Schlüsselwortes `extern` vor der Variablen `ival` in `extern02.cpp`

weiß der Compiler jetzt, dass sich der Linker um die Auflösung kümmert. Die zwei Dateien, welche dies demonstrieren sollen, sehen Sie hier:

<pre>// kapitel06/extern01.cpp #include <iostream> int var=0; // Definition void print(); // Deklaration int main() { var = 123456; print(); }</pre>	<pre>// kapitel06/extern02.cpp #include <iostream> extern int var; // Deklaration void print() { std::cout << var << std::endl; }</pre>

Zugegeben, das ist kein schönes Beispiel, aber es demonstriert das Schlüsselwort `extern` sehr gut. Würden Sie `extern` in der Quelldatei *extern02.cpp* entfernen, würde der Linker beim Auflösen der Symbole die Übersetzung verweigern, weil zwei gleichnamige Bezeichner definiert werden müssten. Probieren Sie es selbst aus.

Beide Dateien beim Übersetzen angeben

Da Sie hier mit *extern01.cpp* und *extern02.cpp* zwei Quelldateien haben, müssen Sie beide Quelldateien bei der Übersetzung mit angeben, um daraus eine ausführbare Datei zu machen. Auf das Übersetzen mehrerer Quelldateien wird noch gesondert in Abschnitt 6.6, »Modulare Programmierung – Code organisieren«, eingegangen.

Im Beispiel konnten Sie sehr schön erkennen, dass Sie mit dem Schlüsselwort `extern` die Möglichkeit haben, eine Variable nur zu deklarieren. Dies entspricht in etwa der Deklaration des Funktionsprototyps in der Datei *extern01.cpp* mit `void print();`. Theoretisch **könnten** Sie hier vor der Deklaration des Funktionsprototyps auch das Schlüsselwort `extern` stellen, um die Lesbarkeit des Programms zu erhöhen – nämlich um dem Leser mitzuteilen, dass die Definition woanders erfolgt.

»extern« bei der Funktionsdeklaration nicht nötig

Bei Funktionsdeklarationen muss das Schlüsselwort `extern` nicht zwangsläufig verwendet werden (ist aber auch nicht falsch). Deklarationen von Funktionsprototypen werden vom Compiler implizit automatisch als `extern` behandelt, wenn nicht noch die Definition in der gleichen Datei folgt.

Natürlich könnten Sie es sich jetzt auch ganz einfach machen und statt *extern02.cpp* eine Headerdatei verwenden, in der Sie die Definition der Variablen und der Funk-

tion schreiben, und diese Headerdatei dann in *extern01.cpp* inkludieren. Genau genommen teilt man auch hier wiederum die Deklaration im Header von der Definition, aber das soll jetzt hier noch nicht behandelt werden, sondern erst – dann ausführlicher – in Abschnitt 6.6.

Tip: Auf globale Variablen wenn möglich verzichten

Wenn es möglich ist, sollten Sie auf globale Variablen (und Objekte) verzichten. Damit ersparen Sie sich seine Menge an Zeit, die Sie in die Suche nach schwer auffindbaren Fehlern investieren müssten, wenn der schlechteste Fall eintritt.

6.2.2 Das Schlüsselwort `static`

Das Schlüsselwort `static` kann auf zwei unterschiedlichen Wege verwendet werden. Je nachdem, ob das Schlüsselwort global oder lokal verwendet wird, ändert sich die Bedeutung oder auch hier der Gültigkeitsbereich.

Globales `static`

Wenn Sie `static` vor einer globalen Variablen oder Funktion setzen, beschränkt sich die Gültigkeit auf das Modul (die Datei). Damit könnten Sie Namenskonflikte zwischen gleichnamigen Bezeichnern in unterschiedlichen Dateien vermeiden. Eine mit `static` gekennzeichnete Variable wird außerdem auch automatisch mit 0 initialisiert. Betrachten Sie hier folgendes fast bekannte Listing:

<pre>// kapitel06/static01.cpp #include <iostream> int var=0; // Definition void print(); // Deklaration int main() { var = 123456; print(); }</pre>	<pre>// kapitel06/static02.cpp #include <iostream> static int var; // Definition void print() { std::cout << var << std::endl; }</pre>

Das Beispiel zeigt, wie es mit `static` für globale Bezeichner schnell zu Doppeldeutigkeiten kommen kann und wie es häufig eher verwirrt, als etwas deutlich zu machen.

Im Gegensatz zu den Beispielen von *extern01.cpp* und *extern02.cpp* aus dem vorherigen Abschnitt wurde in diesem Beispiel das Schlüsselwort `static` nur bei *static02.cpp* vor die Variable `var` gesetzt (statt des Schlüsselwortes `extern`). Damit haben Sie jetzt

in *static02.cpp* eine Variable `var` definiert (welche auch automatisch mit 0 initialisiert wurde), die jetzt nur innerhalb der Datei *static02.cpp* gültig ist. Die andere globale Variable `var` aus der Datei *static01.cpp* weiß nichts von dieser mit `static` gekennzeichneten Variablen, und es kommt somit auch zu keinem Konflikt mehr.

Leider hat das Beispiel eine Doppeldeutigkeit, und es ist nicht zu erkennen, was der Programmierer hier erreichen wollte. Die Wertzuweisung von `var=123456` in *static01.cpp* hat keinerlei Effekt mehr auf die folgende Ausgabe mit `print()`, weil die Funktion `print()` in *static02.cpp* definiert wurde und dort eben das (Datei-)lokale `var` – hier also das `static int var` – verwendet. Mit `var=123456` weisen Sie allerdings den Wert der globalen Variablen `var` von *static01.cpp* zu.

Tipp: Verzichten Sie auf globales static

Aufgrund der Doppeldeutigkeiten, welche beim globalen `static` auftreten können, empfehle ich Ihnen, ganz darauf zu verzichten. Wenn Sie den Gültigkeitsbereich innerhalb einer Datei beschränken müssen, können Sie stattdessen auch auf einen anonymen Namensraum zurückgreifen.

Den anonymen Namensbereich habe ich Ihnen noch vorenthalten. Das Thema um die Namensbereiche wurde ja bereits in Abschnitt 6.1, »Namensräume«, beschrieben. Hierzu sehen Sie eine gleichwertige Alternative zum globalen `static`, um mithilfe von anonymen Namensräumen eine Variable ebenfalls nur innerhalb einer Datei gültig zu machen.

<pre>// kapitel06/static03.cpp #include <iostream> static int file_local; int main() { file_local = 123456; }</pre>	<pre>// kapitel06/static04.cpp #include <iostream> namespace { int file_local; } int main() { file_local = 123456; }</pre>

Im linken Beispiel (*static03.cpp*) wird das Schlüsselwort `static` verwendet, um die Gültigkeit der Variablen `file_local` auf die Datei zu beschränken. Dasselbe erreichen Sie jetzt auch (siehe *static04.cpp*), wenn Sie eine Variable bzw. Funktion innerhalb eines leeren `namespace {}` schreiben. Alles, was zwischen `{` und `}` im anonymen Namensraum steht, ist ebenfalls nur innerhalb der Datei gültig.

Global, static global oder anonym

Zugegeben in diesem Abschnitt wurde das Thema recht umfassend und vielseitig behandelt. Daher soll noch erwähnt werden, dass im Fall der Fälle eine dateilokale `static`-Variable einer globalen Variablen vorzuziehen ist, weil sich die Fehlersuche hiermit zumindest von allen Modulen auf eine Datei reduziert. Trotzdem soll dies hier nicht heißen, dass dateiglobale Sachen so viel besser sind. Und da `static` (als lokales `static`) noch eine weitere Funktion hat, wäre es vielleicht besser, für dateilokale Sachen einen anonymen Namensraum zu verwenden.

Lokales static

Wie bereits eben erwähnt, hat das Schlüsselwort `static` noch eine weitere, recht andere Funktion, welche sich zwar auch wieder auf das Thema Gültigkeitsbereich bezieht, aber eben häufig für Verwirrung sorgt, weil es eben eine andere Bedeutung bekommt. Wenn Sie eine `static`-Variable innerhalb eines Anweisungsblocks (bspw. Funktion) verwenden, bekommt diese Variable einen festen Platz im Datensegment und bleibt somit zur Laufzeit des Programms dort enthalten. Innerhalb einer Funktion bleibt eine mit `static` gekennzeichnete Variable für immer gültig und behält sogar ihren Wert nach dem Ende der Funktion – so dass bei erneutem Funktionsaufruf der beim letzten Funktionsaufruf gesetzte Wert vorhanden ist. In der Praxis ist damit quasi auch möglich, ein mit `static` gekennzeichnetes (lokales) Speicherobjekt aus einer Funktion zurückzugeben, das ja mit einer gewöhnlichen lokalen Variablen auf dem Stapelspeicher (Stack/-Segment) nicht erlaubt ist, weil dieses nach dem Ende der Funktion zerstört würde.

Lokaler Gültigkeitsbereich bleibt allerdings

Hier sollte noch erwähnt werden, dass eine mit `static` gekennzeichnete lokale Variable trotzdem nach wie vor nur innerhalb des Gültigkeitsbereichs (Anweisungsblock) gültig ist, wo diese definiert wurde.

Hierzu ein einfaches Beispiel, welches das lokale `static` als Gedächtnis einer Funktion verwendet:

```
00 // kapitel06/static05.cpp
01 #include <iostream>

02 int pseudozufallszahl(int x);

03 int main() {
04     std::cout << pseudozufallszahl(11111) << std::endl;
```

```

05  std::cout << pseudozufallszahl(11111) << std::endl;
06  std::cout << pseudozufallszahl(11111) << std::endl;
07  }

08  int pseudozufallszahl(int x) {
09      static int startwert = 246532;
10      if( x % 2)
11          startwert+=x;
12      else
13          startwert-=x;
14      return startwert;
15  }

```

Wenn Sie das Beispiel *static05.cpp* ausführen, werden Sie vielleicht überrascht sein, dass in den Zeilen 04 bis 06 dreimal unterschiedliche Werte ausgegeben werden, obwohl die Funktion `pseudozufallszahl()` immer mit demselben Wert aufgerufen wird. Diesen Umstand verdanken Sie der statischen Variablen `startwert` in Zeile 09, welche sich ja bei jedem erneuten Funktionsaufruf den Wert des vorherigen Funktionsaufrufes gemerkt hat und somit immer mit dem zuvor verwendeten Wert weiterarbeitet. Dies ist nur dadurch möglich, weil sich diese Variable im Datensegment und nicht wie lokale Variablen ohne `static` auf dem Stacksegment befindet. Die Zeilen 10 bis 13 sind nur ein Beiwerk, um den übergebenen Wert `x`, wenn dieser ungerade ist, zum `startwert` hinzuzuaddieren oder bei ungeradem Wert zu subtrahieren.

Genauer Zeitpunkt der Initialisierung

Lokale `static`-Variablen werden genau dann initialisiert, wenn der Code, in dem diese stehen, das erste Mal ausgeführt wird. Globale Variablen hingegen sind (oder werden) alle schon beim Programmstart initialisiert.

6.2.3 Aus Alt mach Neu und »deprecated«

In Abschnitt 2.5, »Typ `auto` (C++11)«, haben Sie ja bereits die neue Bedeutung des alten Schlüsselwortes `auto` im neuen C++11-Standard kennengelernt. Die alte Bedeutung von `auto` als Speicherklasse war ohnehin ziemlich überflüssig, weil Variablen innerhalb von Funktionen schon `auto` waren und außerhalb nicht erlaubt sind. Daher wurde die alte Verwendung von `auto` entfernt (C++03) und einer neuen Bedeutung zugeordnet (C++11). Daher ist das alte `auto` eines der wenigen Dinge in C++, welche im neuen C++11-Standard nicht abwärtskompatibel sind.

Deprecated: Schlüsselwort register

Früher konnte man noch mit dem Schlüsselwort `register` dem Compiler vorschlagen, diese Variable in das schnellere Prozessorregister statt den langsameren Arbeitsspeicher zu legen. Aber da der Compiler letztendlich selbst entscheidet, was er in das Prozessorregister legt, war auch dieses Schlüsselwort recht überflüssig und wurde mit C++11 zu »deprecated« erklärt, was bedeutet, dass Sie dieses Schlüsselwort für künftige Projekte nicht mehr verwenden sollten.

6.3 Typqualifikatoren

Es gibt mit `const` und `volatile` zwei Typqualifikatoren, mit denen Sie bei einer Deklaration die Typen weiter modifizieren können. Wobei `volatile` in der Praxis wohl eher seltener benötigt wird.

6.3.1 Das Schlüsselwort `const`

Speicherobjekte, welche Sie mit `const` gekennzeichnet haben, können nach ihrer Definition im Speicher nicht mehr verändert werden. Das Objekt wird damit mit einem Schreibschutz versehen. Jeglicher Versuch, ein mit `const` geschütztes Objekt zu verändern, führt zu einer Fehlermeldung des Compilers.

Doch keine echte Konstante?

Ein Objekt, welches mit `const` gekennzeichnet wird, legt, um es recht genau zu nehmen, nicht wirklich eine echte »physische« Konstante an. Mit `const` wird ein Objekt lediglich als nicht mehr änderbar (*read-only*) markiert und darf somit nicht mehr auf der linken Seite einer Zuweisung stehen. Versuchte Änderungen des Speicherobjekts, welches mit `const` markiert wurde, würde der Compiler sofort bemäkeln. Es ist mit `const_cast<>` trotzdem möglich, u. U. diesen Schreibschutz des Compilers zu »überkleben«.

Besonders häufig wird `const` bei Parametern von Funktionen (und später auch Methoden) verwendet, um ein versehentliches Verändern eines Objekts innerhalb einer Funktion zu verhindern (siehe auch Abschnitt 5.5, »Funktionsparameter«). Mit einem Funktionskopf wie `void funktion(const int& val);` stellen Sie praktisch sicher, dass `val` in der Funktion nicht mehr versehentlich geändert werden kann.

Hierzu ein kurzes Beispiel mit `const`:

```

00 // kapitel06/const01.cpp
01 #include <iostream>
02 #include <string>

03 int quad(const int x);

04 int main() {
05     const std::string r_str = "Text mit Schreibschutz";
06     r_str = "Fehler!!!"; // !!! Fehler !!!
07     std::cout << quad(10) << std::endl;
08 }

09 int quad(const int x) {
10     x*=x; // !!! Fehler !!!
11     return x;
12 }

```

In Zeile 05 kennzeichnen Sie `r_str` mit `const` und versehen den String damit mit einem Schreibschutz. Der Versuch, dieses Objekt in Zeile 06 trotzdem auf die linke Seite einer Zuweisung zu stellen, wird der Compiler mit einem deutlichen »Kopfschütteln« quittieren. Ebenso sieht es in der Funktion `quad()` (Zeile 09 bis 12) aus, wo in Zeile 10 versucht wird, den Wert `x` zu ändern, der im Funktionskopf als `const` qualifiziert wurde. Das `*` stellt ja auch eine erweiterte Form der Zuweisung dar. Somit wird sich der Compiler über diese Zeile (10) beschweren. Den Fehler können Sie ganz einfach beheben, indem Sie Zeile 10 entfernen und in Zeile 11 einfach `return x*x;` tippen.

C++11: Stärkeres const mit constexpr

Mit C++11 wurde mit `constexpr` ein noch stärkeres `const` eingeführt. Allerdings ist `constexpr` schon ein bisschen mehr als nur ein einfaches `const`, weshalb dies gesondert im Abschnitt »constexpr« in Abschnitt 6.4.3 behandelt wird.

extern bei const

Wenn Sie `const`-Objekte global definieren, sind diese, anders als bei gewöhnlichen globalen Objekten, nur in der Quelldatei sichtbar, in der diese definiert wurden. Wenn Sie daher ein globales mit `const` qualifiziertes Speicherobjekt überall sichtbar machen wollen, müssen Sie es zusätzlich noch mit dem Schlüsselwort `extern` kennzeichnen. Betrachten Sie hierzu das folgende Beispiel:

<pre>// kapitel06/extconst01.cpp #include <iostream> extern const double pi=3.1415; void print(); // Deklaration int main() { print(); }</pre>	<pre>// kapitel06/extconst02.cpp #include <iostream> extern const double pi; void print() { std::cout << pi << std::endl; }</pre>

Mit `const` qualifizierte Objekte sind nur in der Quelldatei gültig, in der diese definiert wurden. Sollen diese auch in anderen Dateien sichtbar sein, müssen Sie sie zusätzlich noch mit `extern` deklarieren.

In der Datei *extconst01.cpp* wird `pi` deklariert und definiert. Wenn diese globale `const`-Variable künftig auch in anderen Dateien zur Verfügung stehen soll, muss das Schlüsselwort `extern` davorgestellt werden. In der Datei *exconst02.cpp* finden Sie hingegen nur die Deklaration (ohne Definition), welche nötig ist, damit der Linker einen Verweis auf die globale `const`-Variable erhält und sie in der Definition der Funktion `pi` verwendet werden kann.

6.3.2 Das Schlüsselwort `volatile`

`volatile` wird für eine tiefere Ebene benötigt und wohl von den wenigsten je benutzt werden. Eine mit `volatile` qualifizierte Variable kann von außerhalb des laufenden Programms einen neuen Wert erhalten. Bei der Erstellung von Maschinencode verhindert diese Qualifizierung einer Variablen mit `volatile` eine für die Funktion beeinträchtigende Optimierung, so dass im Programm immer auf den in der Hardware tatsächlich vorhandenen Wert zugegriffen wird.

In normalen Programmen sind solche Optimierungsgründe durchaus berechtigt und sinnvoll. Da legt der Prozessor schon mal gerne eine Variable in seinen Zwischenspeicher (Cache). Bei der Hardwareprogrammierung (wie Treibern) kann ein derartiges Verhalten aber katastrophal sein, weil sich im Cache noch die alten Zustände der Hardware befinden könnten. Da Sie sich nicht darauf verlassen können, dass die zwischengespeicherte Variable noch aktuell ist, erzwingen Sie mit `volatile`, dass die Variable frisch aus dem Hauptspeicher eingelesen wird.

Ein Beispiel zu `volatile` zu schreiben, bringt an dieser Stelle jetzt nicht viel. Daher soll folgender theoretische Code den Sachverhalt kurz demonstrieren:

```
volatile int *status;
...
for(volatile int i = 0; i < port_n; i++ ) {
```

```
...
*status = aktuell[i];
}
```

Ohne `volatile` könnte der Compiler hier bspw. die Zählvariable `i` und auch `port` in den Zwischenspeicher (Cache) des Prozessors legen, um die Laufzeit des Programms zu optimieren. Wenn Sie allerdings die Werte zur Laufzeit der Schleife ändern, wäre das für einen Hardwaretreiber eine Katastrophe. Dank `volatile` stellen Sie sicher, dass alle Zugriffe auf `port` so verlaufen, wie Sie es wollen, und es wird keine Optimierungen geben, so dass die Werte immer frisch aus dem Hauptspeicher gelesen werden.

6.4 Spezielle Schlüsselwörter

C++ hat noch weitere Schlüsselwörter anzubieten. Einige dieser Schlüsselwörter bekommen erst im Zusammenhang mit Klassen (ab Kapitel 7, »Grundlagen zu den Klassen«) einen Sinn und werden daher auch zu gegebener Zeit etwas umfassender beschrieben. Andere Schlüsselwörter wurden neu mit dem C++11-Standard eingeführt.

6.4.1 Das Schlüsselwort `inline` für Funktionen

Mit dem Schlüsselwort `inline` können Sie die Eigenschaft einer Funktion ändern. Allerdings wurde das Schlüsselwort bereits in Abschnitt 5.8.1, »Inline-Funktionen«, umfassend beschrieben, weshalb Sie bei Bedarf bitte dort nachschlagen.

6.4.2 Schlüsselwörter für Klassen (teilweise C++11)

Auch für Klassen gibt es mehrere Schlüsselwörter. Da wären zum einen die Schlüsselwörter `virtual` und `explicit`, welche im Zusammenhang mit Klassen und Methoden stehen. Die Beschreibung dazu finden Sie daher an passender Stelle im Buch. Das Schlüsselwort `virtual` wird ab Abschnitt 9.7, »Polymorphie mithilfe von virtuellen Methoden«, verwendet und beschrieben. Die Bedeutung von `explicit` lernen Sie im Abschnitt »Implizite Konvertierung vermeiden – `explicit`« in Abschnitt 7.4.2 kennen.

Neue Schlüsselwörter mit C++11

Neu in C++11 hinzugekommen sind die Schlüsselwörter `override` und `final`, welche im Zusammenhang mit `virtual` verwendet werden. Beide Schlüsselwörter werden in den Abschnitten 9.9.5, »Überschreiben erzwingen mit `override` (C++11)«, bzw. 9.9.6, »Nicht mehr überschreiben mit `final` (C++11)«, umfassender beschrieben.

Ebenso mit C++11 sind die Schlüsselwörter `delete` und `default` eingeführt worden. Genau genommen sind diese Schlüsselwörter nicht echt neu, sondern nur zusätzlich seit C++11 belegt worden. Beide Schlüsselwörter werden in Abschnitt 7.9, »Erzeugen von Methoden steuern (C++11)«, erläutert.

6.4.3 Neue Schlüsselwörter mit C++11

In diesem Abschnitt werden Sie die neu in C++11 eingeführten Schlüsselwörter `auto`, `decltype` und `constexpr` etwas besser kennenlernen.

auto

In Abschnitt 2.5, »Typ `auto` (C++11)«, wurde dieses Schlüsselwort bereits beschrieben, nämlich wo es dem Programmierer als Typinferenz das Arbeiten vereinfachen kann. Später, wenn Sie außerdem komplexere Typen, wie bspw. umständliche lange Iteratoren (siehe Abschnitt 12.4, »Iteratoren«), verwenden, werden Sie das neue `auto` schätzen, wenn Sie statt immer wieder `std::map<std::string, int>::const_iterator` einfach nur `auto` hinschreiben können. In Abschnitt 5.9, »Die neue Funktionssyntax (C++11)«, haben Sie dann auch noch ein weiteres Anwendungsgebiet von `auto` kennengelernt, nämlich als neue Syntax für Funktionsdeklarationen mit nachgestelltem Rückgabetyt.

decltype

Mit `decltype` können Sie sich die Typinformationen aus einem Ausdruck extrahieren lassen. Auf diese grundlegende Funktion wurde bereits in Abschnitt 5.9.2, »`decltype`«, eingegangen.

constexpr

Mit `constexpr` (kurz für *constant expression*, also konstanter Ausdruck) wurde ein Schlüsselwort eingeführt, mit dem es möglich ist, konstante Werte berechnen zu lassen. Im Gegensatz zu `const` funktioniert dies auch, wenn der Wert aus einer Funktion oder gar einem Konstruktor kommt (siehe Abschnitt 7.4.2, »Konstruktoren«). Betrachten Sie kurz folgendes Beispiel:

```
...
01 unsigned int plus1(unsigned int val=0) {
02     return (val + 1);
03 }
...
04 double rawArray[plus1(10)]; // Fehler !!!
```

In Zeile **04** wird sich der Compiler beschweren, dass in C++03 ein rohes Array mit variabler Länge nicht erlaubt ist. Wenn Sie sich allerdings den Code der Funktion genauer ansehen und den Funktionsaufruf `plus1(10)`, können Sie sofort erkennen, dass die Länge hier ja schon längst feststeht (hier 11).

Hier könnten Sie jetzt einwenden, warum dann nicht gleich 11 zwischen `[]` schreiben? Das ist richtig, aber manchmal gibt es Situationen, in denen man nicht einfach eine Zahl oder ein Literal verwenden kann, obwohl der Wert bereits feststeht. Das Beispiel mit `plus1()` ist zwar trivial, und hier schreibt man wohl besser ein Literal rein, aber es gibt häufig auch komplexere Berechnungen, die man dann eben einfach den Compiler durchführen und ihn einen entsprechenden Wert einsetzen lässt – und hier kommen konstante Ausdrücke mit dem neuen Schlüsselwort `constexpr` ins Spiel.

Hierzu nun die Funktion `plus1()` mit vorangestelltem `constexpr`:

```
...
01 constexpr unsigned int plus1(unsigned int val=0) {
02     return (val + 1);
03 }
...
04 double rawData[plus1(10)];
```

Durch das Voranstellen von `constexpr` in Zeile **01** prüft und berechnet der Compiler jetzt die Konstante für Funktionen und setzt den entsprechenden Wert in Zeile **04** zwischen `[]` ein.

Der Vorteil von `constexpr`-Ausdrücken ist, dass Sie damit sicherstellen können, dass die Daten vom Compiler schon berechnet werden können und somit in den ROM-Speicher (*Read-only-Memory*) gelegt und verwendet werden können. Damit dies auch möglich ist, werden einige Bedingungen vorausgesetzt. Zum Beispiel darf die Funktion nicht `void` als Rückgabetyt verwenden, es dürfen keine Variablen oder neue Typen deklariert werden, und es muss eine Rückgabewertangabe vorhanden sein.

Neben einer Array-Größe können solche konstanten Ausdrücke auch bei Template-Parametern, `enum`-Elementen, `case`-Fallunterscheidungen oder `static_assert`-Ausdrücken nützlich sein.

constexpr bei einem Konstruktor

Auch wenn die Konstruktoren erst in Abschnitt 7.4.2 behandelt werden, soll hier schon mal der Hinweis folgen, falls Sie `constexpr` darauf anwenden wollen, dass der Konstruktor dann aus einer Elementinitialisierungsliste (siehe Abschnitt »Konstruktoren effektiver initialisieren mit Element-Initialisierer« in Abschnitt 7.4.2) und einem leeren Anweisungsblock bestehen muss.

Ein weiterer Vorteil ist auch, dass der Compiler einen Fehler meldet, wenn der Ausdruck nicht zur Kompilierzeit (fürs ROM) berechnet werden kann. Dies kann ein einfaches `const` nicht leisten (auch ein `static const` nicht).

constexpr, das stärkere const

Es wurde bereits im Buch erwähnt, dass `constexpr` in gewisser Hinsicht auch als stärkeres `const` für ganz gewöhnliche Variablen verwendet werden kann. Ein Vorteil von `constexpr` ist es schon, dass Sie `constexpr` im Gegensatz zu `const` gleich mit Daten initialisieren müssen. Der Wert für `constexpr` muss quasi feststehen (daher müssen Funktionen auch einen Wert zurückgeben). Der Zwang zur Initialisierung von `constexpr`-Variablen erspart einem die eine oder andere böse Überraschung, welche bei Versäumnissen mit `const` auftreten können.

Und da der Compiler die Werte zur Übersetzungszeit berechnen kann, kann er diesen Wert als Zahl dann überall dort einsetzen, wo dieser benötigt wird. Es wird also nicht zwangsläufig eine Variable mit einer Speicheradresse angelegt. Betrachten Sie hierzu folgendes Beispiel:

```
...
01 #include <cmath> // für atan()

02 #define PI_V1 atan(1)*4
03 static const double PI_V2 = atan(1)*4;
04 constexpr double PI_V3 = atan(1)*4;
...
05 std::cout << PI_V1 << std::endl;
06 std::cout << PI_V2 << std::endl;
07 std::cout << PI_V3 << std::endl;
...
```

Hier finden Sie in den Zeilen **02** bis **04** drei verschiedene Möglichkeiten, die Kreiszahl `PI` errechnen zu lassen. Die schlechteste (nicht sehr C++-like) ist per `define`-Makro in Zeile **02** (siehe Abschnitt 6.5.2, »`#define` und `#undef`«). Hierbei handelt es sich nur um eine textuelle Ersetzung, bei der der Präprozessor noch vor dem Compilerlauf alle im Code verwendeten `PI_V1` durch `atan(1)*4` ersetzt. Praktisch bedeutet dies, dass überall jetzt `atan(1)*4` im Speicher abgelegt werden muss und die Berechnung an Ort und Stelle ausgeführt werden muss, wo eben `PI_V1` geschrieben wurde.

Besser ist dann schon die zweite Möglichkeit in Zeile **03** mit `const`. Hier wird nur einmal ein fester Speicherplatz zugewiesen und auch die Berechnung nur einmal bei der Zuweisung in Zeile **03** ausgeführt.

Im Gegensatz zu `const` existiert mit `constexpr` in Zeile **04** hingegen nicht zwangsläufig eine Speicheradresse. Es wird praktisch der berechnete Wert überall direkt einge-

setzt (wie bei einem Literal). Es wird nur eine Speicheradresse von `constexpr` verwendet, wenn Sie den Compiler zwingen, einen Speicher im Programm zu verwenden:

```
01 constexpr double PI_V3 = atan(1)*4;
02 const double* my_pi = &PI_V3;
```

Erst durch die explizite Verwendung des Adressoperators in Zeile 02 wird jetzt für `PI_V3` ein Speicher angelegt.

Auch in der Standardbibliothek von C++11 sind viele Dinge jetzt auch als `constexpr` implementiert. Das ist bspw. extrem hilfreich, wenn Sie ein Array mit der Größe eines bestimmten Datentyps mithilfe von `numeric_limits` anlegen wollen, wenn Sie bspw. folgendermaßen (vor dem C++11-Standard) ein Array anlegen wollten:

```
int char_feld[numeric_limits<char>::max()] = {0};
```

Da für die Array-Größe eine statische Größe nötig gewesen ist, ließ sich die hier gezeigte Verwendung nicht benutzen und wurde vom Compiler mit Fehlermeldungen quittiert. Im neuen C++11-Standard ist diese Verwendung allerdings jetzt möglich, weil hier `max()` und viele weitere Funktionen auch `constexpr` sind.

6.5 Präprozessor-Direktiven

Bevor der Compiler und gegebenenfalls danach der Linker etwas zum Übersetzen bekommen, wird vorher noch der Präprozessor ausgeführt (Abbildung 6.1). Der Präprozessor (manchmal auch Präcompiler) ist oftmals ein eigenes Programm, welches die Eingabedaten (den Quelltext) zur weiteren Verarbeitung für ein anderes Programm (hier den Compiler) aufbereitet (meistens eine textuelle Ersetzung durchführt). Den Präprozessorlauf bekommen Sie in der Regel nicht mit, weil dieser gewöhnlich automatisch vor dem Compilerlauf ausgeführt wird.

Präprozessorlauf

Abhängig vom Compiler können Sie selbstverständlich auch einen reinen Präprozessorlauf durchführen. Bei `g++` oder `clang` können Sie diesen mit der Option `-E` durchführen. Aber bei Microsoft Visual C++ können Sie bspw. mit `/E` die Ausgabe des Präprozessors auf die Standardausgabe betrachten. Hierzu empfehle ich Ihnen, bei Bedarf die Dokumentation bzw. das Manual Ihres Compilers zu lesen.

Die Befehle für den Präprozessor werden Direktiven (oder Präprozessor-Direktiven) genannt, beginnen immer mit dem `#`-Zeichen (Hash-Zeichen) am Anfang einer Zeile und enden am Zeilenende. Im Gegensatz zu einer C++-Anweisung benötigt eine Prä-

prozessor-Direktive kein Semikolon am Ende. Im Gegenteil, ein Semikolon am Ende kann gar zu unerwarteten Ergebnisse führen. Längere Direktiven können Sie in der nächsten Zeile fortsetzen, wenn Sie ein Backslash-Zeichen an das Zeilenende setzen. Pro Zeile ist eine Direktive erlaubt.

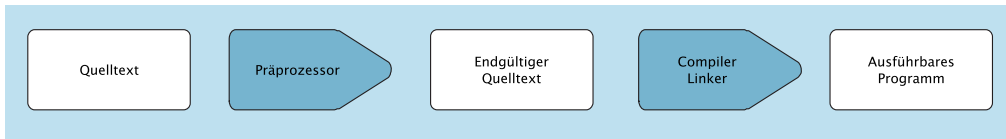


Abbildung 6.1 Der Compiler bzw. Linker bekommt den Quelltext erst nach dem Präprozessorlauf.

6.5.1 #include

Die Direktive `#include` haben Sie bereits in jedem Ihrer Programme verwendet. Mit `#include` wird der benötigte Quellcode der Headerdatei im Quelltext einkopiert. Hierbei gibt es folgende zwei Versionen:

```
01 #include <datei>
02 #include "datei"
```

Wenn Sie die Headerdatei wie in Zeile **01** zwischen die spitzen Klammern `<>` stellen, wird in einem voreingestellten Pfad nach der Datei gesucht. Der voreingestellte Pfad wird in der Regel von Ihrem Compiler oder der Entwicklungsumgebung festgelegt. In der Praxis können Sie weitere Pfade selbst hinzufügen. Wie dies geht, hängt vom System und dem verwendeten Compiler/der Entwicklungsumgebung ab.

Schreiben Sie hingegen die Datei zwischen Anführungszeichen `" "` wie in Zeile **02**, wird zunächst im lokalen Verzeichnis gesucht, wo Sie die Quelldatei gespeichert haben. Wird dort `"datei"` nicht gefunden, wird nochmals im voreingestellten Pfad gesucht, wie dies mit `<datei>` geschieht. In der Praxis wird die Version mit den Anführungszeichen für eigene Headerdateien verwendet.

Groß- und Kleinschreibung bei Headerdateien

Ein wichtiger Hinweis sollte hier noch angebracht werden, bevor Sie eigene Headerdateien schreiben. Beachten Sie, dass C++ nicht wissen kann, wie das Dateisystem mit Dateinamen umgeht. Daher kann nicht genau gesagt werden, ob beim Einkopieren zwischen Groß- und Kleinschreibung unterschieden wird. Persönlich sind meine Erfahrungen, dass Compiler unter Unix/Linux/Mac OS X ganz strikt auf Groß- und Kleinschreibung achten, während es bei Compilern unter MS-Windows in der Regel egal ist, ob hier `#include "datei"` oder `#include "Datei"` steht.

Relativer und absoluter Pfad

Zwischen den Anführungszeichen können Sie auch den relativen oder absoluten Pfad angeben:

```
#include "/home/wolf/cpp-examples/kap006/meinHeader.h"
#include "kap006/meinHeader.h"
#include "../kap006/meinHeader.h"
```

Dem Compiler das include-Verzeichnis mitteilen

Auch hier kann die Verwendung von relativen und absoluten Pfadangaben zu Problemen führen, wenn Sie den Code auf andere Systeme portieren wollen. Die klassische Fehlermeldung hierzu lautet: **meinheader.h: no such file or directory**. Da kann es dann bspw. nötig sein, ein Laufwerk anzugeben (bspw. unter MS-Windows), oder der Pfad muss komplett umgeändert werden. Auch hierfür bietet fast jeder Compiler eine Option an, den include-Pfad per Argument (bspw. `-I include-pfad` beim `g++` bzw. `clang` oder `/I include-pfad` bei MS Visual C++) zur Übersetzung mit zu übergeben. Hierzu empfiehlt es sich, bei Bedarf wieder in der Dokumentation Ihres Compilers nachzulesen.

6.5.2 #define und #undef

Mit der `#define`-Direktive können Dinge wie symbolische Konstanten, Makros, bedingte Kompilierung und Fehlersuche realisiert werden. Mit `#define` können Sie quasi einen Text vom Präprozessor durch einen anderen Text ersetzen lassen. Allerdings, um es nochmals zu erwähnen, bleibt es trotzdem eine Direktive, die vom Präprozessor durchgeführt wird, welcher keine Ahnung von C++ hat. Daher sollten Sie auch echte C++-Dinge in der Praxis verwenden und symbolische Konstanten bzw. Makros mit echten C++-Techniken realisieren.

#define und C++

In der modernen C++-Praxis benötigen Sie die `#define`-Direktive eigentlich nur noch für Dinge wie die bedingte Kompilierung und vielleicht noch zur Fehlersuche (Debugging). Ansonsten gibt es für alles andere bessere Alternativen.

Symbolische Konstanten und Makros

Symbolische Konstanten können mit der `#define`-Direktive verwendet werden, wenn bestimmte konstante Werte in einem umfangreichen Programm mehrmals verwendet werden sollen, bspw.:

```
#define PI 3.1415f // Symbolische Konstante
```

Der Präprozessor würde jetzt vor dem Compilerlauf alle im Quelltext vorhandenen Textfolgen PI durch die Textfolge 3.1415f ersetzen. Hier ist bewusst noch die Rede von einer Textfolge, weil es für den Präprozessor nichts anderes ist.

Bei einem #define-Makro ist dies im Grunde nicht viel anders, nur dass Sie hier auch noch Parameter hinzufügen können, bspw.:

```
#define HOCH2(val) ( val * val ) // Makro
```

Steht jetzt im Quelltext die Textfolge HOCH2(10), ersetzt der Präprozessor diesen Text durch die Textfolge 10 * 10. Hierzu ein kurzes und einfaches Beispiel:

```
00 // kapitel06/define01.cpp
01 #include <iostream>
02 #include <cmath> // für atan()

03 #define PI atan(1)*4
04 #define DE_DE "Kreisfläche   :"
05 #define EN_EN "area of a disk :"
06 #define HOCH2(val) ( val * val )

07 int main() {
08     std::cout << DE_DE << HOCH2(10)*PI << std::endl;
09     double a = HOCH2(10+1)*PI;
10     std::cout << EN_EN << a << std::endl;
11 }
```

Der Präprozessor ersetzt jetzt die in den Zeilen 03 bis 06 verwendeten symbolischen Konstanten und Makros an den passenden Stellen im Quellcode in den Zeilen 07 bis 11. Der Compiler bekommt dann, auf die Zeilen 07 und 11 beschränkt, folgenden Code nach der Textersetzung des Präprozessors zu sehen:

```
...
07 int main() {
08     std::cout << "Kreisfläche   :" << 10*10*3.1415 << std::endl;
09     double a = 10+1*10+1*3.1415;
10     std::cout << "area of a disk   :" << a << std::endl;
11 }
```

Leider enthält dieses Beispiel auch einen unangenehmen Seiteneffekt in der Berechnung in Zeile 09, wenn Sie das Ergebnis aus Zeile 10 mit dem Ergebnis aus Zeile 08 vergleichen. Das Ergebnis aus Zeile 10 ist erheblich kleiner als das Ergebnis aus Zeile 08, obwohl wir ja einen höheren Werte 10+1 verwendet haben. Der Seiteneffekt liegt

an der Punkt-vor-Strich-Regelung, welche hier mit $((10 + (1 * 10)) + (1 * 3.1415))$ rechnet. Den Fehler können Sie beheben, wenn Sie beim Makro HOCH2 entsprechend die Klammern setzen:

```
06 #define HOCH2(val) ( (val) * (val) )
```

Allerdings sind es u.a. solche Seiteneffekte, wie hier mit den vergessenen Klammern, die immer wieder zu Problemen in Bezug auf Makros führen. Bedenken Sie, dass dieses Beispiel wieder recht trivial ist und der Fehler daher leicht zu finden war. Bei komplexeren Makros macht die Fehlersuche (Debugging) allerdings keinen Spaß mehr, wenn es sich um Makros handelt. Neben diesem Seiteneffekt wäre noch die mangelnde Typsicherheit zu bemängeln. Auch die Laufzeit wird negativ beeinflusst, wenn man Ausdrücke wie `#define PI atan(1)*4` verwendet, weil bei jedem Vorkommen von `PI` die Berechnung erneut durchgeführt werden muss. Eine bessere Alternative dazu haben Sie ja bereits im Abschnitt »constexpr, das stärkere const« in Abschnitt 6.4.3 kennengelernt. Auch der Gültigkeitsbereich von Makros ist ab der Position der `#define`-Direktive global sichtbar und lässt sich nicht in einen Namensraum einbetten.

Alternativen für #define-Makros und #define-Konstanten

`#define`-Direktiven sollen hier jetzt nicht ganz verteufelt werden. Bei Dingen wie der bedingten Übersetzung oder für die Fehlersuche sind Makros immer noch sehr wichtig. Aber für Makros und symbolische Konstanten sollten Sie echte C++-Techniken vorziehen. Statt symbolischer Konstanten bieten sich `const` (siehe Abschnitt 6.3.1, »Das Schlüsselwort const«) oder `enum` (siehe Abschnitt 4.7, »Aufzählungstypen (C++11-Version)«) an. Die besseren C++-Alternativen für Makros sind `inline`-Funktionen (siehe Abschnitt 5.8.1, »Inline-Funktionen«) und natürlich die Funktions-Templates (siehe Abschnitt 11.1, »Funktions-Templates«). Wenn Sie den neueren C++11-Standard verwenden können, dann würde ich Ihnen `constexpr` (siehe Abschnitt »constexpr« in Abschnitt 6.4.3) empfehlen.

Symbol und Makro aufheben

Soll eine symbolische Konstante oder ein Makro ab einer bestimmten Stelle im Programm nicht mehr gültig sein, können Sie diese mit der Direktiven `#undef` aufheben:

```
#undef SYMB_KONST
```

Ab der Position, wo Sie die dieses `#undef` setzen, führt der Präprozessor keine Ersetzung mehr von `SYMB_KONST` durch. Befindet sich hinter einer `#undef`-Direktiven trotzdem noch ein `SYMB_KONST`, hängt es vom Compiler ab, was als Nächstes geschieht – meistens wird hier wohl eine Fehlermeldung ausgegeben.

In der Praxis verwendet man `#undef`, um eine symbolische Konstante zunächst zu löschen und anschließend mit einem neuen Wert zu versehen. Auch lassen sich so symbolische Konstanten von Standard-Headerdateien aufheben und mit neuen Werten versehen. Hierzu ein kurzes Beispiel:

```
00 // kapitel06/define02.cpp
...
01 #define A "Kreisfläche : "
02 const double PI = atan(1)*4;

03 int main() {
04     std::cout << A << 10*10*PI << std::endl;
05 #undef A
06     double A = 20*20*PI;
07     std::cout << A << std::endl;
08 #define A "Andere Bedeutung :"
09     std::cout << A << std::endl;
10 }
```

Das Beispiel soll hier keine Schule machen. In Zeile **04** ersetzt der Präprozessor die symbolische Konstante `A` durch das Stringliteral `"Kreisfläche :"`. Diese symbolische Konstante wird in Zeile **05** aufgehoben. Der Präprozessor lässt daher den Bezeichner der Variablen `A` vom Typ `double` in den Zeilen **06** und **07** unberührt. Erst in Zeile **08** bekommt `A` als symbolische Konstante für den Präprozessor wieder eine Bedeutung, so dass in Zeile **09** das `A` durch das Stringliteral `"Andere Bedeutung :"` ersetzt wird.

6.5.3 Bedingte Kompilierung

Bei der bedingten Kompilierung können Sie beeinflussen, welchen Code der Compiler unter einer bestimmten Bedingung erhält und welcher ausgeblendet werden soll. Natürlich sind dies auch wieder nur Präprozessor-Direktiven, aber es kann recht nützlich sein, ganze Codeblöcke ein- bzw. auszublenden. Auch bei der Suche nach Fehlern (Debugging) kann eine solche bedingte Kompilierung sehr hilfreich sein.

Da es hierfür recht viele, aber trotzdem überschaubare Direktiven gibt, finden Sie hier alle in Tabelle 6.1 aufgelistet.

Direktive	Bedeutung
#if AUSDRUCK	Wenn AUSDRUCK ungleich 0 ist, bekommt der Compiler den dahinterstehenden Quellcode zum Übersetzen.

Tabelle 6.1 Präprozessor-Direktiven für die bedingte Kompilierung

Direktive	Bedeutung
<code>#ifdef SYMBOL</code> <code>#if defined SYMBOL</code> <code>#if defined (SYMBOL)</code>	Ist das <code>SYMBOL</code> definiert (bspw. mit <code>#define</code>), bekommt der Compiler den darauffolgenden Quellcode.
<code>#ifndef SYMBOL</code> <code>#if !defined SYMBOL</code> <code>#if !defined (SYMBOL)</code>	Ist das <code>SYMBOL</code> nicht definiert, bekommt der Compiler den darauffolgenden Quellcode.
<code>#elif SYMBOL</code> <code>#elif AUSDRUCK</code>	Ist das <code>SYMBOL</code> definiert (bspw. mit <code>#define</code>) oder der AUSDRUCK ungleich 0, erhält der Compiler den dahintergeschriebenen Programmteil. Einem <code>#elif</code> geht immer ein <code>#if</code> oder <code>#ifdef</code> voraus.
<code>#else</code>	Hier können Sie einen alternativen Programmteil dahinterschreiben, den der Compiler erhält, wenn alle vorangegangenen <code>#if</code> , <code>#ifdef</code> usw. nicht zutreffend waren.
<code>#endif</code>	Damit wird dem Präprozessor das Ende der bedingten Kompilierung mitgeteilt.

Tabelle 6.1 Präprozessor-Direktiven für die bedingte Kompilierung (Forts.)

Mehrfaches Inkludieren vermeiden

In fast jeder Headerdatei finden Sie ein Konstrukt der bedingten Kompilierung, welches überprüft, ob eine Headerdatei bereits inkludiert wurde oder nicht, damit diese nicht noch ein zweites Mal hinzugefügt wird. Folgendes Konstrukt sollten Sie also auch künftig in Ihrer Headerdatei hinzufügen:

```
00 // meinHeader.h
01 #ifndef MEIN_HEADER_H
02 #define MEIN_HEADER_H
03 // Hier kommt der Code für meinHeader.h
...
04 #endif
```

In Zeile **01** überprüfen Sie zunächst, ob das Symbol `MEIN_HEADER_H` noch **nicht** vorhanden ist. Trifft dies zu, wird das Symbol in Zeile **02** definiert. Anschließend folgt (ab Zeile **03**) der Code für die Headerdatei `meinHeader.h`, der vom Präprozessor für den Compiler in die Datei einkopiert werden soll. Das Ende der bedingten Kompilierung zeigen Sie dem Präprozessor mit der `#endif`-Direktiven in Zeile **04** an.

Falls die Bedingung in Zeile **01** nicht zutrifft und das Symbol `MEIN_HEADER_H` bereits definiert wurde, werden die Zeilen dahinter nicht mehr erneut ausgeführt, weil der Code der Headerdatei ja bereits einkopiert wurde, und somit wird die Arbeit des Präprozessors hinter `#endif` (Zeile **04**) weitergeführt.

Compiler- und systemspezifische bedingte Kompilierung

Das Thema geht über den C++-Standard hinaus und ist, wie Sie am Titel ablesen können, compiler- und systemspezifisch. Trotzdem kann es auf jedem System verwendet werden. Wenn ein Symbol nicht existiert, wird der darauffolgende Code ignoriert. Ein einfaches Beispiel soll Ihnen diese bedingte Kompilierung demonstrieren:

```
00 // kapitel06/define03.cpp
...
01 #ifdef __WIN32__
02     #define OS "MS Windows"
03 #elif __APPLE__
04     #define OS "Apple"
05 #elif __linux__
06     #define OS "Linux"
07 #elif __unix__
08     #define OS "Unix"
09 #else
10     #define OS "Unbekannt"
11 #endif

12 int main() {
13     std::cout << "Ihr System: " << OS << std::endl;
14 }
```

In den Zeilen **01**, **03**, **05** und **07** wird jeweils geprüft, ob in der Umgebung die symbolische Konstante `__WIN32__` (für MS-Windows), `__APPLE__` (für Macintosh Apple), `__linux__` (für Linux-Systeme) oder `__unix__` (für Unix-Systeme) definiert ist. Ist die entsprechende symbolische Konstante vorhanden, setzen wir in den Zeilen **02**, **04**, **06** oder **08** die symbolische Konstante `OS` mit einem entsprechenden Stringliteral. Ist keines der Symbole bekannt, wird die `#else`-Direktive aktiv, und wir setzen `OS` auf "Unbekannt". Entsprechend der gesetzten symbolischen Konstante für `OS` ersetzt der Präprozessor in Zeile **13** die Konstante durch das Stringliteral und gibt im Idealfall das Betriebssystem aus, auf dem das Programm übersetzt wurde.

Weitere Symbole für Compiler und Systeme

Neben den hier verwendeten symbolischen Konstanten für Betriebssysteme gibt es noch eine Menge mehr – auch compilerspezifische. Allerdings geht dies, wie bereits eingangs erwähnt, über den normalen C++-Standard hinaus und ist daher auch wirklich compiler- und systemabhängig.

Hilfe bei der Fehlersuche

Ebenfalls recht hilfreich kann eine solche bedingte Kompilierung bei der Suche nach Fehlern sein. Betrachten Sie hierzu folgendes einfache Beispiel:

```
00 // kapitel06/define04.cpp
...
01 #define DEBUG 2

02 int main() {
03     double r=0.0, A=0.0, pi=atan(1)*4;
04     std::cout << "Bitte Radius eingeben: ";
05     std::cin >> r;
06     #if DEBUG > 0
07     std::cout << "DEBUG: " << r << std::endl;
08     #endif
09     A = r*r*pi;
10     #if DEBUG > 1
11     std::cout << "DEBUG: " << r << std::endl;
12     #endif
13     std::cout << "Kreisfläche: " << A << std::endl;
14     // Weitere Berechnungen mit r ...
...
15 }
```

Am Anfang des Programms finden Sie in Zeile **01** eine symbolische Konstante `DEBUG` mit dem Wert 2 definiert. Den Wert können Sie jederzeit verändern. Dadurch dass `DEBUG` auf 2 gesetzt ist, werden die Zeilen **07** und **11** mit an den Compiler gegeben und entsprechende Debugging-Informationen auf dem Bildschirm ausgegeben. Im Beispiel wurde der Wert `r` (für Radius) überwacht, welcher irgendwo versehentlich nach der Eingabe geändert wird, womit die späteren Berechnungen zu falschen Ergebnissen führen. Setzen Sie den Wert der symbolischen Konstante auf 1, wird nur noch die Debug-Information aus Zeile **07** ausgegeben, weil die Bedingung in den Zeilen **10** bis **12** nicht mehr zutreffend ist. Setzen Sie hingegen `DEBUG` gar auf 0, erfolgt gar keine Debug-Ausgabe mehr auf dem Bildschirm.

6.5.4 Weitere Direktiven

Hier fehlen noch einige Präprozessor-Direktiven (Tabelle 6.2), welche recht nützlich sein können.

Direktive	Bedeutung
#error "nachricht"	Schreiben Sie diese Direktive in den Quellcode, wird die Übersetzung mit der Fehlermeldung "nachricht" abgebrochen. Dies kann hilfreich sein, wenn Sie bei einem Projekt an einem Quellcode arbeiten, der noch nicht funktioniert und Sie andere darauf hinweisen wollen.
#line n "datei"	Damit können Sie die Zeilennummer im Quelltext auf n und den Dateinamen auf "datei" setzen. Diese Direktive hat einen Einfluss auf die symbolischen Konstanten __LINE__ und __FILE__.
#pragma	Dies sind compilerspezifische Direktiven. Wenn ein Compiler eine bestimmte #pragma-Direktive nicht kennt, wird diese ignoriert. Damit lassen sich Direktiven verwenden, ohne mit anderen Compilern in Konflikt zu geraten.

Tabelle 6.2 Weitere Präprozessor-Direktiven

Zusätzlich gibt es in C++ noch einige vordefinierte symbolische Konstanten, welche ich Ihnen hier auch nicht vorenthalten will (Tabelle 6.3).

Symbol	Bedeutung
__LINE__	Darin ist die aktuelle Zeilennummer des Quellcodes enthalten. Kann nützlich für Debugging-Zwecke sein.
__FILE__	Darin ist die Zeichenkette der aktuellen Datei enthalten.
__DATE__	eine Zeichenkette mit dem aktuellen Datum
__TIME__	eine Zeichenkette mit der aktuellen Uhrzeit

Tabelle 6.3 Vordefinierte symbolische Konstanten in C++

6.6 Modulare Programmierung – Code organisieren

Wenn der Quellcode umfangreicher wird und mehrere Entwickler an einem Projekt arbeiten, wird in der Regel nicht nur mit einer Quelldatei gearbeitet. Hierbei werden in der gängigen Praxis mehrere Quell- und Headerdateien – auch Module genannt – verwendet.

Nur eine kurze Einführung

Das Thema hier ist nur eine kurze Einführung und stellt auch keine ultimative Vorgehensweise dar – die gibt es nämlich nicht. Trotzdem gibt es einige Richtlinien, die Ihnen helfen, bei umfangreicheren Projekten den Überblick zu bewahren, wenn Sie sich daran orientieren. Das Thema hängt zudem auch von der verwendeten Entwicklungsumgebung, der Anzahl der mitwirkenden Programmierer und zum Teil auch vom verwendeten System ab. Das bedeutet letztendlich, dass Sie bei umfangreicheren Projekten nicht darum herumkommen, sich intensiver mit den Details zu befassen.

6.6.1 Module

Ein Modul ist ganz einfach eine Datei mit einer Sammlung von Funktionen oder Klassen. Idealerweise werden diese Dateien in sinnvolle Gruppen aufgeteilt. Funktionen und Klassen, welche bspw. die Netzwerkprogrammierung oder die Datenbankprogrammierung betreffen, steckt man gewöhnlich jeweils in ein extra Modul. Dass die einzelnen Module dann auch noch in eine Quelldatei (**.cpp*) und Headerdatei (**.h*) aufgeteilt werden, sei jetzt nur am Rande erwähnt.

Der Vorteil solcher logischen Strukturierungen ist, dass Sie solche Module jederzeit auch wieder in anderen Projekten verwenden können. Neben der Wiederverwendbarkeit von Modulen stellt ein Aufteilen von Quellcode die beste Möglichkeit dar, in einer Gruppe von mehreren Personen zu programmieren. Jeder Programmierer erhält eine Aufgabe für ein spezielles Modul. Natürlich ist es auch möglich, dass mehrere Entwickler an derselben Datei arbeiten. Allerdings wird hierfür in der Praxis dann noch eine Versionsverwaltungssoftware verwendet, damit es hier nicht zu einem Chaos kommt. Auch muss ein Projekt dank solcher Module nicht immer wieder komplett übersetzt werden.

Sie ahnen es, dass eine solche Projektverwaltung bei großen Projekten eine umfangreichere und aufwendige Sache sein kann. Es gibt hier zwar keine festen Regeln, wie Sie Ihre Module aufteilen können, aber Sie sollten immer versuchen, ein Modul so zu schreiben, dass es selbstständig arbeiten kann. Genauer: Die Notwendigkeit des Austauschs von Informationen zwischen den Modulen sollte möglichst gering sein.

6.6.2 Sinnvolle Quellcodeaufteilung

In der Praxis wird ein Modul in zwei Teile aufgeteilt, einen öffentlichen und einen versteckten Bereich (Abbildung 6.2):

- Öffentliche Schnittstelle: In der Regel ist es die Headerdatei (meistens mit der Endung **.h*), die die öffentliche Schnittstelle für Funktionen und Klassen anbietet

– genauer sämtliche öffentlichen Deklarationen. Mithilfe dieser Schnittstelle weiß der Benutzer, wie er die Funktionen und Klassen im Modul verwenden kann.

- Private Dateien: Darin finden Sie gewöhnlich die eigentliche Implementation (den eigentlichen arbeitenden Quellcode) der Schnittstellen und Klassen – also die Definitionen. Die privaten Dateien liegen dabei nicht immer als lesbare Quelldatei (mit der Endung *.cpp*) vor, sondern sind häufig auch nur als vorübersetzte Objektdatei (Endung: *.o* oder *.obj*) oder gar als Bibliotheksdatei vorhanden.

Objekt- und Bibliotheksdatei sind nicht portabel

Natürlich sollte Ihnen klar sein, dass Sie eine Objektdatei oder eine Bibliotheksdatei nicht mehr auf anderen Plattformen verwenden können. Eine für Windows erstellte Objekt- oder Bibliotheksdatei lässt sich nicht unter Mac OS X oder Linux verwenden.

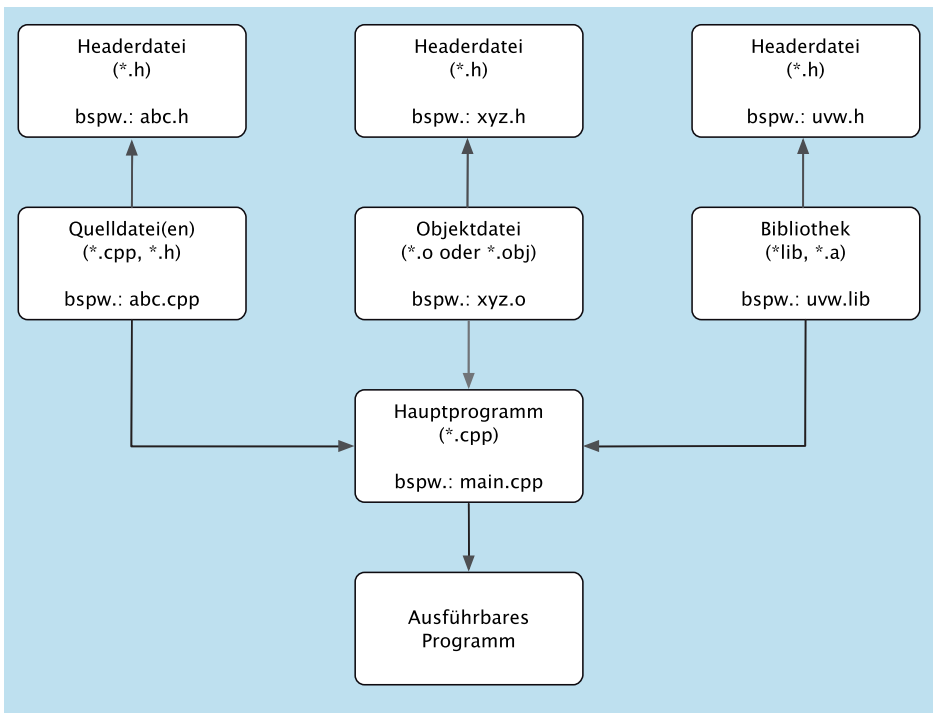


Abbildung 6.2 Eine theoretische Aufteilung von Modulen

Damit es mit der Übersetzung der einzelnen Module auch klappt, müssen Sie dem Compiler bzw. Linker mitteilen, wo er diese finden kann (Abbildung 6.3). Neben dem Inkludieren der Headerdatei(en) mittels `#include` müssen Sie bei Ihrem Projekt auch die Quelldatei (*.cpp*) mit angeben. Der Compiler bzw. Linker kann nicht wissen, was

alles zum Projekt gehört. Bei Entwicklungsumgebungen müssen Sie dem Projekt hierbei gewöhnlich nur eine neue Datei (in der Projektverwaltung) hinzufügen. In der Kommandozeile hingegen geben Sie einfach diese Quelldatei noch zusätzlich bei der Übersetzung mit an. Analog gilt dies auch, wenn es sich um eine Objekt- oder Bibliotheksdatei handelt.

Quelldatei inkludieren

Manchmal sieht man leider auch verzweifelte Inkludierungen von Quelldateien wie:

```
#include "abc.cpp"
```

Es ist zwar nicht falsch, so etwas zu machen, aber doch eher unüblich. Häufig kommt noch hinzu, dass man gar keine Quelldatei hat, sondern nur auf die Headerdatei mit Objekt- oder Bibliotheksdatei beschränkt ist.

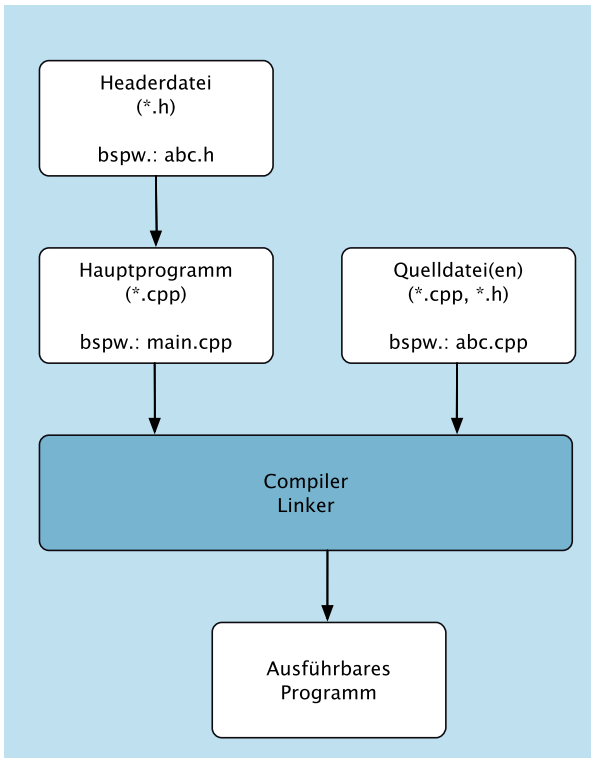


Abbildung 6.3 Damit es mit der Übersetzung der einzelnen Module klappt, müssen immer alle zu einem Projekt gehörenden Dateien angegeben bzw. inkludiert werden.

Wichtig bei einem Projekt mit Modulen ist natürlich vor allem auch, dass dem Compiler bzw. Linker die einzelnen Module auch bekannt sind. Während eine Header-

datei noch in den Quellcode geschrieben wird, muss die Quelldatei (bzw. auch Objekt- und/oder Bibliotheksdatei) dem Projekt bei der Übersetzung hinzugefügt werden. Wie dies genau funktioniert, hängt von der Entwicklungsumgebung oder dem verwendeten Compiler in der Kommandozeile ab.

6.6.3 Die öffentliche Schnittstelle (Headerdatei)

Die öffentliche Schnittstelle (Endung **.h* oder manchmal auch **.hpp*) ist für die Vorbereitung und auch für den Anwender des Moduls enorm wichtig. Gerade wenn die private Datei des Moduls nur als Objekt- oder Bibliotheksdatei mitgeliefert wird, erkennt der Benutzer anhand der Schnittstelle(n), wie er die Funktionen und Klassen des Moduls verwenden kann. Bevorzugt schreibt man in einer Headerdatei die Deklarationen und vereinzelt auch einige Definitionen. In Tabelle 6.4 finden Sie eine kurze Übersicht, was in einer solchen Headerdatei für gewöhnlich enthalten sein kann.

Element	Beispiel
Aufzählungen	<code>enum class Color { RED=1, GREEN, BLUE };</code>
Bedingte Kompilierung	<code>#ifndef __SYMBOL__ #define __SYMBOL__ ... #endif</code>
Bezeichner deklarieren	<code>class EineKlasse;</code>
Funktionen deklarieren	<code>int simple_function(const int v);</code>
include-Direktiven	<code>#include <headerdatei01> #include "headerdatei02"</code>
inline-Funktionen	<code>inline int foo(const int v) { return v*v; }</code>
Kommentare	<code>// Kommentar in einer Zeile</code>
Konstanten definieren	<code>static const double PI = 3.1415;</code>
Symbole, Makros	<code>#define __SYMBOL__ xyz</code>
Namensbereiche	<code>namespace myNamespace { int a, b, c; int simple_func(const int val); };</code>

Tabelle 6.4 Dinge, die man gewöhnlich in einer Headerdatei findet

Element	Beispiel
Templates definieren	<pre>template<class T1> class Cxyz { ... };</pre>
Templates deklarieren	<pre>template<class T1> class Cxyz;</pre>
Typdefinition	<pre>struct bar{ std::string str01; bar* next; };</pre>

Tabelle 6.4 Dinge, die man gewöhnlich in einer Headerdatei findet (Forts.)

Hierzu eine sehr einfach gehaltene kurze Headerdatei *abc.h* zur Demonstration:

```
00 // kapitel06/abc/abc.h
01 #include <iostream>
02 #ifndef ABC_H
03 #define ABC_H

04 namespace myMath {
05     static const double PI=3.1415;
06     double area_of_a_disk( const double radius );
07 }
08 #endif
```

In der Headerdatei finden Sie eine Funktionsdeklaration (siehe Abschnitt 5.3, »Funktionen deklarieren (Vorausdeklaration)«) und die Definition einer Konstante (siehe Abschnitt 6.3.1, »Das Schlüsselwort const«) in einen Namensraum (siehe Abschnitt 6.1, »Namensräume«) verpackt. Natürlich wird hier auch der Präprozessor-Teil mit `#ifndef` für die bedingte Kompilierung verwendet (siehe Abschnitt 6.5.3, »Bedingte Kompilierung«).

6.6.4 Die privaten Dateien

Die eigentlichen Implementierungen der Schnittstellen (also Definitionen) von Funktionen, Klassen usw. werden gewöhnlich in der (privaten) Quelldatei mit der Endung **.cpp* geschrieben. Als Namen verwendet man hierfür meistens denselben Namen wie für die dazugehörige Headerdatei. In unserem Beispiel wäre dies somit *abc.cpp*:

```
00 // kapitel06/abc/abc.cpp
01 #include "abc.h"
```

```

02 namespace myMath {
03     double area_of_a_disk( const double radius ) {
04         return radius * radius * PI;
05     }
06 }

```

Hier finden Sie jetzt im erneut geöffneten Namensbereich die Definition der Funktion `area_of_a_disk()`. Auch die Konstante `PI` aus der Headerdatei `abc.h` wird hier verwendet. Die Headerdatei `abc.h` muss logischerweise auch eingebunden werden.

Randnotiz

Gewöhnlich werden in die private Datei auch anonyme Namensbereiche und exportierte Template-Definitionen reingeschrieben.

Objektdatei erstellen

An dieser Stelle sei angemerkt, dass Sie aus `abc.cpp` und `abc.h` bereits eine Objekt- oder gar Bibliotheksdatei machen könnten. Hierzu ist keine `main`-Funktion nötig. Um eine solche Objektdatei zu erstellen, wird `abc.cpp` und `abc.h` einfach nur kompiliert – ohne Linkerlauf.

Eine so erstellte Objektdatei `abc.o` bzw. `abc.obj` und die Headerdatei `abc.h` können Sie jetzt weitergeben. Den Quellcode von `abc.cpp` müssen Sie so nicht freigeben, und dieser bleibt verborgen. Beim Übersetzen eines ausführbaren Programms müssen Sie hierbei die Headerdatei `abc.h` mit einbinden und dem Linker mitteilen, wo dieser die Objektdatei finden kann. Fast analog verläuft dies beim Erstellen von Bibliotheksdateien. Natürlich sollten Sie wissen, dass solche Objekt- bzw. Bibliotheksdateien nicht mehr portabel und somit systemabhängig sind. Eine auf Linux erstellte Objektdatei kann nicht mehr unter Windows verwendet werden.

6.6.5 Die Client-Datei

Die *Client-Datei* gibt es eigentlich nicht als Begriff, aber ich verwende diesen Begriff relativ gerne, weil es sich hierbei um die Datei handelt, welche auf das (oder die) Modul(e) zugreift – und meistens das Hauptprogramm `main()` enthält. In der Client-Datei müssen Sie dem Projekt nur noch die Headerdatei(en) und die Quelldatei(en) hinzufügen. Hier also die Client-Datei `main.cpp`:

```

00 // kapitel06/abc/main.cpp
01 #include <iostream>
02 #include "abc.h"

```

```

03 int main() {
04     double r=0.0;
05     std::cout << "Radius : ";
06     std::cin >> r;
07     double A = myMath::area_of_a_disk(r);
08     std::cout << "Kreisfläche : " << A << std::endl;
09 }

```

Wie bereits eingangs erwähnt, ist es wichtig, dass Sie dem Projekt die private Quelldatei hinzufügen oder sie in der Kommandozeile mit angeben. Beim Beispiel mit *abc.cpp* und *abc.h* sieht die Übersetzung bei mir in der Kommandozeile mit dem `g++` (mit `clang++` ist dies analog) wie folgt aus (alle Dateien liegen hier im selben Verzeichnis, in dem ich mich gerade auch befinde):

```

J.Wolf $ g++ -o main main.cpp abc.cpp -Wall -pedantic -std=c++11
J.Wolf $ ./main
Radius : 123.123
Kreisfläche : 47622.9
J.Wolf $

```

Bei einer Entwicklungsumgebung ist dies im Prinzip recht ähnlich, nur dass Sie hierbei eben die Datei(en) dem Projekt hinzufügen müssen (Abbildung 6.4). Es reicht also nicht nur aus, einfach eine neue Datei anzulegen.

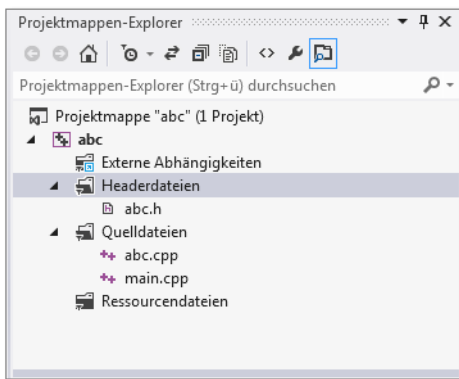


Abbildung 6.4 Dank der Projektverwaltung ist es in einer Entwicklungsumgebung wesentlich einfacher den Überblick zu behalten, wenn einzelne Dateien ordentlich zum Projekt hinzugefügt wurden.

Nur wenn alle Dateien dem Projekt hinzugefügt wurden, klappt es auch bei der Entwicklungsumgebung mit mehreren Modulen. In Abbildung 6.4 sehen Sie den Projektmappen-Explorer von Microsofts Visual C++.

Kapitel 7

Grundlagen zu den Klassen

Bisher haben Sie sich noch auf die prozedurale Programmierung beschränkt. In diesem Kapitel werden Sie (endlich) die Grundlagen zur objektorientierten Programmierung (kurz OOP) kennenlernen.

Wichtiger Hinweis für C++-Einsteiger

In diesem Kapitel lernen Sie die wichtigsten Grundlagen zur OOP kennen. Es ist daher fast unumgänglich, dass Sie diese Beispiele auch auf einem Rechner testen, um zu verstehen, was genau passiert. Es ist sehr wichtig, dass Sie die Themen in diesem Kapitel verstanden haben, ehe Sie mit den weiteren Kapiteln im Buch fortfahren. Dieses Kapitel stellt somit eine Art »Schlüsselkapitel« für das Buch dar. Hier sind auch viele neuere C++11-Techniken hinzugekommen. Wenn Ihr Compiler kein C++11 kann, dann können Sie diese Abschnitte auch einfach nur durchlesen. Die Beispiele in den Nicht-C++11-Abschnitten sind so geschrieben, dass sie auf klassischen C++03-Compilern auch übersetzt und ausgeführt werden können. Die Listings finden Sie auf der Bonus-Seite zum Buch unter <https://ssl.galileo-press.de/bonus-seite> und auf der Buch-CD wieder.

Der bisherige Weg, den Sie gegangen sind (oder der bisher im Buch verwendet wurde), ist, eine Lösung für ein Problem in Form einer oder mehrerer Funktion(en) zu entwerfen. Idealerweise übergaben Sie der Funktion einzelne Parameter (Daten) für deren Arbeit, und es wurde, abhängig von der Lösung des Problems, ein Wert zurückgegeben. Die Lösung eines Problems wird auch als Algorithmus bezeichnet. Bei dieser Vorgehensweise spricht man von einer *prozeduralen Programmierung*. Gelegentlich findet man hier auch noch die Begriffe strukturierte Programmierung oder imperative Programmierung. Bei der prozeduralen Programmierung sind die Daten und Funktionen ziemlich losgelöst voneinander (Abbildung 7.1).

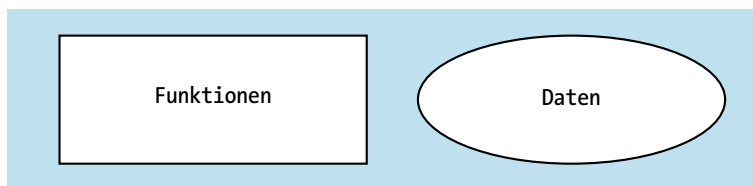


Abbildung 7.1 Bei der prozeduralen Programmierung gibt es Funktionen und Daten.

Bei der *objektorientierten Programmierung* hingegen ist das höchste Ziel, die Daten und Funktionen, welche auf diese Daten zugreifen können, zu einer Einheit, einem Objekt, zusammenzufassen (Abbildung 7.2). Funktionen fremder Objekte ist es somit nicht möglich (so sollte es zumindest sein), diese Daten aus Versehen zu ändern. Genau betrachtet, entspricht dieser objektorientierte Ansatz eher unserer realen Welt. Wenn Sie sich umsehen, werden Sie feststellen, dass unsere Welt im Wesentlichen aus Objekten besteht, welche auf die eine oder andere Art miteinander funktionieren.

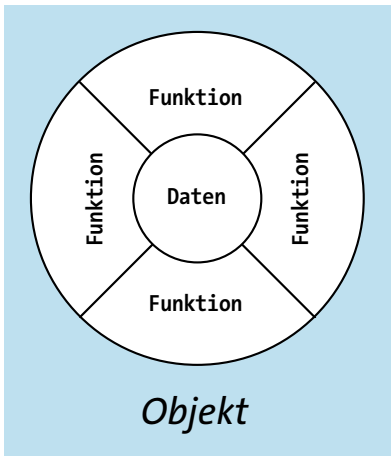


Abbildung 7.2 Bei der objektorientierten Programmierung werden Daten und Funktionen möglichst eng zu einem Objekt zusammengefasst.

Einfachstes und beliebtes Beispiel dürfte ein Objekt *Auto* sein. Betrachtet man ein Auto, stellt man fest, dass es verschiedene Eigenschaften wie Farbe, Gewicht, Leistung, Geschwindigkeit usw. besitzt. Neben den Eigenschaften kann ein Auto auch bestimmte Methoden ausführen, welche u. a. auch die Eigenschaften ändern können. So verändert bspw. die Methode »Fahren« die Eigenschaft »Geschwindigkeit« und die Methode »Bremsen« ebenfalls. Bezogen auf das vielleicht noch alte prozedurale Bild sind die Eigenschaften die Daten und die Methoden die Funktionen (Abbildung 7.3).

Begriffe: Funktionen, Elementfunktionen oder Methoden

Leider findet man in der Literatur und anderen Quellen häufig unterschiedliche Begriffe für die Funktionen einer Klasse. Gewöhnlich werden solche Funktionen als Methoden bezeichnet. Allerdings bezeichnet man Methoden nicht als solche, weil sie eine Funktion in einer objektorientierten Sprache sind, sondern weil eine Methode ein Teil eines Objekts ist.

Neben dem Begriff *Methode* findet man häufig auch noch den Begriff *Elementfunktion* (oder auch *Memberfunktion*). An dieser Stelle soll jetzt auch gar keine große Abhandlung über (die Begriffe) Methoden und Funktionen folgen. Verwenden Sie am einfachsten und auch aus didaktischen Gründen einfach den Begriff Methoden für die Funktionen einer Klasse und den Begriff Funktionen für Funktionen der prozeduralen Programmierung – so wie es in diesem Buch von nun auch gemacht wird.

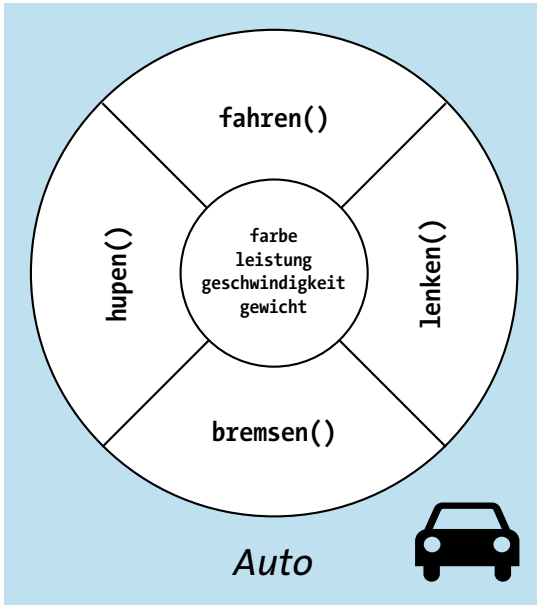


Abbildung 7.3 Ein Objekt Auto, wie man es aus dem Alltag kennt, stellt eine Sammlung von Eigenschaften (Daten) und die darauf arbeitenden Methoden (Funktionen) dar.

7.1 Prinzip von Klassen

Zur Verwaltung von gleichartigen Objekten verwendet man in C++ (und in vielen anderen Programmiersprachen auch) das *Konzept der Klassen*. Eine Klasse stellt quasi eine Vorlage dar, aus welcher in der Praxis eine oder mehrere *Instanzen* (bzw. Exemplare) erstellt werden können. Ein Objekt ist somit nichts anderes als eine Instanz (bzw. ein Exemplar) einer bestimmten Klasse. Bezogen nochmals auf unser Beispiel des Autos können Sie sich das Schreiben einer Klasse wie einen Konstruktionsplan für ein Auto vorstellen. Ein Konstruktionsplan ist ja noch kein Auto. Und so sind Klassen letztendlich nur Baupläne für Objekte, welche Sie erst zur Laufzeit des Programms (als Instanz) erstellen.

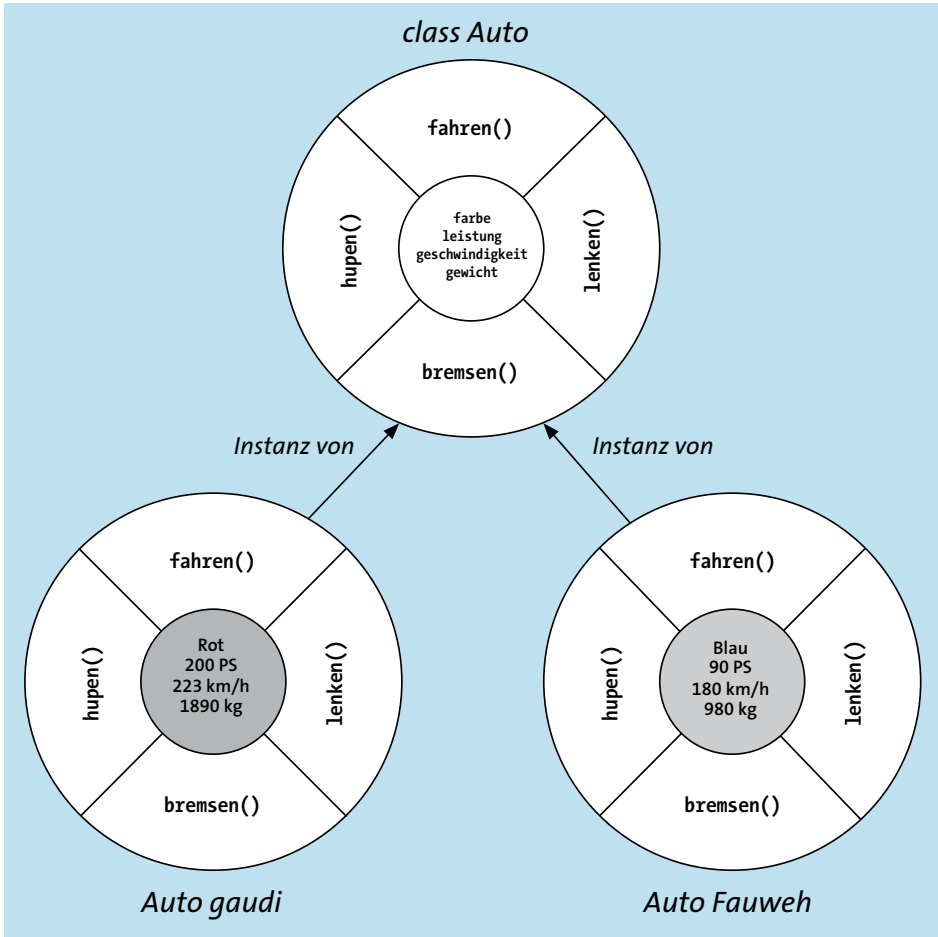


Abbildung 7.4 Gleichartige Objekte mit denselben Eigenschaften (Daten) und gleichen Methoden (aber unterschiedlichen Eigenschaftswerten) gehören zur selben Klasse. Somit ist jedes Objekt die Instanz einer Klasse.

Somit ist die Klasse praktisch ein Bauplan zum Entwerfen von Objekten mit Eigenschaften und Methoden (Abbildung 7.4). Genau genommen belegt eine Klasse selbst noch keinen Speicherplatz im eigentlichen Sinne, sondern erst, wenn Sie ein Objekt von dieser Klasse erstellt (instanziert) haben.

7.2 Klassen erstellen

Nachdem Sie jetzt erfahren haben, dass eine Klasse ein Bauplan für eine Reihe von gleichen bzw. ähnlichen Objekten ist, soll jetzt eine solche Klasse entworfen werden. Rein programmtechnisch haben Sie bereits Klassen in Abschnitt 4.5, »Strukturen«,

kennengelernt. In C++ ist nämlich jede Struktur (`struct`) oder gar eine Union (siehe Abschnitt 4.6, »Unions«) bereits eine Klasse. Allerdings wird in C++ gewöhnlich eine Klasse mit dem Schlüsselwort `class` erstellt. Auf den Unterschied zwischen `class` und `struct` wird im Abschnitt »Standardzugriffsrechte« in Abschnitt 7.6 eingegangen.

Wie bereits erwähnt, wird in C++ eine Klasse mit dem Schlüsselwort `class` realisiert. Für den Bezeichner gelten alle bekannten Regeln für gewöhnliche Datentypen. In der Praxis (keine Regel) wird der erste Buchstabe einer Klasse häufig großgeschrieben. Hierzu ein leeres Grundgerüst einer Klasse als Beispiel:

```
01 class Personal {
02     // Eigenschaften und Methoden von Personal
03 }; // <- Semikolon am Ende !!!
```

Eingeleitet wird eine Klasse mit dem Schlüsselwort `class`, gefolgt von einem gültigen Bezeichner, wie dies in Zeile 01 geschieht. Zwischen die geschweiften Klammern werden die Eigenschaften (Daten) und Methoden der Klasse geschrieben. Ganz wichtig ist es, den Bauplan einer Klasse mit einem **Semikolon abzuschließen**, wie es hier in Zeile 03 zu sehen ist. Tun Sie dies nicht, wird sich der Compiler bei Ihnen beschweren.

Module und Klassen

In Abschnitt 6.6, »Modulare Programmierung – Code organisieren«, sind wir ja bereits auf die Organisation von Code eingegangen. Bei Klassendefinitionen wird empfohlen, diese in einer Headerdatei (mit der Endung `*.h`) zu schreiben. Es wird außerdem empfohlen, nur ein Modul pro Klasse zu verwenden und dieses Modul auch wie die Klasse zu benennen.

Eigenschaften einer Klasse

Wenn Sie eine Klasse erstellen, müssen Sie sich schon vor dem Schreiben Gedanken machen, was Ihre Klasse alles für Eigenschaften besitzen soll. Ein sorgfältige Planung und Überlegung im Vorfeld kann Ihnen eine Menge an Nacharbeiten ersparen. Zwar ist es jederzeit möglich, einer Klasse nachträglich weitere Eigenschaften hinzuzufügen, aber hierbei müssen Sie dann auch andere Codestellen Ihres *Bauplans* anpassen.

In unserem Fall soll eine einfache Klasse `Personal` erstellt werden. Hierfür könnten Eigenschaften wie Name, Personalnummer, Berufsbezeichnung, Gehalt, Alter, Betriebszugehörigkeit usw. nötig werden. Für das Beispiel im Buch beschränken wir uns auf ein paar wenige Eigenschaften, um die Beispiele nicht zu umfangreich werden zu lassen. Wir begnügen uns bei unserer Klassendefinition zunächst mit dem Namen, der Personalnummer und dem Gehalt. Hierzu also unsere Klasse mit den eben erwähnten Eigenschaften:

```

00 // personal.h
01 #include <iostream>
02 #include <string>
03 #ifndef PERSONAL_H
04 #define PERSONAL_H

05 class Personal {
06     std::string name;        // Name
07     unsigned int pers_nr;    // Personalnummer
08     unsigned int gehalt;     // monatliches Gehalt
09 };
10 #endif

```

7.3 Objekte einer Klasse erzeugen

Es wurde eingangs schon umfassend erklärt, daher hier nur eine Kurzfassung, was ein Objekt und was der Unterschied zur Klasse ist: Eine Klasse ist im Grunde lediglich ein Bauplan (oder eine Vorlage) für ein Objekt. Ein Objekt wiederum ist nichts anderes als eine Instanz einer Klasse, die tatsächlich auch Speicherplatz benötigt.

Genau genommen haben Sie ja bereits mit den Strukturen mit `struct` in Abschnitt 4.5, »Strukturen«, Objekte erzeugt. Denn sobald man in C++ programmiert, ist auch eine Struktur eine Klasse. Auf den Unterschied zwischen `struct` und `class` wird in Abschnitt 7.6, »Exkurs: Zugriffskontrolle auf die Klassenmitglieder«, eingegangen. Somit haben Sie quasi schon Grundkenntnisse zu den Klassen.

Was enthält eine Instanz alles?

Mit jedem Objekt einer Klasse, welches Sie instanziierten, wird gleich Speicher für die Eigenschaften bereitgestellt. Bezogen auf die Klasse `Personal` hat somit jedes Objekt gleich Speicher für die Eigenschaften (= Daten) `name`, `pers_nr` und `gehalt`. Zwar hat jedes Objekt seine eigenen Eigenschaften, aber alle Objekte arbeiten mit denselben Methoden. Es wäre ja wirklich ein Overkill, wenn alle Objekte noch eigene Methoden (Funktionen der Klasse; siehe Abschnitt 7.7, »Methoden – die Funktionen der Klasse«) verwenden würden. Damit die Methode auch wirklich weiß, mit welchem Objekt sie gerade arbeitet, steht ein »unsichtbarer« `this`-Zeiger zur Verfügung, den Sie in Abschnitt 7.7.6, »This-Zeiger«, kennenlernen werden und welchen Sie sich wie folgt vorstellen können:

```

Personal personal;
Personal* const this = &personal;

```

Objekt erzeugen

Ein einzelnes Objekt wird somit mit folgender einfachen Syntax instanziiert:

```
Klassenname Bezeichner;
```

Bezogen auf unsere Klasse `Personal` könnten Sie wie folgt neue Objekte anlegen:

```
01 Personal person01;
02 Personal person02, person03;
```

In Zeile **01** wurde ein neues Objekt von `Personal` als Instanz der Klasse definiert. In Zeile **02** wurden zwei Objekte der Klasse `Personal` instanziiert. Der Zugriff auf die **öffentlichen** Mitglieder der Klasse `Personal` könnte jetzt wie bei den Strukturen über den Punktoperator erfolgen.

Vektor von Objekten erzeugen

Natürlich können Sie hier auch ganze Vektoren oder rohe C-Style-Arrays definieren:

```
01 #include <vector>
...
02 std::vector<Personal> zeitarbeiter(100);
03 std::vector<Personal> festanstellung;
04 Personal cStyleArray[20];
```

In Zeile **02** werden 100 Objekte der Klasse `Personal` angelegt. Zeile **03** hingegen legt einen leeren Vektor der Klasse `Personal` an. Und Zeile **04** zeigt außerdem, dass es theoretisch auch möglich ist, alte C-Style-Arrays anzulegen. Hier werden 20 Objekte der Klasse `Personal` angelegt. Allerdings werden C-Style-Arrays in C++ nicht empfohlen. Auch hier erfolgt der Zugriff auf die öffentlichen Mitglieder der Klasse `Personal` mithilfe des Punktoperators. Trotzdem muss hierbei dazwischen auch noch der Index des entsprechenden Elements mit `[n]` bzw. `at(n)` erfolgen.

Hier sollte noch angemerkt werden, dass `std::vector` seine Elemente automatisch initialisiert hat, das C-Array aber nicht.

Objekte dynamisch zur Laufzeit anlegen

Es ist zwar auch möglich, Objekte zur Laufzeit zu erzeugen, allerdings sollte man so etwas immer mit Bedacht einsetzen und sich überlegen, ob man nicht auf eine andere Alternative zurückgreifen sollte:

```
01 Personal *person01;
02 person01 = new Personal;
...
03 delete person01;
```


In Zeile 01 finden Sie einen `Personal`-Zeiger, mit dem Sie in Zeile 02 einen Speicher zur Laufzeit für ein Objekt `Personal` anfordern. Hierbei erfolgt der Zugriff auf die Mitglieder der Klasse `Personal` mit dem Pfeiloperator `->`. Am Ende müssen Sie den Speicher mit `delete` (Zeile 03) wieder freigeben.

Verzichten Sie auf `new` bei Objekten

Wenn es irgend geht, sollten Sie bei Objekten auf die dynamische Speicherreservierung verzichten und stattdessen auf eine der vielen Containerklassen, wie bspw. `vector`, zurückgreifen. Damit machen Sie sich zum einen das Leben erheblich einfacher, und zum anderen hat die Vergangenheit gezeigt, dass die Verwendung von `new` und `delete` zu vielen Fehlern (Stichwort Speicherlecks/Memory Leaks) geführt hat. Wenn Sie trotzdem eine dynamische Speicherreservierung benötigen, sollten Sie wenn möglich die neuen Smart Pointer von C++11 verwenden. Im konkreten Fall würde ich Ihnen zu `std::unique_ptr` raten (siehe Abschnitt 16.3.3, »Unique Pointer«). Der Vorteil ist damit, dass Sie sich nicht mehr um die Freigabe des Speichers mit `delete` kümmern müssen.

7.4 Klassen(-Eigenschaften) initialisieren

In den folgenden Abschnitten werden Sie zunächst verschiedene Möglichkeiten kennenlernen, wie Sie die Eigenschaften, also die Daten, Ihrer Klasse mit gültigen Werten initialisieren können. Das ist (und sollte es auch sein) einer der vielen Vorteile der OOP, dass es gar nicht mehr passieren kann oder sollte, dass mit nichtinitialisierten Speicherobjekten gearbeitet wird. Daher lernen Sie hier gleich von Anfang an, wie Sie es richtig machen können.

Als Beispiel dient unsere Klasse `Personal`, welche wir eben mit ein paar Eigenschaften wie Namen, `Personalnummer` und dem monatlichen Gehalt versehen haben. Unsere Klasse enthält ja auch Daten vom Typ `std::string`, bei denen wir uns ohnehin darauf verlassen können, dass der Name (hier ist die Eigenschaft `Personal::name` gemeint) für jedes Objekt automatisch immer mit einem leeren String initialisiert wird. Die beiden anderen Daten wie `Personalnummer` und das monatliche Gehalt haben allerdings ohne weitere Vorkehrungen einen undefinierten Wert. Wenn Sie also jetzt ein Objekt (oder eine Instanz von `Personal`) mit

```
Personal person01;
```

anlegen, wären die Eigenschaften `pers_nr` und `gehalt` mit einem undefinierten Wert belegt. Nur `name` hätte aufgrund der Vorzüge von `std::string` automatisch einen leeren String.

7.4.1 Klassenelemente direkt initialisieren (C++11)

Die wohl einfachste Möglichkeit, die Eigenschaften einer Klasse mit Werten zu initialisieren, bietet uns gleich der neue C++11-Standard an, indem Sie dies gleich bei der Klassendefinition wie folgt reinschreiben:

```
00 // kapitel07/7_4_1/personal.h
...
01 class Personal {
02     std::string name{""};      // Name
03     unsigned int pers_nr{0};   // Personalnummer
04     unsigned int gehalt{1800}; // monatliches Gehalt
05 };
...
```

Hier werden `name`, `pers_nr` und `gehalt` gleich mit Standardwerten initialisiert. Den leeren String bei `name` könnten Sie sich hier sparen, weil das die Klasse `std::string` für uns schon übernimmt. Die Personalnummer in Zeile 03 initialisieren wir hier einfach mit 0, und als Grundgehalt erhält jeder neue Angestellte gleich 1800 als Einstiegsgehalt (Zeile 04).

Wie erwähnt, funktioniert dies nur mit neueren C++11-Compilern. Recht Ähnliches können Sie allerdings auch mit dem Element-Initialisierer machen (siehe Abschnitt »Konstruktoren effektiver initialisieren mit Element-Initialisierer« in Abschnitt 7.4.2).

7.4.2 Konstruktoren

Wenn Sie ein Objekt erzeugen (hier bspw. eine Instanz von `Personal`), wird der Konstruktor (kurz auch als *ctor* bezeichnet) aufgerufen. Haben Sie keinen eigenen Konstruktor geschrieben, wird automatisch eine Standardversion – der **Standardkonstruktor** – zur Verfügung gestellt und verwendet. Das bedeutet, im Fall der Klasse `Personal` würde somit mit folgendem Aufruf

```
00 // kapitel07/7_4_2a/main.cpp
...
01 Personal person01;
02 Personal person02{}; // Vereinheitlichte C++11-Initialisierung
```

der Standardkonstruktor aufgerufen, da wir ja bisher noch keinen eigenen Konstruktor geschrieben haben. Allerdings muss hier auch erwähnt werden, dass die vom Compiler zur Verfügung gestellten Standardkonstruktoren nicht garantieren, dass alle Eigenschaften (Daten) automatisch mit 0 initialisiert werden. Sauberer C++-Stil ist es daher, sich **nicht** auf den Compiler zu verlassen, sondern selbst einen Konstruk-

tor zu schreiben. Der Standardkonstruktor wird immer aufgerufen, wenn Sie beim Anlegen eines Objekts keine weiteren Parameter mit angegeben haben, wie dies eben in den Zeilen 01 und 02 gezeigt wurde.

Hauptnutzen von Konstruktoren

Konstruktoren sind dazu da, beim Anlegen einer Instanz einer Klasse die Initialisierungen von Eigenschaften (Daten) durchzuführen bzw. den Eigenschaften einen Standardwert zuzuweisen.

Ein Konstruktor ist letztlich auch nur eine Methode – allerdings eine besondere. Neben dem Klassentyp `class` steht der Konstruktor auch für `struct` und `union` zur Verfügung. Im Gegensatz zu den gewöhnlichen Methoden (siehe Abschnitt 7.7, »Methoden – die Funktionen der Klasse«) unterscheiden sich Konstruktoren von diesen deutlich in den folgenden zwei Punkten:

- ▶ Der Name des Konstruktors ist derselbe wie der Name der Klasse.
- ▶ Der Konstruktor hat keinen Rückgabewert (auch nicht `void`).

Um einen Konstruktor selbst zu schreiben, müssen Sie diesen zunächst in der Klassendefinition der Headerdatei deklarieren:

```
00 // kapitel07/7_4_2a/personal.h
...
01 class Personal {
02     std::string name;        // Name
03     unsigned int pers_nr;    // Personalnummer
04     unsigned int gehalt;     // monatliches Gehalt
05 public:
06     Personal();
07 };
...
```

In Zeile 05 sehen Sie hier mit `public` ein sehr wichtiges Schlüsselwort, ohne das der Standardkonstruktor nicht direkt von außen verwendet werden könnte. Standardmäßig sind nämlich alle Mitglieder einer Klasse von außen nicht erreichbar – sprich `private`. Alles, was hinter Zeile 05 steht, kann jetzt auch von außen erreicht werden. Mehr darüber erfahren Sie gesondert in Abschnitt 7.6, »Exkurs: Zugriffskontrolle auf die Klassenmitglieder«. In Zeile 06 finden Sie hier die Deklaration des Standardkonstruktors.

Die Definition des Standardkonstruktors wird gewöhnlich aus Übersichtlichkeitsgründen außerhalb der Klassendefinition in einer extra Quelldatei mit der Endung `*.cpp` geschrieben:

```

00 // kapitel07/7_4_2a/personal.cpp
01 #include "personal.h"

02 Personal::Personal() {
03     name     = "";
04     pers_nr  = 0;
05     gehalt   = 1800;
06 }

```

Für den Zugriff auf Mitglieder einer Klasse von außen wird der Scope-Operator `::` (auch Bereichsoperator) benötigt. Die Verwendung mit `Klassenname::Mitglied` ist relativ einfach, wie Sie dies in Zeile **02** sehen können. In der Definition des Konstruktors (Zeile **02** bis **06**) werden die einzelnen Eigenschaften der Klasse mit gültigen Werten initialisiert.

Standardkonstruktor vs. direktes Initialisieren von C++11

Wenn Sie einen eigenen Standardkonstruktor schreiben und auch die direkte Initialisierung von Abschnitt 7.4.1 verwendet haben, erhält der selbst geschriebene Konstruktor den Vorzug, wenn es um die Initialisierung der Eigenschaften mit Werten geht. Ansonsten, wenn kein Standardkonstruktor existiert, werden die Werte der direkten Initialisierung verwendet. In der Praxis sollten Sie sich aber nur für eine Möglichkeit entscheiden. Wenn Ihr Compiler den C++11-Standard kann, dann können Sie gegebenenfalls auf den Standardkonstruktor verzichten und die einzelnen Eigenschaften gleich initialisieren, wie dies in Abschnitt 7.4.1 gezeigt wurde.

Hier sollte noch erwähnt werden, dass Sie die Definition des Konstruktors auch direkt in der Klassendefinition hätten schreiben können. Hierbei können Sie auf den Scope-Operator verzichten:

```

// personal.h
class Personal {
    std::string name;      // Name
    unsigned int pers_nr;  // Personalnummer
    unsigned int gehalt;    // monatliches Gehalt
public:
    Personal() {
        name     = "";
        pers_nr  = 0;
        gehalt   = 1900;
    }
};

```

Allerdings wird eine solche Definition eines Konstruktors innerhalb einer Klassendefinition eher selten verwendet und soll hier nur zu Demonstrationszwecken gezeigt werden.

Ein selbst geschriebener Konstruktor

Sobald Sie mindestens einen Konstruktor selbst geschrieben haben, stellt der Compiler keinen Standardkonstruktor mehr zur Verfügung. Allerdings wird in der Praxis ja ohnehin empfohlen, einen eigenen Konstruktor zu schreiben.

Konstrukturen mit Parameter

Neben dem Standardkonstruktor ist es durchaus gängige Praxis, weitere Konstrukturen für alle möglichen Szenarios mit Parametern zu schreiben, um möglichst viele Versionen beim Erzeugen eines Objekts abzudecken. Damit sollte es auch möglich sein, Objekte gleich beim Instanziiieren mit bestimmten Werten für die Eigenschaften zu initialisieren, bspw.:

```
00 // kapitel07/7_4_2b/main.cpp
...
01 Personal person01("Max Mustermann",100480, 2200);
02 Personal person02{"Maxi Musterfrau",100481, 2150}; //C++11
03 Personal person03("Hans Muster",100482);
04 Personal person04{"Marie Muster"}; // C++11
```

In den Zeilen **01** und **02** werden die Objekte gleich bei der Erzeugung mit den entsprechenden Werten für `name`, `pers_nr` und `gehalt` initialisiert. In Zeile **03** wird ein Objekt angelegt, wo nur die Eigenschaften `name` und `pers_nr` initialisiert werden und in der letzten Zeile wird nur die Eigenschaft `name` direkt belegt.

Solche und gegebenenfalls viele weitere Szenarios, neue Objekte zu erzeugen, können Sie dank Konstruktoren mit Parameter realisieren. Hierzu nun die Headerdatei *personal.h*, erweitert um weitere Konstruktoren mit Parametern, um eben beschriebene Objekte instanziiieren zu können:

```
00 // kapitel07/7_4_2b/personal.h
...
01 static const unsigned int GG = 1900;

02 class Personal {
03     std::string name;        // Name
04     unsigned int pers_nr;    // Personalnummer
05     unsigned int gehalt;     // monatliches Gehalt
```

```

06 public:
07     Personal();
08     Personal(std::string n,unsigned int pn,unsigned int g);
09     Personal(std::string n, unsigned int pn);
10     Personal(std::string n);
11 };
...

```

In der Klassendefinition können Sie sehr schön in den Zeilen **08** bis **10** die Deklarationen der neuen Konstruktoren mit Parametern sehen. Natürlich müssen auch diese Konstruktoren im `public`-Bereich deklariert sein. Die Definition der einzelnen Konstruktoren mit Parametern erfolgt jetzt, wie auch schon der Standardkonstruktor, in der anderen Quelldatei *personal.cpp*:

```

00 // kapitel07/7_4_2b/personal.cpp
01 #include "personal.h"
...
02 Personal::Personal( std::string n,
                      unsigned int pn, unsigned int g ) {
03     name = n;
04     pers_nr = pn;
05     gehalt = g;
06 }

07 Personal::Personal(std::string n, unsigned int pn) {
08     name = n;
09     pers_nr = pn;
10     gehalt = GG;
11 }

12 Personal::Personal(std::string n) {
13     name = n;
14     pers_nr = 0;
15     gehalt = GG;
16 }

```

Ich denke mir, dass die Definitionen der einzelnen Konstruktoren mit Parametern hier keinerlei Erklärung mehr bedürfen und es recht offensichtlich ist, was hier gemacht wird. Natürlich können Sie den Code noch um weitere Konstruktoren mit Parametern erweitern, sofern dies sinnvoll und logisch erscheint.

Objekt-Array und Konstruktoren

An dieser Stelle ist noch ein Hinweis nötig, auch wenn die Instanziierung einer Klasse noch nicht Thema ist. Und zwar darf die Klasse, wenn Sie ein Objekt-Array erstellen wollen (bspw. `Personal p[10]`), entweder gar keinen Konstruktor haben, oder es muss der Standardkonstruktor vorhanden sein. Würde diese Klasse nur Konstruktoren mit Parametern enthalten, könnte bspw. kein Objekt-Array wie folgt erzeugt werden:

```
Personal *persPtr = new Personal[100];
```

Konstruktoren effektiver initialisieren mit Element-Initialisierer

Um die Eigenschaften einer Klasse bei Konstruktoren mit Parametern zu initialisieren, gibt es mit dem *Element-Initialisierer* noch einen wesentlich effektiveren Weg als im Anweisungsblock des Konstruktors. Wenn Ihre Klasse dann noch Objekte als Eigenschaften (wie bspw. `std::string`, `std::vector` usw. oder andere eigene Klassen) enthält, dann sollten Sie auf jeden Fall die Initialisierungsliste (= Element-Initialisierer) verwenden. Aber dazu gleich mehr.

Bei der Deklaration der Konstruktoren mit Parametern in der Klassendefinition der Headerdatei von *kapitel07/7_4_2b/personal.h* müssen Sie nichts ändern. Die Initialisierungsliste (bzw. den Element-Initialisierer) müssen Sie erst bei der Definition des bzw. der Konstruktoren verwenden. Hier also nochmals die Definitionen der Konstruktoren mit Parametern – jetzt aber mit dem Element-Initialisierer:

```
00 // kapitel07/7_4_2c/personal.cpp
01 #include "personal.h"

02 Personal::Personal() : name(""), pers_nr(0), gehalt(GG) { }

03 Personal::Personal( std::string n, unsigned int pn,
                      unsigned int g) :
    name(n), pers_nr(pn), gehalt(g) { }

04 Personal::Personal(std::string n, unsigned int pn) :
    name(n), pers_nr(pn), gehalt(GG) { }

05 Personal::Personal(std::string n) :
    name(n), pers_nr(0), gehalt(GG) { }
```

Wie Sie an den Konstruktoren (inklusive Standardkonstruktor) sehr schön sehen können, wird die Initialisierungsliste durch einen Doppelpunkt nach der Parameterklammer des Konstruktors eingeleitet. Nach diesem Doppelpunkt werden die einzel-

nen Eigenschaften aufgelistet, welche initialisiert werden sollen. Den Initialwert für die Eigenschaften geben Sie gleich in den Klammern der Eigenschaft mit an.

Die Vorgehensweise, mit der Initialisierungsliste einzelne Eigenschaften mit Werten zu initialisieren, bringt bei einfachen Datentypen keinen echten Vorteil in der Effizienz. Aber wenn Ihre Klasse selbst auch Objekte enthält (wie in unserem Beispiel mit `std::string`), dann sollten Sie immer einen Element-Initialisierer verwenden. Bezogen auf unser Beispiel in Zeile 03 sollte der Konstruktor **mindestens** wie folgt definiert werden:

```
Personal::Personal( std::string n, unsigned int pn,
                    unsigned int g) : name(n) {
    pers_nr = pn;
    gehalt  = g;
}
```

Ohne den Element-Initialisierer von `name(n)` würde das `std::string`-Objekt von `Personal` zunächst einmal mit einem leeren String belegt, was ja einer der Vorzüge ist, wenn Sie fertige Standardklassen verwenden. Wenn Sie allerdings jetzt ein Objekt der Klasse `Personal` anlegen und der Konstruktor seine Arbeit ausführt, wird dem `std::string`-Objekt der Klasse `Personal` erst der eigentliche Wert übergeben. Ohne die Initialisierungsliste würde dem `std::string`-Objekt der Klasse `Personal` somit zweimal ein Wert übergeben – einmal ein leerer String und dann der eigentliche Wert.

Auch wenn sich die Effizienz einer Initialisierungsliste bei Konstruktoren erst mit (Teil-)Objekten einer Klasse bezahlt macht, spricht eigentlich überhaupt nichts dagegen auch gleich die einfachen Daten eines Konstruktors auf diese Weise zu initialisieren, wie dies bei den Definitionen der Konstruktoren in Listing *kapitel07/7_4_2c/personal.cpp* gemacht wurde.

Konstante Objekte als Eigenschaften

Neben dem Vorteil, ein besseres Laufzeitverhalten zu erzielen, um ein Teilobjekt einer Klasse gleich mit einer Initialisierungsliste zu initialisieren, ist es nur mit dem Element-Initialisierer möglich, auch konstante Eigenschaften in einer Klasse verwenden und mit einem Konstruktor initialisieren zu können. Mehr dazu erfahren Sie in Abschnitt 7.8.1, »Konstante Elemente in einer Klasse«.

Konstruktoren delegieren (C++11)

Eine Menge an Tipparbeit können Sie sich mit einem neuen C++11-Feature sparen, der *Delegation von Konstruktoren*. Im Grunde müssen Sie so nur noch einen zentralen Konstruktor definieren, welcher von allen anderen Konstruktoren aufgerufen

wird. In der folgenden Klassendefinition finden Sie eine solche Delegation von Konstruktoren:

```
00 // kapitel07/7_4_2d/personal.h
...
01 static const unsigned int GG = 1900;

02 class Personal {
03     std::string name;        // Name
04     unsigned int pers_nr;    // Personalnummer
05     unsigned int gehalt;     // monatliches Gehalt
06 public:
07     Personal():Personal("", 0, GG){};
08     Personal(std::string n):Personal(n, 0, GG){};
09     Personal(std::string n, unsigned int pn):Personal(n,pn,GG){};
10     Personal(std::string n, unsigned int pn, unsigned int g);
11 };
...
```

In den Zeilen **07**, **08** und **09** können Sie sehr schön sehen, wie alle drei Konstruktoren hinter dem Doppelpunkt den (Haupt-)Konstruktor aus Zeile **10** aufrufen. Der Vorgang lässt sich einfach erklären, bspw. wenn Sie Objekte wie folgt erzeugen:

```
00 // kapitel07/7_3_2d/main.cpp
...
01 Personal personal01;
02 Personal personal02("Max Mustermann");
03 Personal personal03("Maxi Musterfrau", 100483);
```

Für alle drei Objekte wird der Konstruktor aus Zeile **10** von *personal.h* verwendet, um die Eigenschaften zu initialisieren. Wenn Sie bspw. ein Objekt wie in Zeile **01** von *main.cpp* anlegen, wird der Konstruktor in Zeile **07** von *personal.h* aufgerufen. Dieser delegiert die Initialisierung der Eigenschaften an den Konstruktor aus Zeile **10**, dessen Definition sich in *personal.cpp* befindet. Wenn die Eigenschaften vom Objekt initialisiert wurden und der Anweisungsblock ausgeführt wurde, fährt der aufgerufene Konstruktor ebenfalls noch mit seinem Anweisungsblock (!) fort (hier der leere Anweisungsblock in Zeile **07**). Sie können gerne zur Demonstration etwas in den Anweisungsblöcken der einzelnen Konstruktoren schreiben, um nachvollziehen zu können, wann welcher aufgerufen wird.

Wie eben beschrieben, läuft dies auch mit den Instanzen der Zeilen **02** und **03** von *main.cpp* ab, nur dass hier eben entsprechende Konstruktoren der Zeilen **08** und **09** von *personal.h* aufgerufen und delegiert werden.

Jetzt fehlt uns nur noch der Code von *personal.cpp*, den wir dank dieser Konstruktor-Delegation jetzt auf einen einzigen Konstruktor reduzieren können:

```
00 // kapitel07/7_4_2d/personal.cpp
01 #include "personal.h"

02 Personal::Personal( std::string n, unsigned int pn,
    unsigned int g) : name(n), pers_nr(pn), gehalt(g) { }
```

Der Vorteil von solchen Konstruktor-Delegationen liegt auf der Hand. Neben der erheblichen Codereduzierung ist es nun auch leichter, wenn Sie dem Objekt neue Eigenschaften hinzufügen wollen. Sie müssen nur an wenigen Stellen im Code nachbessern. Und meiner persönlichen Meinung nach lässt sich diese Konstruktor-Delegation auch viel einfacher lesen und verstehen.

Position der Delegation

Es sollte noch erwähnt werden, dass diese Konstruktor-Delegation nur innerhalb der Initialisierungsliste hinter dem Doppelpunkt funktioniert. Wenn Sie eine »Delegation« innerhalb des Anweisungsblocks schreiben, was ja in C++03 gehen würde, dann würde lediglich ein temporäres Objekt erzeugt, was Sie auf keinen Fall wollen.

(Standard-)Kopierkonstruktor

Ein weiterer Konstruktor, der Ihnen ohne Code neben dem Standardkonstruktor zur Verfügung steht, ist der Kopierkonstruktor. Dank des Kopierkonstruktors können Sie zwei Objekte von derselben Klasse mit `a1=a2` oder `a1(a2)` mitsamt den Eigenschaften kopieren. In unserem aktuellen Quellcode von *personal.cpp* und *personal.h* könnten Sie somit folgendermaßen neue Objekte erstellen, ohne extra Code schreiben zu müssen:

```
00 // kapitel07/7_4_2e/main.cpp
...
01 Personal person01("Max Mustermann", 100480, 2200);
02 Personal person02(person01);    // Kopierkonstruktor
03 Personal person03 = person01;  // Kopierkonstruktor
```

In Zeile **02** wird eine exakte Kopie von `person01` in `person02` erstellt – mitsamt allen Eigenschaften. Dasselbe wird auch in Zeile **03** gemacht. Allerdings ist dies hier ein Sonderfall und optional, und es wird hier tatsächlich der Kopierkonstruktor und nicht der Zuweisungsoperator aufgerufen, denn es handelt sich hier um eine Initialisierung.

Weitere Aufrufe des Kopierkonstruktors

Hier zwar nicht abgedruckt, aber ebenfalls aufgerufen wird ein Kopierkonstruktor, wenn ein Objekt als Kopie (*Call by value*, siehe Abschnitt 5.5.1, »Funktionsparameter als Kopie (Call by Value)«) an eine Funktion übergeben oder als Rückgabewert einer Funktion (siehe Abschnitt 5.6, »Rückgabewert aus einer Funktion«) zurückgegeben wird.

Wenn allerdings spezielle Eigenschaften, wie bspw. rohe Zeiger, in einer Klasse enthalten sind, dann führt der vom Compiler zur Verfügung gestellte Kopierkonstruktor nicht mehr zum gewünschten Verhalten. Es werden zwar nach wie vor alle Eigenschaften kopiert, aber zwei Objekte, welche eine gemeinsame Eigenschaft haben – nämlich einen Zeiger, der auf dieselbe Adresse verweist –, führen selten zu einem guten Ende. In solch einem Fall müssen Sie den Kopierkonstruktor selbst implementieren. Und wenn Sie schon den Kopierkonstruktor selbst implementieren, dann sollten Sie auch gleich den Zuweisungsoperator (überladen und) implementieren (Stichwort *The Big Three*).

The Big Three (die großen Drei)

The Big Three ist eine Regel (bzw. Richtlinie), welche besagt: Wenn Sie mindestens einen der Konstruktoren selbst implementieren, also **Kopierkonstruktor**, **Zuweisungsoperator** (siehe Abschnitt 8.2, »Zuweisungsoperator – operator=«) oder **Destruktor** (siehe Abschnitt 7.5, »Objekte zerstören – Destrukturen«), dann sollten Sie auch die anderen beiden hinzufügen und sich nicht mehr mit der vom Compiler zur Verfügung gestellten Version zufriedengeben. Denn, wenn Sie einen der Konstruktoren selbst implementieren müssen, bedeutet dies ja auch, dass die Standardversionen nicht mehr ausreichend sind.

Wie bereits erwähnt, ist in unserem Fall eine Implementierung des Kopierkonstruktors **nicht** nötig, weil die Eigenschaften unserer Klasse `Personal` noch problemlos mit der Standardversion des Compilers kopiert werden können. Trotzdem soll hier kurz der Kopierkonstruktor demonstriert werden. Ein eigener Kopierkonstruktor hat immer die folgende Syntax:

```
Klassenname ( const Klassenname & );
```

Deklariieren müssen Sie den Kopierkonstruktor ebenfalls wieder im `public`-Bereich (es sei denn, Sie definieren alles gleich `inline` bei der Klassendefinition – logischerweise). Im Fall von der Klasse `Personal` sieht die Deklaration im `public`-Bereich von `personal.h` wie folgt aus:

```
00 // kapitel07/7_4_2e/personal.h
...

```

```

01 class Personal {
02     std::string name;      // Name
03     unsigned int pers_nr;  // Personalnummer
04     unsigned int gehalt;   // monatliches Gehalt
05 public:
...
06     Personal( const Personal & p );
07 };
...

```

Bei der Definition müssen Sie jetzt Eigenschaft für Eigenschaft kopieren, damit das neue Objekt die privaten Daten vom Original bekommt. Wie bereits erwähnt, wäre dies in unserem Fall nicht wirklich nötig gewesen, weil es der Standardkopierkonstruktor hier genauso gut kann und weil hier keine komplexeren Eigenschaften, wie bspw. rohe Zeiger, vorhanden sind. Hierzu nun die Definition des Kopierkonstruktors in *personal.cpp*:

```

00 // kapitel07/7_4_2e/personal.cpp
...
01 Personal::Personal( const Personal & p ) {
02     name = p.name + " - Kopie";
03     pers_nr = p.pers_nr;
04     gehalt = p.gehalt;
05 }

```

In den Zeilen **02** bis **04** wird Eigenschaft für Eigenschaft vom Original dem zu kopierenden Objekt zugewiesen. Zur Unterscheidung wurde bei *name* zusätzlich noch der Zusatz " - Kopie" hinzugefügt.

In dem Fall bietet sich auch wieder die effektivere Initialisierungsliste an, da wir hier ja wieder ein Teilobjekt mit *std::string* als Eigenschaft haben:

```

00 // kapitel07/7_4_2e/personal.cpp
...
01 Personal::Personal( const Personal & p ) :
    name(p.name+"-Kopie"),pers_nr(p.pers_nr),gehalt(p.gehalt) {}

```

Kopierkonstruktor in der Praxis

Sie werden eine sinnvollere Implementierung des Kopierkonstruktors in Abschnitt 7.8.2, »Roher Zeiger als Element in einer Klasse«, näher kennenlernen. Für die Implementierung des Zuweisungsoperators sind Kenntnisse in der Operatorüberladung nötig, was Thema von Kapitel 8, »Operatoren überladen«, speziell von Abschnitt 8.2, »Zuweisungsoperator – operator=«, sein wird.

Verschiebekonstruktor (C++11)

Neben dem Standardkopierkonstruktor und dem Standardzuweisungsoperator wurden mit dem neuen C++11-Standard jetzt zwei weitere Verschiebesemantiken (Move-Semantiken) hinzugefügt. Ziel und Zweck der neuen Move-Semantik ist es, Techniken in C++ anzubieten, bei denen Objekte nicht kopiert, sondern verschoben werden sollen. Es gibt einfach Objekte, die nicht kopiert werden sollten. Ein gedankenloses Kopieren von Objekten mit geöffneten Dateien oder Sperren als Eigenschaft könnte fatale Folgen haben, wenn beide Objekte bspw. in eine Datei schreiben oder beide Objekte versuchen, diese Datei zu sperren und die Sperrung wieder aufzuheben.

Außerdem und vor allem sind solche Verschiebeoperationen wesentlich effizienter als Kopieroperationen. Für solche Zwecke gibt es jetzt den neuen Verschiebekonstruktor (Move-Konstruktor) und den neuen Verschiebe-Zuweisungsoperator (Move-Zuweisungsoperator) in C++11.

Kopieren vs. Verschieben schnell verständlich gemacht

Am einfachsten lässt sich dieser Unterschied erläutern, indem Sie sich den Kopierkonstruktor und Kopier-Zuweisungsoperator als eine Art Kopieren & Einfügen in einer Textverarbeitung vorstellen. Die neuen Verschiebeoperationen mit dem Move-Konstruktor und Move-Zuweisungsoperator können Sie sich dagegen als den Ausschneiden-&-Einfügen-Mechanismus einer Textverarbeitung vorstellen. Etwas umfangreicher wird auf die neue Move-Semantik in Abschnitt 16.1, »Move-Semantik und Perfect Forwarding«, des Buches eingegangen.

Der Move-Konstruktor ist ein Verwandter des Kopierkonstruktors, bei dem das Argument mit den Zeichen `&&` anstatt `&` gekennzeichnet wird. Ein `&&`-Argument stellt ein temporäres Objekt dar, welches in Kürze gelöscht wird. Im Gegensatz zum Kopierkonstruktor darf der Parameter **nicht** mehr `const` sein, weil hierbei zur Erzeugung eines neuen Objekts die Daten aus dem Objekt »weggenommen« werden sollen, so dass das alte Objekt gar keine Daten mehr enthält oder zumindest »leer« ist.

Selbst implementieren oder die Standardversion verwenden

Auch hier gilt Gleiches wie schon beim Kopierkonstruktor: Der Compiler bietet Ihnen eine Standardversion an, welche für den normalen Hausgebrauch – wie in unserem Beispiel mit der Klasse `Personal` – noch ausreichend ist. Aber auch hier gilt, dass man bei kritischen Daten, wie bspw. dynamischen Elementen (flachen Kopien) einer Klasse, diese Spezialfunktionen selbst implementieren sollte.

Bezogen auf unser Beispiel der Klasse `Personal` könnten Sie, ohne weiteren Code zu schreiben, die Eigenschaften von Objekten in andere Objekte verschieben. Hierzu

wurde zum neuen C++11-Standard die neue Standardfunktion `std::move()` implementiert. Ein einfaches Beispiel hierzu:

```
00 // kapitel07/7_4_2f/main.cpp
...
01 Personal person01{"Max Mustermann", 100488, GG};
02 Personal person02 = std::move(person01);
03 Personal person03{std::move(person02)};
```

In Zeile **01** wird ein neues Objekt `person01` erstellt, dessen Daten in Zeile **02** mittels `std::move()` in das Objekt `person02` verschoben werden. Das bedeutet auch, dass der Inhalt von `person01` jetzt »leer« ist. Das Gleiche machen wir nochmal in Zeile **03**, wo wir den Inhalt `person02` jetzt nach `person03` verschieben, womit jetzt auch noch der Inhalt von `person02` leer ist. Wie bereits im Hinweisblock erwähnt, ist in unserem Beispiel noch keine eigene Implementierung des Move-Konstruktors nötig, aber an dieser Stelle soll trotzdem schon mal kurz gezeigt werden, wie Sie einen solchen Move-Konstruktor selbst implementieren können.

Ein solcher Move-Konstruktor hat folgende Syntax:

```
Klassenname ( Klassenname && );
```

Die Deklaration des Move-Konstruktors in unsere Klasse `Personal` sieht wie folgt aus:

```
00 // kapitel07/7_4_2f/personal.h
...
01 class Personal {
02     std::string name;        // Name
03     unsigned int pers_nr;    // Personalnummer
04     unsigned int gehalt;     // monatliches Gehalt
05 public:
...
06     Personal( const Personal & p ) = delete;
07     Personal( Personal && p );
...
08 };
```

Neben der Deklaration des Move-Konstruktors in Zeile **07** wurde hier noch das Kopieren mit dem Kopierkonstruktor verboten, indem wir hinter der Deklaration des Kopierkonstruktors in Zeile **06** `delete` zuweisen. Dies ist ebenfalls eine neues Feature von C++11, welches in Abschnitt 7.9.2, »delete«, noch etwas genauer erläutert wird. Das Kopieren von `Personal`-Objekten zu verbieten, ist in diesem Beispiel an sich recht sinnvoll, weil ein Kopieren von `Personal` in der Realität auch recht wenig Sinn ergibt, da man eine Person in der Regel auch eher zu einer anderen Kostenstelle »versetzt« (verschiebt).

Hierzu noch die Definition des Move-Konstruktors in *personal.cpp*:

```
00 // kapitel07/7_4_2f/personal.cpp
...
01 Personal::Personal( Personal && p ) :
    name(p.name), pers_nr(p.pers_nr), gehalt(p.gehalt) {
02     p.name="";
03     p.pers_nr = 0;
04     p.gehalt = 0;
05 }
```

Zunächst werden hier die Daten mithilfe einer Initialisierungsliste dem neuen Objekt übergeben (Zeile 01). Die ursprünglichen Daten »räumen« wir dann im Anweisungsblock in den Zeilen 02 bis 04 auf.

Zum Weiterlesen

Das Thema mit der neuen Verschiebesemantik wurde hier nur recht kurz und einfach gehalten. Die neue Verschiebesemantik wird in Abschnitt 16.1, »Move-Semantik und Perfect Forwarding«, noch umfassender beschrieben. Auch auf die Implementierung von Klassen für das Verschieben wird nochmals gesondert in Abschnitt 7.10, »Klassen für das Verschieben schreiben (C++11)«, eingegangen.

Die großen Drei ... und wie ist das dann in C++11?

In C++11 sind ja jetzt mit dem Move-Konstruktor und dem Move-Zuweisungsoperator zwei weitere Spezialfunktionen hinzugekommen, die vom Compiler generiert werden können. Hier haben Sie jetzt quasi fünf »Sonderfunktionen« (Kopierkonstruktor, Zuweisungsoperator, Destruktor, Move-Konstruktor und Move-Zuweisungsoperator), die der Compiler generieren kann. In reinem C++11 sind die Regeln etwas klarer und deutlicher vorgegeben. Definieren Sie in C++11 eine der fünf Sonderfunktionen, generiert der Compiler nicht mehr »automatisch« die anderen Sonderfunktionen. Wenn Sie bspw. den Move-Konstruktor selbst definieren, wird in C++11 kein Kopierkonstruktor oder Zuweisungsoperator mehr vom Compiler generiert. Aber aufgrund der nötigen Kompatibilität zur Vorgängerversion C++03 bleibt zumindest die Regel *der großen Drei* erst mal noch bestehen.

Implizite Konvertierung vermeiden – explicit

Sie haben bereits Konstruktoren mit einem Parameter kennengelernt (bspw. `Personal(std::string)`), wo per Initialisierung ein Objekt erzeugt werden kann. Solche Konstruktoren werden auch als Konvertierungskonstruktoren bezeichnet. In unse-

rem Beispiel soll jetzt noch ein Konvertierungskonstruktor von `Personal(const char*)` eingeführt werden:

```
00 // kapitel07/7_3_2g/personal.h
...
01 class Personal {
02     std::string name;        // Name
03     unsigned int pers_nr;    // Personalnummer
04     unsigned int gehalt;     // monatliches Gehalt
05 public:
...
06     Personal( const char * c_str );
...
};
```

Die Definition dieses Konvertierungskonstruktors kann wie folgt aussehen:

```
00 // kapitel07/7_3_2g/personal.cpp
...
01 Personal::Personal( const char * c_str ) :
    name(c_str), pers_nr(0), gehalt(GG) {}
```

Dank dieser Implementierung des Konvertierungskonstruktors ist jetzt Folgendes möglich:

```
00 // kapitel07/7_3_2g/main.cpp
...
01 Personal person01("Max Mustermann");
02 const char name01[] = "Hans Mustersohn";
03 Personal person02(name01);
04 Personal person03 = name01;
05 Personal person04 = "Maxi Musterfrau";
```

Die Konvertierungen in den Zeilen **01** und **03** von `const char*` funktionieren auch mit dem Konvertierungskonstruktor `Personal(std::string)` und benötigen keinen Konvertierungskonstruktor `Personal(const char*)`. Beide Konvertierungen werden **explizit** durchgeführt. Bei den beiden Konvertierungen in den Zeilen **04** und **05** hingegen wird bei der Initialisierung eine **implizite** Konvertierung durchgeführt, sie sind somit auf den Konvertierungskonstruktor `Personal(const char*)` angewiesen, um ein neues Objekt daraus erstellen zu können.

In der Praxis ist eine solche implizite Konvertierung allerdings nicht immer erwünscht. Eine implizite Konvertierung kann zu unerwarteten Ergebnissen führen, wenn sie durchgeführt wird, wo man sie gar nicht erwartet. Wollen Sie keine solchen impliziten Konvertierungen zulassen, können Sie dies mit dem Schlüsselwort `explicit` verhindern. Hierbei müssen Sie lediglich das Schlüsselwort `explicit` vor der

Deklaration des Konvertierungskonstruktors setzen, und die impliziten Konvertierungen der Zeilen **04** und **05** von *main.cpp* lassen sich nicht mehr verwenden. Der Compiler würde jetzt die Übersetzung verweigern.

Hierzu das Beispiel, in dem wir jetzt veranlassen, dass keine implizite Konvertierung von `Personal(const char*)` mehr möglich ist und nur noch explizit angefordert werden kann:

```
00 // kapitel07/7_3_2h/personal.h
...
01 class Personal {
02     std::string name;        // Name
03     unsigned int pers_nr;    // Personalnummer
04     unsigned int gehalt;     // monatliches Gehalt
05 public:
...
06     explicit Personal( const char * c_str );
...
};
```

Folgendes ist jetzt wegen des Schlüsselwortes `explicit` verboten:

```
01 Personal person03 = name01;           // Geht nicht mehr
02 Personal person04 = "Maxi Musterfrau"; // das auch nicht
```

Diese beiden Konvertierungen können Sie jetzt nur noch explizit wie folgt durchführen:

```
00 // kapitel07/7_3_2h/main.cpp
...
01 Personal person03(name01);
02 Personal person04("Maxi Musterfrau");
```

Fehler mit einheitlicher Initialisierungsliste vermeiden (C++11)

Mit einem einfachen Trick können Sie außerdem noch eine automatische Typumwandlung verhindern, indem Sie die neue vereinheitlichte Initialisierungsliste von C++11 verwenden. Das Thema (Stichwort *Narrowing*) wurde ja bereits in Abschnitt 2.9.2, »Automatische Typumwandlung beschränken (C++11)«, umfassend beschrieben. Betrachten Sie folgenden Codeausschnitt dazu:

```
01 Personal person05{"Franzi Mustertochter", 100.480, GG};
02 Personal person06("Frank Schwiegermutter", 100.480, GG);
```

Beim Anlegen eines Objekts wurde beim zweiten Parameter, der Personalnummer (`pers_nr`), versehentlich jeweils ein Punkt hingeschrieben, womit jetzt quasi ein `double`, statt wie erwartet eines `unsigned int` verwendet wird. Im alten C++03-Standard würde dieser Wert vom Compiler implizit in den passenden Typ umgewandelt

und die Werte dahinter abgeschnitten. Zeile **02** würde somit anstandslos den Wert 100 für die Personalnummer des neuen Objekts `person06` verwenden.

Mit der neuen vereinheitlichten Initialisierung, wie diese in Zeile **01** mit den geschweiften Klammern verwendet wurde, können Sie eine solche implizite Umwandlung verhindern. Hier sollte sich der Compiler mit einer Fehlermeldung zu einem *Narrowing* von `double` nach `unsigned int` bei Ihnen melden – ein sehr schönes weiteres Beispiel der neuen vereinheitlichten Initialisierung von C++11.

7.4.3 Initialisieren mit Methoden

Die Methoden werden zwar erst in Abschnitt 7.7, »Methoden – die Funktionen der Klasse«, behandelt, aber es sollte Ihnen nicht vorenthalten werden, dass Sie die Eigenschaften einer Klasse auch mit den Methoden initialisieren können. So können Sie bspw. eine Methode mit dem Bezeichner `init()` schreiben, um damit alle Eigenschaften der Klasse zu initialisieren. Für solche grundlegenden Initialisierungen sind allerdings in der Regel die Konstruktoren verantwortlich. Aber trotzdem wird häufig auch eine solche `init()`-Methode mit angeboten, womit man ja zusätzlich auch jederzeit nachträglich alle Eigenschaften eines Objekts ändern kann. In der Praxis verwendet man allerdings für Änderungen von einzelnen Eigenschaften sogenannte *Setter* (= Änderungsmethode) – welche allerdings wiederum zur Initialisierung einzelner Eigenschaften verwendet werden können. Wie allerdings erwähnt, soll das Thema Methoden erst in Abschnitt 7.7 umfassender behandelt werden.

Hier soll nun schlicht und einfach eine Methode `initPersonal()` geschrieben werden, mit der Sie alle Eigenschaften der Klasse initialisieren (und nachträglich ändern) können. Mit dieser Methode würden Sie quasi (in diesem Beispiel der Fall) ganz ohne einen Konstruktor auskommen.

Zunächst die Deklaration der Methode in *personal.h*:

```
00 // kapitel07/7_4_3/personal.h
...
01 static const unsigned int GG = 1900;

02 class Personal {
03     std::string name;    // Name
04     unsigned int pers_nr; // Personalnummer
05     unsigned int gehalt;  // monatliches Gehalt
06 public:
07     void initPersonal(std::string n="", unsigned int pn=0,
                        unsigned int g=GG);
08 };
...
```

Und jetzt noch die Definition in *personal.cpp*, welche, wie der Konstruktor auch, den Bereichsoperator benötigt:

```
00 // kapitel07/7_4_3/personal.cpp
01 #include "personal.h"

02 void Personal::initPersonal(
    std::string n, unsigned int pn, unsigned int g) {
03     name = n;
04     pers_nr = pn;
05     gehalt = g;
06 }
```

Die Verwendung dieser Methode, um ein Objekt mit Werten zu initialisieren, nachdem dieses erzeugt wurde, ist jetzt denkbar einfach, wie der folgende Codeausschnitt von *main.cpp* zeigen soll:

```
00 // kapitel07/7_4_3/main.cpp
...
01 Personal person01, person02;
02 person01.initPersonal("Max Mustermann", 100489, 2500);
03 person02.initPersonal();
```

Wie bereits erwähnt, können Sie diese Methode allerdings erst dann verwenden, wenn das Objekt bereits erzeugt wurde, wie dies in Zeile **01** der Fall ist. Beide Objekte werden hier mit dem vom Compiler zur Verfügung gestellten Standardkonstruktor instanziiert. In den Zeilen **02** und **03** bekommen die Objekte *person01* und *person02* dann ihre Werte. Dass der leere Aufruf in Zeile **03** funktioniert, ist den Standardparametern zu verdanken (verhält sich wie in Abschnitt 5.5.6, »Default-Parameter (Standardparameter)«, beschrieben), welche wir in der Deklaration angegeben haben.

Konstrukturen vs. Methoden

Wie allerdings zu Beginn des Abschnitts (7.4) erwähnt, sollten für das Anlegen eines neuen Objekts immer zuerst die Konstrukturen im Fokus des Programmierers liegen, um die Eigenschaften mit gültigen Werten zu initialisieren. Dies ist auch der Grund, warum in diesem Abschnitt die Möglichkeit, dies mit Methoden zu realisieren, erst zum Schluss gezeigt wurde.

Hierbei muss noch hinzugefügt werden, dass a) mit der Verwendung von regulären Methoden zur Initialisierung alle Felder zweimal gesetzt werden und – noch wichtiger – dass man b) dieses Vorgehen in Vererbungshierarchien (später) sehr vorsichtig einsetzen muss, da die eine oder andere Überraschung passieren kann.

7.5 Objekte zerstören – Destruktoren

Bei den »Big Three« bereits schon mal erwähnt, aber bisher noch nicht behandelt, lernen Sie jetzt den Destruktor kennen. Der Destruktor (häufig auch nur *dtor*) ist das Gegenstück zum Konstruktor. Wie der Konstruktor wird der Destruktor auch automatisch aufgerufen – mit dem Unterschied, dass der Destruktor nicht beim Erzeugen eines Objekts, sondern beim Zerstören eines Objekts aufgerufen wird. Der Destruktor wird für Aufräumarbeiten, wie bspw. das Freigeben von Speicherplatz, Aufheben von Sperren, Dateifreigaben und anderen Rückgaben von bestimmten Ressourcen, verwendet.

Auch wenn Sie in den bisherigen Beispielen noch keinen Destruktor zum Abbau eines Objekts geschrieben oder verwendet haben, so wurde trotzdem einer aufgerufen. Wie schon bei den Standardversionen der Konstruktoren wird auch hier eine Standardversion zu Verfügung gestellt, wenn Sie keinen eigenen implementiert haben. Allerdings gilt auch hier, wie schon beim Kopierkonstruktor (und Zuweisungsoperator) erwähnt, dass die Standardversion nicht für alle Fälle ausreichend ist. In unseren Beispielen war ein selbst implementierter Destruktor auch noch nicht nötig, weil wir keine kritischen Eigenschaften implementiert haben. Trotzdem soll hier ein einfacher Destruktor implementiert werden, damit Sie Bescheid wissen, wann diese aufgerufen werden.

7.5.1 Destruktor deklarieren

Wie auch der Konstruktor besteht der Destruktor nur aus dem Klassennamen und hat keinen Rückgabewert (auch kein `void`) und auch keinen Parameter. Zusätzlich wird dem Destruktor das Zeichen `~` (Komplementzeichen) vorangestellt, bspw.:

```
~Klassenname(); // Deklaration des Destruktors
```

In der Klasse `Personal` deklarieren wir den Destruktor selbstverständlich auch wieder im `public`-Bereich. Hierzu die Headerdatei `personal.h` mit dem Destruktor:

```
00 // kapitel07/7_5/personal.h
...
01 class Personal {
02     std::string name;      // Name
03     unsigned int pers_nr;  // Personalnummer
04     unsigned int gehalt;   // monatliches Gehalt
05 public:
06     Personal();
07     Personal(std::string n, unsigned int pn, unsigned int g);
08     Personal(std::string n, unsigned int pn);
09     Personal(std::string n);
```

```

10  ~Personal(); // Deklaration des Destruktors
11  };
...

```

7.5.2 Destruktor definieren

Bei der Destruktordefinition können Sie ähnlich wie schon bei der Definition von Konstruktoren vorgehen. Nur müssen Sie hierbei immer beachten, das Komplementzeichen zu verwenden, weil Sie sonst einen Standardkonstruktor definieren würden. Die Syntax der Definition lautet somit:

```

Klassenname::~Klassenname() {
    // Anweisungen
}

```

Bezogen auf die Definition in der Klasse `Personal` kann ein Destruktor in *personal.cpp* wie folgt aussehen:

```

00 // kapitel07/7_5/personal.cpp
...
01 Personal::~Personal() {
02     std::cout << name << " entfernt\n";
03 }

```

In der Praxis dient ein Destruktor natürlich nicht dazu, einfach nur einen Text auf dem Bildschirm auszugeben, wie hier im Beispiel zu sehen, sondern um diverse Aufräumarbeiten oder Ressourcenfreigaben zu machen. Die Textausgaben bei diesem Destruktor dienen im Augenblick nur dazu, im anschließenden Beispiel zu verstehen, wann der Destruktor aufgerufen wird.

7.5.3 Aufruf des Destruktors

Der Destruktor wird in der Regel nicht direkt aufgerufen. Der Aufruf erfolgt gewöhnlich implizit durch die Verwendung von `delete` oder beim Verlassen des Gültigkeitsbereichs. Globale Objekte hingegen werden erst beim Verlassen der `main()`-Funktion zerstört.

Das folgende Beispiel soll Ihnen jetzt in verschiedenen Fällen demonstrieren, wann der Destruktor aufgerufen wird:

```

00 // kapitel07/7_5/main.cpp
01 #include <iostream>
02 #include <memory> // C++11
03 #include "personal.h"
04 Personal person01("Globaler Mustermann");

```

```

05 void simple_function() {
06     Personal person02("simple_function Mustermann");
07 }

08 int main() {
09     Personal* person03 = new Personal("new Mustermann");
10     static Personal person04("static Mustermann");
11     simple_function();
12     delete person03;
13     Personal person05("lokal main Mustermann");
14     {
15         Personal person06("lokal lokal main Mustermann");
16         std::unique_ptr<Personal> personal07(
17             new Personal("unique_ptr Mustermann") );
18     }
19 }

```

Die Ausgabe des Programms sieht wie folgt aus:

```

simple_function Mustermann entfernt
new Mustermann entfernt
unique_ptr Mustermann entfernt
lokal lokal main Mustermann entfernt
lokal main Mustermann entfernt
static Mustermann entfernt
Globaler Mustermann entfernt

```

Zunächst demonstriert das Beispiel sehr schön, dass der Destruktor immer aufgerufen wird, wenn der Gültigkeitsbereich eines Objekts verlassen wird. Somit gilt auch hier für den Gültigkeitsbereich von Objekten dasselbe wie schon für die einfachen Datentypen (siehe Abschnitt 5.4, »Exkurs: Gültigkeitsbereiche«):

- ▶ Objekte, die lokal definiert wurden und nicht zur Speicherkategorie `static` gehören, werden am Ende des entsprechenden Anweisungsblocks zerstört (im Beispiel die Objekte in den Zeilen **06**, **13** und **15**). Gleiches gilt auch für den neu in C++11 eingeführten klugen Zeiger `std::unique_ptr` in Zeile **16**, welcher den Speicherbereich am Ende seines Gültigkeitsbereichs in Zeile **17** automatisch wieder freigibt.
- ▶ Dynamische Objekte, welche mit dem klassischen `new` reserviert werden, werden an Ort und Stelle gelöscht, wenn der Speicher explizit mit `delete` freigegeben wird (wie dies im Beispiel in den Zeilen **09** mit `new` und **12** mit `delete` demonstriert wird).
- ▶ Globale oder mit `static` erzeugte Objekte werden erst am Ende des Programms (beim Verlassen von `main()`) zerstört, wie es im Beispiel bei den Zeilen **04** und **10** der Fall ist.

Destruktor manuell aufrufen?

Es wurde bereits erwähnt, dass ein Destruktor in der Regel nicht manuell aufgerufen wird. Ein versuchter Aufruf von `person01.~Personal();` dürfte bei den meisten Rechnern zum Programmabsturz führen. Es kann aber auch passieren, dass das Programm eben weiterläuft. Der weitere Ablauf des Programms ist somit undefiniert, weil zweimal der Destruktor für das gleiche Objekt aufgerufen würde:

- einmal, weil Sie diesen explizit aufgerufen haben
- und einmal, wenn der Gültigkeitsbereich endet und das Objekt automatisch zerstört wird

Theoretisch gibt es hier die Möglichkeit, im Zeitraum, wenn Sie denn Konstruktor explizit aufgerufen haben und bevor der Gültigkeitsbereich des Objekts endet, ein `Placement-new` zu verwenden, um das undefinierte Verhalten zu umgehen. Sie zerstören quasi ein Objekt und generieren gleich wieder ein Neues daraus. Allerdings muss ich eingestehen, dass ich dies noch nie wirklich benötigt oder in der Praxis gesehen habe. Ich will Ihnen das unschöne Beispiel hierzu nicht vorenthalten:

```
01 {
02     Personal p01("Max Mustermann");
03     Personal *pPtr = &p01;
04     p01.~Personal();
05     new (pPtr) Personal("Maxi Musterfrau");
06 }
```

Der Code ist nicht schön und soll hier auch nicht Schule machen. In Zeile **04** wird der Destruktor explizit aufgerufen, und in Zeile **05** wird gleich mit einem `new-Replacement` ein neues Objekt am selben Platz im Speicher erzeugt. Als Mantra bleibt mir hier nur zu sagen, dass Sie den Destruktor wirklich nur dann manuell aufrufen sollten, wenn es nicht mehr anders geht und Sie genau wissen, was Sie da tun!

Sind Sie sich nicht sicher, wann Sie einen Destruktor benötigen und wann nicht, dann möchte ich Sie nochmals kurz auf den Hinweiskasten »The Big Three (die großen Drei)« im Abschnitt »(Standard-)Kopierkonstruktor« in Abschnitt 7.4.2 verweisen.

7.6 Exkurs: Zugriffskontrolle auf die Klassenmitglieder

Bevor Sie die Methoden kennenlernen, benötigen Sie noch Kenntnisse über die Zugriffskontrollen auf die Klassenmitglieder. Bisher haben Sie es als gegeben hingenommen, wenn in den Beispielen `public` verwendet wurde oder die Rede von `private` war. Erst mit den Schlüsselwörtern `public`, `private` und `protected` kann die Datenkapselung in C++ richtig gesteuert werden.

Achtung: Theorie

Das Thema ist relativ theoretisch, aber immens wichtig in der OOP – oder besser: Ohne die Zugriffsrechte lässt sich die Datenkapselung der OOP gar nicht realisieren. Daher wurde der Abschnitt etwas losgelöst als Exkurs geschrieben, womit Sie jederzeit hierher zurückkommen und nachlesen können.

Würden Sie in einer Klasse (`class`) keines dieser Schlüsselwörter verwenden, dann wären alle darin enthaltenen Eigenschaften und Methoden automatisch `private`. Das würde bedeuten, es wäre von außen überhaupt kein Zugriff möglich. Es liegt auf der Hand, dass eine Klasse, in der alles `private` ist und kein Zugriff von außen möglich ist, kaum einen Sinn macht. Um Zugriff auf Mitglieder einer Klasse von außen zu gewähren, deklariert man diese als `public`. Zusätzlich gibt es noch den Zugriffsmodus `protected`, dessen Sinn aber erst später bei der Vererbung in Abschnitt 9.3, »Zugriffsschutz anpassen«, erläutert wird.

In der klassischen objektorientierten Programmierung werden dabei meistens die Eigenschaften (Daten) von außen abgeschirmt und `private` gemacht. Der Zugriff auf die Daten sollte stets über Methoden (auch Konstruktoren) erfolgen, weshalb diese gewöhnlich mit `public` öffentlich gemacht werden. In der Praxis werden dann zum Lesen oder Verändern der Eigenschaften (falls nötig) Methoden wie `getVariableX` bzw. `setVariableX` zur Verfügung gestellt. Solche Methoden werden auch als *Getter* und *Setter* bezeichnet. Gerade beim Zugriff auf Eigenschaften mit den `set`-Methoden können die übergebenen Werte von außen auch noch auf Korrektheit hin überprüft werden. Mehr dazu in Abschnitt 7.7, »Methoden – die Funktionen der Klasse«.

Datenkapselung nicht aufweichen

An dieser Stelle muss noch erwähnt werden, dass der Schutz der Eigenschaften jederzeit umgangen werden kann. Der Schutz der Datenkapselung verhindert zwar Änderungen der Daten, bewahrt aber nicht vor mutwilliger Aufweichung des Konzepts der Datenkapselung. C++ liefert Ihnen nur die Mittel; für eine saubere Implementierung von abstrakten Typen sind also nach wie vor Sie als Programmierer verantwortlich.

Standardzugriffsrechte

Es wurde bereits erwähnt, dass `struct` (und `union`) ebenfalls ein Klassentyp ist. Der Unterschied zwischen `struct` und `class` liegt an den voreingestellten Zugriffsrechten der Klassentypen. Beim `struct`-Klassentyp ist das voreingestellte Zugriffsrecht gleich `public`. Das bedeutet, ohne weitere Angaben sind alle Mitglieder öffentlich zugänglich. Beim `class`-Klassentyp hingegen sind alle Mitglieder zunächst mit `private` eingestellt und von außen nicht erreichbar.


```

class KlasseC{
    // Hier ist standardmäßig alles private und von außen
    // nicht erreichbar. Klassen nur mit gesperrten Mitgliedern
    // sind eigentlich völlig nutzlos.
};

struct StructS {
    // Hier ist standardmäßig alles public und öffentlich
    // von außen erreichbar. Bei Klassen mit nur öffentlichen
    // Mitgliedern wird die Datenkapselung aufgeweicht.
};

```

Der Unterschied der beiden Klassentypen ist somit, dass Sie bei `class` alle Mitglieder, die von außen erreichbar sein sollen, explizit mit `public` freigeben müssen. Bei `struct` hingegen müssen Sie alle Mitglieder, welche von außen geschützt werden sollen, explizit mit `private` schützen. Folgende zwei Klassentypen sind somit in der Verwendung völlig identisch:

```

class KlasseC {
    int eigenschaft;           // Standardmäßig private
public:
    void set_eigenschaft(int);
};

struct StructS {
    void set_eigenschaft(int); // Standardmäßig public
private:
    int eigenschaft;
};

```

Zugriffsrechte setzen

Das Setzen der Zugriffsrechte erfolgt innerhalb der Klassentypen mit `private` oder `public`, gefolgt von einem Doppelpunkt. Alles, was dahinter steht, wird dann mit dem entsprechenden Zugriffsrecht versehen, bis entweder das Zugriffsrecht mit `public` oder `private` geändert wird oder das Ende der Klassendefinition erreicht wurde. Sie können also durchaus mehrmals die Zugriffsrechte in einer Klasse ändern, bspw.:

```

class KlasseC {
    // Für alle Mitglieder, welche hier stehen,
    // gilt noch das Standardzugriffsrecht private
public:
    // Auf die hier folgenden Mitglieder kann
    // ein Zugriff von außen erfolgen

```

```
private:
    // Auf die folgenden Mitglieder ist wiederum
    // kein Zugriff mehr von außen möglich
public:
    // Und hier sind wieder Mitglieder für die
    // Öffentlichkeit zugänglich
};
```

In der Praxis werden die Zugriffsrechte allerdings eher in einen `private`- und einen `public`-Teil aufgeteilt, was auch wesentlich übersichtlicher erscheint. In der Regel werden die Eigenschaften (= Daten) von außen abgeschirmt und in den `private`-Teil gesteckt und die Methoden (= Funktionen der Klasse) kommen in den öffentlichen `public`-Teil. Ein typisches Klassengerüst sieht demnach häufig wie folgt aus:

```
class KlasseC {
private:
    Typ Daten01;
    Typ Daten02;
    ...
public:
    RTyp Methode01(parameter);
    RTyp Methode02(parameter);
    ...
};
```

Das Schlüsselwort `private` wäre in diesem Klassengerüst an dieser Stelle nicht nötig gewesen, weil das hier ohnehin das Standardzugriffsrecht gewesen wäre.

Zugriff auf `public`-Mitglieder

Immer war hier die Rede vom Zugriff von außen. Wenn ein Zugriff von außen auf eine Klasse erfolgt, dann geschieht dies über ein Bindeglied, genauer ein Objekt. Sie können nicht, wie Sie dies von der prozeduralen Programmierung her kennen, einfach eine Funktion global aufrufen. Um also auf `public`-Mitglieder einer Klasse zugreifen zu können, wird zunächst ein Objekt benötigt (Abbildung 7.5).

Über das Objekt können Sie mit dem Punktoperator oder gegebenenfalls indirekt mit dem Pfeiloperator auf die öffentlichen Mitglieder der Klasse zugreifen. Genau genommen kennen Sie diesen Zugriff bereits, Sie haben ihn bereits unwissentlich auf `public`-Mitglieder bei den Strukturen in den Abschnitten »Zugriff auf die einzelnen Elemente einer Strukturvariablen« und »Roher Strukturzeiger« in Abschnitt 4.5 verwendet und kennengelernt. Nur haben Sie dort direkt auf die Daten zugegriffen, was in der klassischen OOP allerdings nicht so sein sollte. Hier greifen Sie auf die Methoden zu, welche dann wiederum auf den Daten operieren:

```
KlasseC objekt01;
objekt01.methode01(parameter);
KlasseC *objPtr = &objekt01;
objPtr->methode02(parameter);
```

Zugriff auf private-Mitglieder

Der Zugriff auf die `private`-Mitglieder einer Klasse ist von außen über ein Objekt nicht möglich. In der Regel sind es meistens Eigenschaften (= Daten), die `private` sind. Solche `private`-Mitglieder können nur innerhalb einer Klasse erreicht werden (Abbildung 7.5). Somit könnten `private`-Eigenschaften nur innerhalb einer Klasse geändert und direkt verwendet werden. Gleiches gilt gegebenenfalls auch, wenn Sie Methoden im `private`-Bereich geschrieben haben. Diese lassen sich nur noch innerhalb der Klasse aufrufen.

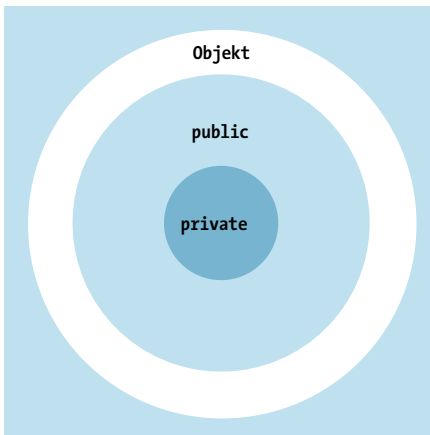


Abbildung 7.5 Schematische Darstellung der Datenkapselung eines Objekts.

Von außerhalb des Objekts sollte kein Zugriff auf die `public`- bzw. `private`-Mitglieder erfolgen. Auf die `public`-Mitglieder selbst kann nur über das Objekt zugegriffen werden. Die `private`-Mitglieder wiederum können nur innerhalb der Klasse verwendet werden (bspw. von `public`-Mitgliedern wie Methoden).

friend

An dieser Stelle soll noch kurz auf das Schlüsselwort `friend` hingewiesen werden, das zwar nicht direkt zur Zugriffskontrolle verwendet wird, aber welches die Zugriffskontrolle von Klassen und Funktionen erleichtern (oder auch aufweichen) kann. Verwenden Sie dieses Schlüsselwort, haben Sie vollen Zugriff auf alle Mitglieder einer Klasse. Das Schlüsselwort wird in Abschnitt 7.7.9, »Globale friend-Funktion«, und in Abschnitt 7.11, »Friend-Klassen«, behandelt.

7.7 Methoden – die Funktionen der Klasse

Nachdem Sie mit den Eigenschaften (= Daten) der Klasse vertraut sind, und die Grundlagen zu den Zugriffsrechten (`public`, `private`) kennen, soll jetzt endlich das letzte fehlende Bindeglied einer grundlegenden Klasse, die Methoden (auch Memberfunktionen oder Elementfunktionen genannt) der Klasse, behandelt werden. Alles, was Sie bisher zu den gewöhnlichen Funktionen der prozeduralen Programmierung kennengelernt haben, gilt auch für die Methoden, mit dem einzigen Unterschied, dass Methoden eben Teil einer Klasse sind und daher in der Regel auch nur über eine Instanz einer Klasse aufgerufen werden können. Ziel in einem rein in OOP programmierten Programm ist es, dass möglichst jede Funktion (außer `main()`) eine Methode einer Klasse sein sollte. Vereinzelt werden allerdings trotzdem sogenannte prozedurale Hilfsfunktionen verwendet.

7.7.1 Methoden deklarieren und definieren

Im Grunde haben Sie ja mit den Konstruktoren eine bestimmte Form von Methoden kennengelernt. Für die Deklaration und Definition von Methoden gilt daher in der Regel dasselbe wie für die Konstruktoren. Üblicherweise werden die Methoden in der Klassendefinition in einer Headerdatei deklariert und in einer separaten Quelldatei definiert. Um wieder zurück zu unserer Klasse `Personal` zu kommen, wollen wir diese Klasse jetzt um eine Methode zur Ausgabe aller Eigenschaften erweitern.

Methoden-Deklaration in Klassendefinition

Auch wenn die Methode in der Klasse noch nicht definiert, sondern nur deklariert wurde, spricht man bei `class Klasse{ };` von einer Klassendefinition und nicht von einer Klassendeklaration. Es ist eine Definition, weil hier ein neuer Datentyp definiert wird. Eine Klassendeklaration wäre demnach `class Klasse;` ohne die geschweiften Klammern.

Die **Deklaration** in der Headerdatei sieht demnach wie folgt aus:

```
00 // kapitel07/7_7_1/personal.h
...
01 class Personal {
02     std::string name;      // Name
03     unsigned int pers_nr;  // Personalnummer
04     unsigned int gehalt;   // monatliches Gehalt
05 public:
06     Personal();
07     Personal(std::string n, unsigned int pn, unsigned int g);
08     Personal(std::string n, unsigned int pn);
```

```

09  Personal(std::string n);
10  void print_data();
11  };
...

```

Hier finden Sie in Zeile **10** die Deklaration der Methode `print_data()` im public-Bereich vor, damit Sie anschließend auch darauf zugreifen können.

Die Definition einer Methode schreibt man üblicherweise in einer separaten Quelldatei, wo auch die Konstruktoren definiert wurden. Wie schon bei den Konstruktoren müssen Sie hierbei den Scope-Operator `::` und den Klassennamen verwenden. Die Syntax hierzu lautet:

```

RTyp Klassenname::methodenname( parameter ) {
    // Anweisungen für die Methode
}

```

Hiermit teilen Sie dem Compiler mit, dass `methodenname` eine Methode der Klasse `Klassenname` ist. Würden Sie `Klassenname::` hier nicht verwenden, so würden Sie lediglich eine globale prozedurale Funktion definieren.

Bezogen auf die Klasse `Personal` und der Methode `print_data()` sieht die Definition außerhalb der Klasse in *personal.cpp* wie folgt aus:

```

00 // kapitel07/7_7_1/personal.cpp
...
01 void Personal::print_data() {
02     std::cout << "Name    : " << name << std::endl;
03     std::cout << "Pers.Nr.: " << pers_nr << std::endl;
04     std::cout << "Gehalt  : " << gehalt << "\n\n";
05 }

```

Bei dieser Methode können Sie außerdem sehr gut erkennen, dass Sie innerhalb der Klasse direkt auf die Eigenschaften (`name`, `pers_nr` und `gehalt`) zugreifen können – ohne irgendwelche Scope-Operatoren verwenden zu müssen. Dasselbe gilt auch, wenn Sie innerhalb einer Methode eine andere Methode aufrufen. Auch hier können Sie ganz ohne Scope-Operator auf die Methode zugreifen. Alle Eigenschaften und Methoden innerhalb einer Klasse harmonisieren somit miteinander, wodurch, richtig programmiert, Methoden ohne Umwege auf Eigenschaften und andere Methoden zugreifen können.

Jetzt müssen Sie lediglich noch ein Objekt von `Personal` erzeugen und können damit auf die Methode `print_data()` mithilfe des Punktoperators zugreifen, um alle Eigenschaften des Objekts auszugeben. Hierzu ein Beispiel in der Praxis, das dies demonstrieren soll:

```

00 // kapitel07/7_7_1/main.cpp
...
01 Personal person01("Max Mustermann");
02 Personal person02("Maxi Musterfrau", 400500, 2300);

03 person01.print_data();
04 person02.print_data();

```

In den Zeilen **01** und **02** werden zwei Objekte erzeugt. In den Zeilen **03** und **04** verwenden wir mithilfe des Objekts und des Punktoperators die Methode `print_data()`, um den Inhalt der Eigenschaften der beiden Objekte auf dem Bildschirm auszugeben.

Die Ausgabe des Programms sieht wie folgt aus:

```

Name      : Max Mustermann
Pers.Nr.: 0
Gehalt    : 1900

```

```

Name      : Maxi Musterfrau
Pers.Nr.: 400500
Gehalt    : 2300

```

Implizit inline

Es ist auch möglich, dass Sie die Methode innerhalb der Klassendefinition definieren – wenn auch in der Praxis bei größeren Projekten eher unüblich. Eine in der Klassendefinition definierte Methode gilt implizit als `inline`-Methode (auch ohne das Schlüsselwort `inline`). Damit schlagen Sie dem Compiler vor, die Elementfunktion nicht über den Stack (Stapel) aufzurufen, sondern direkt im Code zu verwenden. Hier der Quellcode, mit dem die Methode `print_data()` in der Klassendefinition definiert wurde und somit implizit `inline` ist:

```

00 // kapitel07/7_7_1/inline/personal.h
...
01 class Personal {
02     std::string name;      // Name
03     unsigned int pers_nr;  // Personalnummer
04     unsigned int gehalt;   // monatliches Gehalt
05 public:
06     Personal();
07     Personal(std::string n, unsigned int pn, unsigned int g);
08     Personal(std::string n, unsigned int pn);
09     Personal(std::string n);
10     void print_data(){

```

```

11     std::cout << name << std::endl;
12     std::cout << pers_nr << std::endl;
13     std::cout << gehalt << std::endl;
14 }
15 };
...

```

Explizit inline

Methoden können Sie auch explizit als `inline` definieren, was durchaus die gängigere Methode darstellt. Hierbei gehen Sie genauso vor wie schon bei den `inline`-Funktionen und setzen bei der Definition vor die Klassenmethode das Schlüsselwort `inline`.

Auf unser Beispiel der Klasse `Personal` bezogen, müssten Sie hierzu in der Datei `personal.cpp` vor die entsprechenden Elementfunktionen das Schlüsselwort `inline` schreiben (natürlich setzt das voraus, dass Sie in der Headerdatei `personal.h` in der Klassendefinition noch keine Elementfunktionen definiert haben, wie eben im Abschnitt »Implizit inline« gesehen). Hier das Listing mit der Methode `print_data()` dazu (gekürzte Fassung):

```

00 // kapitel07/7_1/expl_inline/personal.cpp
...
01 inline void Personal::print_data(){
02     std::cout << name << std::endl;
03     std::cout << pers_nr << std::endl;
04     std::cout << gehalt << std::endl;
05 }

```

7.7.2 Zugriffsmethoden – Setter und Getter

Die einfachsten Zugriffsmethoden auf die Eigenschaften dürften die *Setter* und *Getter* sein. Beide sind allerdings keine Fachbegriffe, sondern eher üblicher Programmiererjargon.

Die *Getter* (engl. von *to get*: etwas holen) sind einfach **Abfragemethoden**, welche eine bestimmte Eigenschaft eines Objekts abfragen. Die Eigenschaft kann dabei direkt aus der Variablen des Objekts entnommen oder auch berechnet werden. In der Praxis werden solche Methoden häufig mit dem Bezeichner `get` eingeleitet – gefolgt vom Bezeichner der Eigenschaft (bspw. `get_gehalt()`).

Auf der anderen Seite verwendet man die *Setter* (engl. von *to set*: etwas einstellen) als **Änderungsmethoden**. Solche Änderungsmethoden sollten natürlich in Verbindung mit einer Überprüfung des übergebenen Wertes auf Gültigkeit erfolgen. Wenn ein ungültiger Wert übergeben wurde, könnten Sie mit einer Ausnahme reagieren (siehe

Kapitel 10, »Ausnahmebehandlung (Exceptions«). In der Praxis werden solche Methoden mit dem Bezeichner `set` eingeleitet – gefolgt vom Bezeichner der entsprechenden Eigenschaft (hier bspw. `set_gehalt(val)`).

Im folgenden Beispiel will ich Ihnen einige solcher Abfrage- und Änderungsmethoden vorstellen, welche alle auf die Eigenschaften der Klasse operieren. Zunächst die Deklarationen in der Headerdatei *personal.h*:

```
00 // kapitel07/7_7_2/personal.h
...
01 class Personal {
02     std::string name;      // Name
03     unsigned int pers_nr;  // Personalnummer
04     unsigned int gehalt;   // monatliches Gehalt
05 public:
06     Personal();
07     Personal(std::string n, unsigned int pn, unsigned int g);
08     Personal(std::string n, unsigned int pn);
09     Personal(std::string n);
10     void set_name(const std::string &n);
11     void set_pers_nr(unsigned int nr);
12     void set_gehalt(unsigned int g);
13     std::string get_name();
14     unsigned int get_pers_nr();
15     unsigned int get_gehalt();
16     void print_data();
17 };
...
```

Die Deklarationen der Zugriffsmethoden sind in den Zeilen 10 bis 15 definiert und bedürfen auch keiner weiteren Erläuterung mehr. Jetzt zu den Definitionen dieser Zugriffsmethoden in der Quelldatei *personal.cpp*:

```
00 // kapitel07/7_7_2/personal.cpp
...
01 void Personal::print_data() {
02     std::cout << "Name      : " << get_name() << std::endl;
03     std::cout << "Pers.Nr.: " << get_pers_nr()
04         << std::endl;
05     std::cout << "Gehalt   : " << get_gehalt() << "\n\n";
06 }

06 void Personal::set_name(const std::string &n) {
07     if( n == "" ) {
```



```

08     std::cout << "Leerer String nicht erlaubt\n";
09     // Weitere Maßnahmen treffen ...
10 }
11 // gegebenenfalls weitere Überprüfungen
12 else {
13     name = n;
14 }
15 }

16 void Personal::set_pers_nr(unsigned int nr) {
17     if( (nr < 400000) || (nr > 400999) ) {
18         std::cout << "400000 - 400999 als Nummer erlaubt\n";
19         // gegebenenfalls weitere Maßnahmen treffen
20     }
21     else {
22         pers_nr = nr;
23     }
24 }

25 void Personal::set_gehalt(unsigned int g) {
26     // Mindestgehalt prüfen
27     if( g < GG ) {
28         std::cout << "Achtung unter dem Mindestgehalt\n";
29         gehalt = GG; // Mindestgehalt setzen
30         std::cout << "Setzen auf Mindestgehalt\n";
31     }
32     // gegebenenfalls maximales Gehalt prüfen
33     gehalt = g;
34 }

35 std::string Personal::get_name() {
36     if(name == "") {
37         std::cout << "Noch kein Name vergeben\n";
38     }
39     return name;
40 }

41 unsigned int Personal::get_pers_nr() {
42     if(pers_nr == 0) {
43         std::cout << "Personalnummer noch nicht vergeben\n";
44         // gegebenenfalls weitere Maßnahmen vornehmen
45     }
46     else if( (pers_nr < 400000) || (pers_nr > 400999) ) {

```

```

47     std::cout << "Ungültige Personalnummer!!!\n";
48     // gegebenenfalls weitere Maßnahmen vornehmen
49 }
50 // Hier bspw. auch auf doppelte Personalnummern testen
51 return pers_nr;
52 }

53 unsigned int Personal::get_gehalt() {
54     // Mindestgehalt prüfen und gegebenenfalls setzen
55     if( gehalt < GG ) {
56         std::cout << "Achtung unter dem Mindestgehalt\n";
57     }
58     // Hier könnte eine Abfrage stehen, ob eine
59     // Brutto- oder Nettorückgabe erwünscht wird
60     return gehalt;
61 }

```

Die Zugriffsfunktionen stellen natürlich nur Beispiele dar und nicht, wie es wirklich sein muss. Da in diesem Beispiel keine stumpfsinnigen Getter verwendet wurden, welche gegebenenfalls auch leere Werte ausgeben, wurden diese Methoden auch gleich in der Methode `print_data()` in den Zeilen 02 bis 04 für die Ausgabe aller Eigenschaften implementiert.

Wenn Sie bspw. die Methode `get_name()` verwenden, welche in den Zeilen 35 bis 40 definiert wurde, wird in der Methode zunächst überprüft (Zeile 36), ob es sich hier nur um einen leeren String handelt (weil bspw. ein leeres Objekt via `Personal person;` instanziiert wurde). Ist dies der Fall, wird eine entsprechende Ausgabe gemacht, bevor der Inhalt von Eigenschaft `name` zurückgegeben wird. Ähnlich ist dies auch beim Gegenstück von `set_name()` in den Zeilen 06 bis 15. Hier würde sich zusätzlich noch eine Möglichkeit anbieten (ab Zeile 09), eine Eingabeaufforderung abzusetzen, um den Anwender um einen gültigen Namen zu bitten und diesen an die Eigenschaft `name` zu übergeben. Wenn kein leerer String an `set_name()` als Parameter übergeben wurde, wird die Eigenschaft `name` in Zeile 13 gesetzt.

Ähnlich wurde diese auch bei `get_pers_nr()` (Zeilen 41 bis 52) und `set_pers_nr()` (Zeilen 16 bis 24) realisiert. Auch hier stellen wir zunächst sicher, dass die Eigenschaft `pers_nr` des Objekts bereits einen gültigen Wert enthält. Ein ungültiger Wert wäre bspw. gleich 0 oder ein Wert kleiner als 400.000 oder größer als 400.999. In beiden Fällen wird eine entsprechende Meldung auf dem Bildschirm ausgegeben. Bei `set_pers_nr()` würde sich bspw. ab Zeile 19 noch anbieten, den Anwender nachträglich nach einer gültigen Personalnummer zu fragen und sie zu setzen.

Und das Gleiche haben wir auch noch mit den letzten beiden Zugriffsmethoden `set_gehalt()` (Zeilen 25 bis 34) und `get_gehalt()` (Zeilen 53 bis 61) gemacht. Auch hier prü-

fen wir, ob die Person entsprechend bezahlt wird und nicht unter dem Mindestgehalt (hier mit GG definiert) liegt. Im Fall von `set_gehalt()` setzen wir dies gegebenenfalls nachträglich auf GG. Natürlich ließen sich auch interessantere Methoden erstellen. Bei `get_gehalt()` könnten Sie bspw. anbieten, das Gehalt in Brutto oder Netto zurückzugeben – den entsprechenden Algorithmus müssten Sie hier implementieren.

Hierzu ein Beispiel, welches diese Setter und Getter im Einsatz demonstrieren soll:

```
00 // kapitel07/7_7_2/main.cpp
...
01 Personal person01;
02 person01.print_data();
03 person01.set_name("Maxi Musterfrau");
04 person01.set_pers_nr(400333);
05 person01.set_gehalt(2000);
06 person01.print_data();

07 Personal person02("Maxi Musterfrau", 12345, 1500);
08 person02.print_data();
09 person02.set_pers_nr(person01.get_pers_nr()+1);
10 person02.set_gehalt(person01.get_gehalt());
11 person02.print_data();
```

In Zeile **01** erzeugen wir ein neues Objekt, dessen Inhalt wir gleich in Zeile **02** mit `print_data()` ausgeben, wo gleich alle drei Getter ausgeführt werden und auch melden, dass unser Objekt, welches ja lediglich mit dem Standardkonstruktor initialisiert wurde, noch keinen gültigen String und keine gültige Personalnummer enthält. In den Zeilen **03**, **04** und **05** setzen wir dann entsprechende Werte und lassen diese erneut mit `print_data()` in Zeile **06** ausgeben.

In Zeile **07** erzeugen wir ein weiteres Objekt mit absichtlich falscher Personalnummer und weit unter dem Mindestgehalt, was die folgende Ausgabe mit `print_data()` in Zeile **08** mit den darin verwendeten Gettern auch bemängeln wird. In den Zeilen **09** und **10** setzen mit dann die Personalnummer und das Gehalt des Objekts auf einen gültigen Wert. Dabei bedienen wir uns beim Rückgabewert der Getter vom Objekt, welches wir in Zeile **01** instanziiert haben. Zu guter Letzt geben wir auch hier in Zeile **11** nochmals alle Eigenschaften des Objekts aus.

Das Programm bei der Ausführung:

```
Noch kein Name vergeben!!!
Name      :
Personalnummer noch nicht vergeben!!!
Pers.Nr.: 0
Gehalt   : 1900
```

Name : Maxi Musterfrau
 Pers.Nr.: 400333
 Gehalt : 2000

Name : Maxi Musterfrau
 Ungültige Personalnummer!!!
 Pers.Nr.: 12345
 Achtung unter dem Mindestgehalt!!!
 Gehalt : 1500

Name : Maxi Musterfrau
 Pers.Nr.: 400334
 Gehalt : 2000

Klassen nur aus Zugriffsmethoden?

Klassen sollten natürlich nicht nur aus Zugriffsfunktionen (Settern und Gettern) bestehen. Wenn Sie ein wenig nach Settern und Gettern im Internet suchen, werden Sie feststellen, dass die Meinungen über ihre Verwendung ziemlich auseinandergehen. Definitiv stellt sich die Frage nach dem Sinn einer Klasse, die nur Getter und Setter enthält und wo einfach nur ein Wert direkt an die Eigenschaften übergeben (bzw. von diesen zurückgegeben) wird, wie bspw.:

```
void Personal::set_gehalt(unsigned int g) {
    gehalt = g;
}
```

Besteht Ihre Klasse nur aus solchen leeren Settern und Gettern, ist dies fast schon so, als würden Sie die Eigenschaften in den `public`-Bereich stellen. Zumindest würden Sie sich damit eine Menge Zeilen an Code sparen – was Sie aber trotzdem nicht machen sollten. Denn selbst so leere Setter haben den Vorteil (als Grundgerüst), dass Sie, falls Sie mal nachträglich etwas ändern müssen/wollen, dies hierüber jederzeit tun können und nicht im kompletten Code danach suchen müssen. Ansonsten sollten Setter auch genutzt werden, um Überprüfungen auf gültige Werte durchzuführen, bevor Sie einer Eigenschaft einen (neuen) Wert übergeben oder irgendetwas zurückgeben.

7.7.3 Zugriff auf die öffentlichen Mitglieder einer Klasse

Nachdem unsere Klasse mit einigen öffentlichen Methoden belegt ist, soll hier nochmals kurz auf den Zugriff darauf eingegangen werden. Das Thema wurde ja bereits in Abschnitt 7.6, »Exkurs: Zugriffskontrolle auf die Klassenmitglieder«, behandelt. Allerdings war es dort doch recht theoretisch.

Punktoperator

In den meisten Beispielen haben und werden Sie ganz einfach mit dem Punktoperator nach dem instanziierten Objekt auf die öffentlichen Mitglieder einer Klasse zugreifen – wie Sie dies ja bereits schon von den Strukturen her kennen. Dazu muss hier, denke ich, nicht mehr allzu viel geschrieben werden.

```
01 Personal person01;
02 person01.set_gehalt(3000);
```

In Zeile **02** setzen Sie den privaten Wert `gehalt` des Objekts `person01` mithilfe der Methode `set_gehalt()` auf 3000. Mithilfe des Punktoperators hinter dem Objektbezeichner realisieren Sie den Zugriff auf die öffentliche Methode.

Punktoperator bei Vektoren bzw. Arrays

Es wurde bereits bei der Erzeugung von Objekten (siehe Abschnitt 7.3, »Objekte einer Klasse erzeugen«) erwähnt, dass Sie möglichst auf dynamische Speicherreservierungen bei Objekten verzichten und stattdessen auf fertige Standard-Containerklassen, wie bspw. `std::vector`, zurückgreifen sollten. Der Zugriff mithilfe von Objekten von Standardcontainern einer bestimmten Klasse ist nicht unwesentlich komplizierter als mithilfe des Punktoperators. Der folgende Codeausschnitt zeigt Ihnen kurz, wie einem leeren `std::vector<Personal>` neue Objekte hinzugefügt werden und diese auch wieder ausgegeben werden können:

```
00 // kapitel07/7_7_3/main.cpp
...
01 #include <vector>
...
02 std::vector<Personal> zeitarbeiter;
03 Personal tmp;
04 tmp.set_name("Max Mustermann");
05 tmp.set_pers_nr(400432);
06 tmp.set_gehalt(3000);
07 zeitarbeiter.push_back(tmp);

08 Personal person01("Franz Mustersohn", 400455);
09 Personal person02("Maxi Musterfrau");
10 zeitarbeiter.push_back(person01);
11 zeitarbeiter.push_back(person02);

12 for(unsigned int i=0; i < zeitarbeiter.size(); ++i) {
13     zeitarbeiter.at(i).print_data();
14     // oder: zeitarbeiter[i].print_data();
15 }
```

In Zeile **02** erstellen wir ein leeres `std::vector` der Klasse `Personal`. In Zeile **03** wird ein temporäres Objekt angelegt, welchem wir in den Zeilen **04** bis **06** Daten übergeben und in Zeile **07** ans Ende unseres Vektors `zeitarbeiter` mit der Methode `push_back()` hängen.

In den Zeilen **08** und **09** werden zwei weitere Objekte der Klasse `Personal` erzeugt und in den Zeilen **10** und **11** jeweils am Ende des Vektors hinzugefügt.

Am Ende geben wir die Eigenschaften von jedem Element im Vektor in einer Schleife (Zeile **12** bis **15**) auf dem Bildschirm aus. Die Anzahl der Elemente im Vektor können Sie mit der Methode `size()` ermitteln. Für den entsprechenden Index im Vektor verwenden wir `at(i)` und für die Ausgabe die Methode `print_data()`. Alternativ können Sie hierbei auch den Indizierungsoperator `[i]` verwenden. In C++11 hätten Sie die Schleife zur Ausgabe aller Eigenschaften auch wie folgt implementieren können:

```
...
// C++11-Style:
for( Personal &p : zeitarbeiter ) {
    p.print_data();
}
```

Tip: Optimierung mithilfe der Move-Semantik

Wenn Sie in diesem Beispiel die Ausgabe des Kopierkonstruktors und/oder des Destruktors aktiviert haben, werden Sie feststellen, dass hierbei eine Menge Kopien gemacht werden (u.a. auch für jedes `push_back()`). So etwas schreit geradezu nach einer Optimierung mit der neuen Move-Semantik von C++11. Auf der Buch-CD finden Sie daher im Listing den Kopierkonstruktor und den Move-Konstruktor auskommentiert, wo Sie nach Belieben experimentieren können. Hier würde sich bspw. anbieten, dem Kopierkonstruktor ein `delete` zuzuweisen und so nur noch den Move-Konstruktor (der natürlich implementiert sein muss) auszuführen. Der Move-Konstruktor wurde bereits kurz im Abschnitt »Verschiebekonstruktor (C++11)« in Abschnitt 7.4.2 beschrieben und wird nochmals in Abschnitt 7.10, »Klassen für das Verschieben schreiben (C++11)«, behandelt.

Indirekter Zugriff

Wenn Sie einen rohen Zeiger verwenden, müssen Sie einen indirekten Zugriff mit dem Pfeiloperator `->` auf die öffentlichen Mitglieder einer Klasse verwenden. Ein einfacher Codeausschnitt dazu:

```
01 Personal person01("Max Mustermann", 400654, 2500);
02 Personal *PersonPtr = &person01;
03 PersonPtr->set_gehalt(3300);
04 PersonPtr->print_data();
```

In Zeile **02** übergeben wir dem Zeiger `PersonPtr` die Adresse von `person01`. In Zeile **03** setzen wir über die öffentliche Methode `set_gehalt()` den privaten Wert `gehalt` von `person01` indirekt mit dem Zeiger `PersonPtr` auf 3300. Für den indirekten Zugriff von Zeigern wird der Pfeiloperator verwendet. Auch die Ausgabe aller Eigenschaften von `person01` in Zeile **04** führen wir indirekt über den `PersonPtr` durch.

7.7.4 `const`-Methoden (read-only)

Nicht immer will man, dass eine Methode schreibend auf die privaten Eigenschaften einer Klasse zugreifen darf. Ein gutes Beispiel dafür stellen unsere Abfragemethoden (die Getter) dar, mit denen Sie ohne Weiteres, sei es aus Versehen oder purer Faulheit, Folgendes schreiben könnten:

```
...
53 unsigned int Personal::get_gehalt() {
54     // Mindestgehalt prüfen und gegebenenfalls setzen
55     if( gehalt < GG ) {
56         std::cout << "Achtung unter dem Mindestgehalt!!!\n";
57         gehalt = GG;
58     }
59     ...
62 }
```

Zugegeben, der schreibende Zugriff in Zeile 57 stellt nicht wirklich einen Fehler dar, aber der Übersichtlichkeit zuliebe sollten Sie das Setzen von Werten den `set`-Methoden überlassen. Damit Ihnen solche Dinge nicht versehentlich passieren, müssen Sie diese Methoden einfach mit dem Schlüsselwort `const` am Ende der Deklaration und Definition kennzeichnen.

Die Deklaration dazu würde daher wie folgt aussehen:

```
// kapitel07/7_7_4/personal.h
class Personal {
...
public:
...
    unsigned int get_gehalt() const;
};
...
```

Jetzt noch die Definition dazu:

```
00 // kapitel07/7_7_4/personal.cpp
...
```

```

53 unsigned int Personal::get_gehalt() const {
54     // Mindestgehalt prüfen und gegebenenfalls setzen
55     if( gehalt < GG ) {
56         std::cout << "Achtung unter dem Mindestgehalt!!!\n";
57         gehalt = GG; // Jetzt Fehler !!! Nur Lesen erlaubt !!!
58     }
...
62 }

```

Dank des Schlüsselwortes `const` am Ende der Methode würde der Compiler die Übersetzung wegen Zeile 53 mit einer Fehlermeldung quittieren, weil versucht wird, die Eigenschaft `gehalt` in der Methode zu ändern, wo doch in der Methode jetzt nur noch Lesen erlaubt ist. Selbst der Versuch über `set_gehalt(GG)`; würde in Zeile 57 nicht mehr funktionieren!

Neben `get_gehalt()` sollten Sie auch noch `get_pers_nr()` und `get_name()` mit `const` kennzeichnen und nur das Lesen erlauben.

Konstante Objekte

Wenn wir schon bei `const` sind, ist ein guter Zeitpunkt, hier kurz konstante Objekte »einzustreuen«. Wie auch jeden gewöhnlichen Datentyp können Sie selbstverständlich auch Objekte bei der Erzeugung mit `const` kennzeichnen. Das würde bedeuten, dass ein Objekt nur noch lesbar wäre:

```

01 const Personal person01("Max Mustermann", 400654, 2500);
02 std::cout << person01.get_name() << std::endl; // OK
03 person01.set_gehalt(2000); // !!! Fehler !!!!

```

Der lesende Zugriff in Zeile 02 macht noch keine Probleme. Aber der schreibende Zugriff in Zeile 03 funktioniert nicht mehr. Zwar beschwert sich der Compiler, dass es eine Methode mit der Signatur `Personal::set_gehalt(unsigned int) const` nicht gibt, aber eine Nachimplementierung dieser `set`-Methode wäre nicht unbedingt sinnvoll.

Für solche Fälle können Sie sich auf zwei Arten rüsten. Entweder Sie reagieren darauf und implementieren entsprechende Methoden, welche dann entsprechend reagieren (bspw. Fehlermeldung ausgeben). Ein Beispiel:

```

// Deklaration in personal.h
...
class Personal {
...
public:
...

```



```

    void set_gehalt(unsigned int g);
    void set_gehalt(unsigned int g) const;
...
};

// Definition in personal.cpp
...
void Personal::set_gehalt(unsigned int) const {
    std::cerr
        << "Fehler: const-Objekte sind nicht beschreibbar\n";
}

```

Eine zweite Alternative wäre, dass Sie solche Zugriffsmethoden verbieten, indem Sie diese entweder in den `private`-Bereich der Klasse (C++03) stecken oder mit `delete` (C++11) gleich beim Design verbieten:

```

// Deklaration in personal.h
...
class Personal {
...
public:
...
    void set_gehalt(unsigned int g);
    void set_gehalt(unsigned int g) const = delete; // C++11
...
};

```

7.7.5 Objekte als Methodenparameter

Wie Sie gewöhnliche Basisdatentypen als Funktionsparameter in Form einer Kopie, Referenz, eines rohen Zeigers usw. übergeben, konnten Sie bereits in Abschnitt 5.5, »Funktionsparameter«, sehen. Zwar wurden dort die Funktionen behandelt, aber den Ablauf können Sie für Objekte als Methodenparameter ebenfalls übernehmen. In diesem Abschnitt soll das Thema nochmals aufgerollt werden, aber mit dem Fokus auf Objekten als Methodenparametern.

Ziel soll es hier sein, eine einfache Methode zu schreiben, um die Personalnummer zweier Objekte von der Klasse `Personal` zu vergleichen. Nach unserem Wissensstand können wir dies im Augenblick mit einer Referenz als Parameter (siehe Abschnitt 5.5.2, »Funktionsparameter als Referenz (Call by Reference)«) realisieren. Eine Kopie kommt auf keinen Fall in Frage, weil hier extra für den Parameter jeweils umsonst ein Konstruktor- und ein Destruktoraufruf durchgeführt werden müssten. Die rohen Zeiger sind umständlich zu verwenden, und dafür wurden in C++ ja schließlich auch

die Referenzen eingeführt. All diese Themen wurden ja in Abschnitt 5.5 des Buches bereits umfassend behandelt. Der Aufruf würde demnach wie folgt aussehen:

```
Klasse obj1, obj2;
...
obj1.methode( obj2 );
```

Bezogen auf unsere Klasse `Personal` schreiben Sie die Deklaration der Methoden dafür in den `public`-Bereich der Klassendefinition der Headerdatei *personal.h*:

```
00 // kapitel07/7_7_5/personal.h
...
01 class Personal {
...
02     public:
...
03         bool compare_pers_nr(const Personal &p2) const;
};
...
```

Da wir einen Vergleich anstellen wollen, gibt diese Methode `bool` zurück, und weil darin auch nichts geändert werden darf, kennzeichnen wir diese Methode auch gleich noch mit `const`.

Die Definition dieser Methode erfolgt wieder in einer separaten Quelldatei, in unserem Fall wieder in *personal.cpp*, wo diese wie folgt aussieht:

```
00 // kapitel07/7_7_5/personal.cpp
...
01 bool Personal::compare_pers_nr(const Personal &p2) const {
02     if( get_pers_nr() == p2.get_pers_nr() ) {
03         return true; // Beide Nummern sind gleich
04     }
05     return false; // Nummern sind unterschiedlich
06 }
```

Der Code bedarf wohl keiner großen Erläuterung. In Zeile 02 vergleichen wir die Personalnummer des aufrufenden Objekts (`obj1.methode(obj2)`) mit `get_pers_nr()` mit der Personalnummer des als `const`-Referenz übergebenen Objekts (`obj1.methode(obj2)`) mit `p2.get_pers_nr()`. `true` wird zurückgegeben, wenn beide Nummern gleich sind, und `false`, wenn nicht.

Aufrufen können Sie diese neue Methode, wie Sie dies bereits von den Zugriffsmethoden her kennen. Hierzu ein Codeausschnitt in der `main`-Funktion:

```
00 // kapitel07/7_7_5/main.cpp
...
```

```

01 Personal person1("Max Mustermann", 400321, 2200);
02 Personal person2("Maxi Musterfrau", 400231, 2000);
03 Personal person3("Franz Mustersohn", 400321, 3100);

04 if(person1.compare_pers_nr(person2)) {
05     std::cout << "Achtung! Pers.Nr. sind identisch!!!\n";
06     std::cout << person1.get_name() << "=="
                << person2.get_name() << std::endl;
07 }
08 if(person1.compare_pers_nr(person3)) {
09     std::cout << "Achtung! Pers.Nr. sind identisch!!!\n";
10     std::cout << person1.get_name() << "/"
                << person3.get_name() << std::endl;
11 }
...

```

Hier werden jeweils zwei Vergleiche durchgeführt. Während der Vergleich in Zeile **04** noch `false` zurückliefert, wird der Vergleich in Zeile **08** `true` zurückgeben, und der Anweisungsblock dahinter wird mit dem Hinweis ausgeführt, welche Personen dieselbe Personalnummer haben.

In der Praxis könnte man sich solche Vergleiche sparen, indem man gleich von Anfang an beim Konstruktor darauf achtet, dass jede Person eine andere Personalnummer bekommt. Es gibt mehrere Möglichkeiten, dies zu tun. Ein Beispiel wäre, die Eigenschaft als statische Eigenschaft in der Klasse zu schreiben. Ein Beispiel dazu finden Sie in Abschnitt 7.8.3, »Statische Eigenschaften in einer Klasse«.

==-Operator überladen

In der Praxis löst man solche Vergleiche von Objekten einer Klasse auch, indem man einen Operator wie `==` oder `!=` überlädt. Die Operatorüberladung wird in Kapitel 8, »Operatoren überladen«, behandelt.

7.7.6 This-Zeiger

Außerhalb einer Methode einer Klasse ist es einfach, die Adresse für ein Objekt über den Adressoperator mit bspw. `&obj` zu ermitteln bzw. auszugeben. Aber innerhalb einer Methode funktioniert das nicht mehr. Daher enthält jede Methode, die nicht mit dem Schlüsselwort `static` gekennzeichnet wurde, einen versteckten Zeiger auf das Objekt, mit dem sie aufgerufen wurde – vereinfacht praktisch ein Zeiger auf die eigene Instanz. Bei einem Aufruf von `obj.methode()` finden Sie in `methode(){}` einen versteckten Zeiger mit dem Namen `this` auf das Objekt `obj`. Folgender Codeausschnitt soll Ihnen das kurz demonstrieren:

```

00 // kapitel07/7_7_6_simple/simple.cpp
...
01 class SimpleClass {
02 public:
03     void simplePrint() const;
04 };

05 void SimpleClass::simplePrint() const {
06     std::cout << this << std::endl;
07 }

08 int main() {
09     SimpleClass obj;
10     obj.simplePrint();
11     std::cout << &obj << std::endl;
12 }

```

In der Klasse `SimpleClass` (Zeile 01 bis 04) wird nur eine einzige Methode definiert (Zeile 05 bis 07), welche nichts anderes macht, als die Adresse des `this`-Zeigers auszugeben – also die Adresse für die Instanz eines Objekts der Klasse `SimpleClass`. Aufgerufen wird diese Methode in Zeile 10 mit dem Objekt `obj`. Natürlich entspricht die Adresse des Objekts innerhalb einer Methode der Adresse mit dem Adressoperator, welche in Zeile 11 ausgegeben wird.

In der Praxis können Sie diesen `this`-Zeiger verwenden, um bspw. innerhalb einer Methode mit `*this` auf das ganze Objekt zuzugreifen oder es gar zurückzugeben (bspw. bei der `=`-Operatorüberladung, siehe Abschnitt 8.2, »Zuweisungsoperator – operator«). An dieser Stelle ein sinnvolles Beispiel zu erstellen, ist schwer, daher folgt hier nur kurz ein Codeausschnitt, in dem `*this` verwendet wird, um zwei Objekte miteinander zu tauschen. Zunächst die Deklaration in der Klassendefinition im `public`-Bereich in der Headerdatei *personal.h*:

```

00 // kapitel07/7_7_6/personal.h
...
01 class Personal {
02     std::string name;        // Name
03     unsigned int pers_nr;    // Personalnummer
04     unsigned int gehalt;     // monatliches Gehalt
05 public:
...
06     void myswap(Personal &p2);
07 };

```

Jetzt zur Definition der Methode `Personal::myswap` in der Quelldatei `personal.cpp` und dem `this`-Zeiger beim Einsatz:

```
00 // kapitel07/7_7_6/personal.cpp
...
01 void Personal::myswap(Personal &p2) {
02     Personal tmp = p2;
03     p2 = *this;
04     *this = tmp;
05 }
```

In Zeile **02** wird ein temporäres Objekt `tmp` erzeugt, welches den Inhalt von `p2` zwischenspeichert. In Zeile **03** verweist die Referenz `p2` jetzt auf das aufrufende Objekt mithilfe von `*this`, wodurch im Augenblick beide (`*this` und `p2`) auf dasselbe Objekt referenzieren. Am Ende in Zeile **04** bekommt das aktuell aufrufende Objekt (`*this`) den Inhalt des zuvor temporär gesicherten Objekts in `tmp`, womit der Tausch der Daten beider Objekte abgeschlossen wäre. Ohne den `this`-Zeiger wäre dies so kaum möglich gewesen.

Jetzt noch ein Beispiel, welches den Tausch der Eigenschaften zweier `Personal`-Objekte demonstrieren soll:

```
00 // kapitel07/7_7_6/main.cpp
...
01 Personal person01("Max Mustermann", 400321, 2200);
02 Personal person02("Maxi Musterfrau", 400231, 2000);
03 person01.print_data();
04 person02.print_data();
05 person01.myswap(person02);
06 person01.print_data();
07 person02.print_data();
...
```

Objekte tauschen mit `std::swap` – die bessere Alternative

Um zu demonstrieren, dass mit `*this` innerhalb einer Methode auf ein Objekt als Ganzes zugegriffen werden kann, war die Methode `Personal::myswap` recht gut geeignet. Aber in der Praxis kann ich Ihnen diese Methode so nicht empfehlen. Sicherlich hängt dies auch vom Anwendungsfall ab, aber im konkreten Beispiel bietet Ihnen das Funktions-Template `std::swap()` eine wesentlich effektivere und erprobte Alternative. Um `std::swap()` verwenden zu können, müssen Sie die den Header `<utility>` (in C++03 noch `<algorithm>`) mit angeben. Bezogen auf unser Beispiel könnten Sie das Funktions-Template wie folgt einsetzen:

```
05 std::swap(person01, person02);
```

Theoretisch könnten hiermit auch gleichnamige Bezeichner (oder Mitglieder einer Klasse im Allgemeinen) innerhalb einer Methode verwendet werden. Würde es hier eine lokale Variable `xyz` und eine Eigenschaft der Klasse mit `xyz` geben, könnten Sie mit `this->xyz` auf das Mitglied der Klasse zugreifen und mit `xyz` auf den lokalen Bezeichner – trotzdem sollte man in der Praxis keine gleichen Bezeichner verwenden. Bezogen auf die Methoden `Personal::compare_pers_nr(const Personal&)` aus Abschnitt 7.7.5 könnten Sie quasi auch wie folgt auf das Mitglied einer Klasse zugreifen:

```
00 // kapitel07/7_7_5/this/personal.cpp
...
01 bool Personal::compare_pers_nr(Personal &p2) const {
02     if( this->get_pers_nr() == p2.get_pers_nr() ) {
03         return true; // Beide Nummern sind gleich
04     }
05     return false; // Nummern sind unterschiedlich
06 }
```

Diese explizite Verwendung vom `this`-Zeiger auf Mitglieder der Klasse ist, wie in diesem Beispiel in Zeile 02 gezeigt wurde, nicht nötig und kann hier auch weggelassen werden.

7.7.7 Objekte als Rückgabewert

Alles, was Sie bereits im Zusammenhang mit Rückgabewerten von Funktionen gelesen haben (siehe Abschnitt 5.6, »Rückgabewert aus einer Funktion«), können Sie auch für die Methoden von Klassen übernehmen. In diesem Abschnitt soll zusätzlich noch auf die Rückgabe von Objekten aus Methoden eingegangen werden.

Im folgenden Beispiel soll eine Methode implementiert werden, welche ein neues Objekt der Klasse `Personal` zurückgibt. Im Beispiel werden zwei Objekte zum Vergleichen verwendet, und als Rückgabewert folgt das Objekt mit der am besten verdienen Person (`Personal::get_gehalt()`). Hierzu zunächst die Deklaration in der Headerdatei `personal.h`:

```
00 // kapitel07/7_7_7/personal.h
...
01 class Personal {
02     std::string name;        // Name
03     unsigned int pers_nr;    // Personalnummer
04     unsigned int gehalt;     // monatliches Gehalt
05 public:
...
06     Personal compare_gehalt(Personal &p2) const;
07 };
```

Jetzt folgt noch die Definition der Methode in der Quelldatei *personal.cpp*:

```
00 // kapitel07/7_7_7/personal.cpp
...
01 Personal Personal::compare_gehalt(Personal &p2) const {
02     if( get_gehalt() > p2.get_gehalt() ) {
03         return *this;
04     }
05     return p2;
06 }
```

In Zeile **02** wird verglichen, ob das Gehalt des aufrufenden Objekts (auch möglich als `this->get_gehalt()`) größer ist als das als Referenz übergebene Objekt `p2`. Trifft dies zu, wird das aufrufende Objekt als Ganzes mit `*this` als Kopie zurückgegeben. Ansonsten wird `p2` als Kopie zurückgegeben. Alternativ könnte hier auch eine dynamische Speicherreservierung verwendet werden.

Jetzt noch die Verwendung der Methode `Personal::compare_gehalt()` in der Praxis:

```
00 // kapitel07/7_7_7/main.cpp
...
01 Personal person01("Max Mustermann", 400321, 2200);
02 Personal person02("Maxi Musterfrau", 400231, 2400);
03 Personal person2expensive = person01.compare_gehalt(person02);

04 person2expensive.print_data();
...
```

In Zeile **03** wird ein neues Objekt erzeugt, welches entweder mit den Eigenschaften vom Objekt `person01` oder `person02` initialisiert wird. Dies ist abhängig davon, wer von den beiden Personen mehr verdient.

7.7.8 Globale Hilfsfunktionen

In der Praxis wird empfohlen, für Algorithmen eine Klasse für Methoden zu schreiben. Trotzdem werden Sie feststellen, dass selbst die Standardbibliothek eine Menge an Hilfsfunktionen anbietet. Häufig handelt es sich hierbei auch um Funktions-Templates (siehe Abschnitt 11.1, »Funktions-Templates«). Nicht immer müssen Sie eine weitere Methode erstellen, sondern können auch eine Hilfsfunktion (eine gewöhnliche Funktion) mit den Klassen als Parameter schreiben.

Wenn Sie eine solche Hilfsfunktion für Ihre Klasse schreiben wollen/sollen, wird diese gewöhnlich auch in derselben Datei deklariert, in der die Klassendefinition steht, und in derselben Datei definiert, in der Sie die Methoden definieren. Es kann

außerdem hilfreich sein, einen extra Namensbereich dafür zu erstellen, wenn Sie viele Klassen und Hilfsfunktionen in Ihrem Projekt verwenden.

Wertübergabe als Kopie, roher Zeiger oder Referenz?

Die Wertübergabe als **Kopie** (siehe Abschnitt 5.5.1, »Funktionsparameter als Kopie (Call by Value)«) würde ich nicht empfehlen, weil für jedes Objekt im Parameter beim Funktionsaufruf der Konstruktor und am Ende wieder der Destruktor aufgerufen werden muss. Der Aufwand, der hierbei betrieben werden muss, ist einfach zu groß.

Einen rohen **Zeiger** (siehe Abschnitt 5.5.3, »Funktionsparameter als rohe Zeiger«) als Wertübergabe eines Objekts an die Hilfsfunktion würde ich Ihnen hier auch nicht empfehlen, weil die Verwendung von rohen Zeigern in Funktionen den Code nur verkompliziert und zum Fehlermachen einlädt. Außerdem sind rohe Zeiger nicht sehr C++-like, und schließlich wurden u.a. speziell hierfür die Referenzen eingeführt.

Wie in den meisten Fällen ist die Wertübergabe vom Objekt als **Referenz** an eine Hilfsfunktion oft die beste Lösung (siehe Abschnitt 5.5.2, »Funktionsparameter als Referenz (Call by Reference)«). Damit genießen Sie dieselben Vorzüge wie bei den Zeigern und haben denselben Komfort wie bei der Übergabe als Kopie.

Da Hilfsfunktionen häufig nur kleine Helferlein sind, sollten Sie versehentliche Veränderungen der Eigenschaften verhindern, indem Sie die Parameter auch gleich mit `const` signieren (siehe Abschnitt 5.5.5, »Schreibschutz für die Referenzparameter mit `const`«).

Hierzu soll nun eine einfache Hilfsfunktion erstellt werden, welche zwei `Personal`-Objekte Eigenschaft für Eigenschaft miteinander vergleicht und `true` zurückgibt, wenn beide Objekte komplett identische Eigenschaften enthalten. Ansonsten wird `false` zurückgegeben.

Zunächst die Deklaration in der Headerdatei *personal.h*:

```
00 // kapitel07/7_7_8/personal.h
...
01 class Personal;
02 bool compare_Personal(const Personal &p1, const Personal &p2);

03 class Personal {
...
04 };
```

Die globale Hilfsfunktion wird hierbei in Zeile 02 noch vor der Klassendefinition geschrieben. Da an dieser Stelle dem Compiler die Klasse noch gar nicht bekannt ist, würde sich dieser mit einer Fehlermeldung beschweren. Daher wurde hier in Zeile 01

eine Vorausdeklaration geschrieben, um die Parameter der Hilfsfunktion mit dem Klassennamen bekannt zu machen. Alternativ könnten Sie die Funktionsdeklaration auch hinter der Klassendefinition nach Zeile 04 schreiben. Dann könnten Sie hier auf die Vorausdeklaration in Zeile 01 verzichten.

Jetzt zur Funktionsdefinition in der Quelldatei *personal.cpp*:

```
00 // kapitel07/7_7_8/personal.cpp
...
01 bool compare_Personal(const Personal &p1, const Personal &p2) {
02     if( p1.get_name() == p2.get_name() &&
        p1.get_pers_nr() == p2.get_pers_nr() &&
        p1.get_gehalt() == p2.get_gehalt() ) {
03         return true;
04     }
05     return false;
06 }
```

Die Hilfsfunktion benötigt keinen Scope-Zugriff via `Personal::`, weil es sich ja jetzt um eine gewöhnliche globale Funktion handelt. In Zeile 02 überprüfen wir alle drei Eigenschaften von den Objekten `p1` und `p2` mit dem logischen Und-Operator auf Gleichheit.

Jetzt fehlt nur ein kurzer Codeausschnitt, welcher diese globale Hilfsfunktion im Einsatz demonstrieren soll:

```
00 // kapitel07/7_7_8/main.cpp
...
01 Personal person01("Max Mustermann", 400321, 2200);
02 Personal person02("Max Mustermann", 400321, 2200);

03 if( compare_Personal(person01, person02) ) {
04     std::cout << "Fehler: Gleicher Datensatz gefunden!!!\n";
05 }
```

7.7.9 Globale friend-Funktion

Da die Eigenschaften gewöhnlich im `private`-Bereich der Klassendefinition geschrieben stehen, besteht von außen kein Zugriff auf diese Eigenschaften. Nur so ist sichergestellt, dass die Eigenschaften vor unsachgemäßer Änderung geschützt sind und dass OOP-Prinzip der Datenkapselung eingehalten wird. Folgendes ist, bezogen auf unsere Klasse `Personal`, bspw. nicht erlaubt:

```
00 // kapitel07/7_7_9/main.cpp
...
```

```

01 void printout(Personal &p) {
02     std::cout << p.name << std::endl; // !!! Fehler !!!
03 }
...

04 Personal person01("Max Mustermann", 400321, 2200);
05 printout(person01);

```

Einen Zugriff, wie er in Zeile 02 zu sehen ist, wird der Compiler mit einer Fehlermeldung quittieren, weil die Eigenschaft `name` in der Klasse `Personal` mit `private`-Zugriffsrechten versehen ist. Der Zugriff hier ist (korrekterweise) nur über die Zugriffsfunktion `get_name()` möglich.

Trotzdem gibt es in C++ eine Möglichkeit, die Eigenschaften einer Klasse von außen zu erreichen, indem man diese Funktion mit dem Schlüsselwort `friend` zum Freund erklärt. Hierzu müssen Sie lediglich die globale Funktion in der Klassendefinition mit dem Schlüsselwort `friend` deklarieren:

```

00 // kapitel07/7_7_9/personal.h
...
01 class Personal {
02     std::string name;      // Name
03     unsigned int pers_nr;  // Personalnummer
04     unsigned int gehalt;   // monatliches Gehalt
05 public:
...
06     friend void printout(Personal &p);
07 };
...

```

Dank des Schlüsselwortes `friend` ist es jetzt möglich, mit einer globalen Funktion auf die privaten Eigenschaften einer Klasse zuzugreifen.

Solche `friend`-Funktionen weichen allerdings die Datenkapselung auf und sollten nicht zur Gewohnheit werden. Es gibt aber einen begründeten Fall, in dem es ohne eine `friend`-Funktion gar nicht erst geht, nämlich bei einigen Operatorüberladungen wie dem unären `+` oder den Operatoren `<<` und `>>` für die Aus- und Eingabe von Objekten. Die Operatorüberladung wird in Kapitel 8, »Operatoren überladen«, behandelt.

Kein `this`-Zeiger für Freunde

Eine `friend`-Funktion ist keine Methode einer Klasse, weshalb hier auch kein `this`-Zeiger zur Verfügung steht.

friend-Klassen

Neben friend-Funktionen gibt es auch friend-Klassen, die in Abschnitt 7.11, »Friend-Klassen«, kurz behandelt werden.

7.7.10 Methoden überladen

Alles, was Sie bereits in Abschnitt 5.7, »Funktionen überladen«, gelesen haben, können Sie auch für das Überladen von Methoden übernehmen. Auch hier gilt, dass sich die Signatur der Methoden mindestens in einem der folgenden Dinge unterscheiden muss:

1. Die Datentypen der Parameter der Methoden sind unterschiedlich:

```
00 // kapitel07/7_7_10/personal.h
...
01 class Personal {
02     std::string name;      // Name
03     unsigned int pers_nr;  // Personalnummer
04     unsigned int gehalt;   // monatliches Gehalt
05 public:
...
06     void set_data();
07     void set_data(std::string n);
08     void set_data(unsigned int pn);
...
09 };
```

2. Die einzelnen Methoden haben eine unterschiedliche Anzahl an Parametern:

```
00 // kapitel07/7_7_10/personal.h
...
01 class Personal {
02     std::string name;      // Name
03     unsigned int pers_nr;  // Personalnummer
04     unsigned int gehalt;   // monatliches Gehalt
05 public:
...
06     void set_data(std::string n);
07     void set_data(std::string n, unsigned int pn);
08     void set_data(std::string n, unsigned int pn, unsigned int g);
...
09 };
```

Die Definition der Methodenüberladung sollte kein großes Problem mehr für Sie darstellen. Diese können Sie ja zur Übung selbst implementieren oder den Quellcode von der Buch verwenden.

7.7.11 Die neue Funktionssyntax – decltype und auto (C++11)

Dieser Abschnitt kann als Fortsetzung von Abschnitt 5.9, »Die neue Funktionssyntax (C++11)«, gesehen werden, wo Sie die neue Funktionssyntax und `decltype` bereits in der Praxis mit globalen Funktionen kennengelernt haben. In diesem Abschnitt soll das Beispiel jetzt noch mit Methoden von Klassen verwendet werden.

Die neue Rückgabesyntax mit Methoden

In Abschnitt 5.9 haben Sie bereits gesehen, wie Sie einen Funktionskopf mit der neuen Rückgabesyntax »vereinfachen« können, anstatt Folgendes zu verwenden:

```
int addieren( int a, int b );
```

Wenn Sie diesen Funktionsprototyp jetzt mit der neuen Rückgabesyntax verwenden wollen, sieht das wie folgt aus:

```
auto addieren( int a, int b ) -> int;
```

In solch einem Beispiel erscheint die »Vereinfachung« wohl eher trivial. Daher jetzt eine einfache Klasse, bei der Sie die »Vereinfachung« etwas deutlicher erkennen können:

```
class OS {
public:
    enum OSType { WIN, MAC, UNIX, LINUX };
    void setOS( OSType t );
    OSType getOS( ) const;
private:
    OSType type;
};
```

Die Schwierigkeit an dieser Klasse ist jetzt, die Methode `OS::getOS()` zu definieren. Gerne wird diese wie folgt realisiert:

```
OSType OS::getOS( ) const { // Fehler !!!
    return type;
}
```

Der Compiler würde diese Methode mit einer Fehlermeldung quittieren, weil er mit `OSType` als Rückgabewert an dieser Stelle nichts anfangen kann. Richtig in diesem Beispiel wäre Folgendes gewesen:

```
OS::OSType OS::getOS( ) const {
    return type;
}
```

Um künftig nicht mehr über solch komplexe Scope-Operatoren-Orgien zu stolpern, könnten Sie diesen Code mit der neuen Rückgabesyntax wie folgt ändern:

```
...
auto OS::getOS( ) const -> OSType {
    return type;
}
```

Mit der neuen Rückgabesyntax können wir uns jetzt darauf verlassen, dass der Compiler hier die richtige Rückgabesyntax für uns verwendet.

decltype für Eigenschaften und Methoden

Wollen Sie hingegen Typinformationen aus einem Ausdruck herausziehen, kann Ihnen `decltype` behilflich sein. Wenn Sie bspw. eine fremde Schnittstelle verwenden, aber nicht genau wissen oder keine Lust haben, nachzusehen, was für einen Typ diese Funktion zurückgibt, dann verwenden Sie einfach `decltype` wie folgt:

```
01 #include <ctime> // für time
...
02 class Messen {
03     decltype(time(nullptr)) stamp;
04     public:
05         void setStamp() {
06             stamp = time(nullptr);
07         }
08         auto getStamp() const -> decltype(time(0L)) {
09             return stamp;
10         }
11 };
```

In Zeile 03 wurde eine Membervariable `stamp` verwendet. Da ich hier keine Lust hatte, nachzusehen, welchen Typ die Uralt-Funktion `time()` aus der Headerdatei `<ctime>` zurückliefert, habe ich das Ganze einfach in `decltype(time(nullptr))` gepackt und dem Compiler die Arbeit überlassen, den Typ für diesen Ausdruck zu ermitteln. Das selbe funktioniert auch mithilfe der neuen Rückgabesyntax, wie dies in Zeile 08 demonstriert wird. Hier sehen Sie gleich den Vorteil aller Typinferenzen mit `auto`, neuer Rückgabesyntax und `decltype` gleichzeitig im Einsatz. Der Compiler macht intern aus Zeile 03 Folgendes:

```
03     clock_t stamp;
```

Zum Nachlesen

Die neue Funktionssyntax und `decltype` werden kurz in Abschnitt 5.9, »Die neue Funktionssyntax (C++11)«, erläutert. Das Schlüsselwort `auto` hingegen wird in Abschnitt 2.5, »Typ `auto` (C++11)«, beschrieben.

7.8 Spezielle Eigenschaften einer Klasse

Bisher waren die Eigenschaften einer Klasse immer recht einfach zu verwenden. Es gibt aber auch spezielle Eigenschaften, welche man verwenden kann. Da wären noch die Möglichkeiten konstante, dynamische oder statische Eigenschaften zu verwenden. Alle drei Dinge müssen speziell behandelt werden, weshalb sich dieser Abschnitt damit befassen wird.

7.8.1 Konstante Elemente in einer Klasse

Wenn Sie in Ihrer Klasse konstante Eigenschaften (`const`) implementieren, ist eine direkte Zuweisung mit dem `=`-Operator an eine Konstante nicht mehr möglich. In solch einem Fall müssen Sie die Initialisierungsliste des Konstruktors verwenden.

Hierzu unsere Klassendefinition `Personal` mit einer neuen konstanten Eigenschaft:

```
00 // kapitel07/7_8_1/personal.h
...
01 class Personal {
02     std::string name;           // Name
03     unsigned int pers_nr;      // Personalnummer
04     unsigned int gehalt;       // monatliches Gehalt
05     const std::time_t datum;   // Einstellungsdatum;
06 public:
07     Personal();
08     Personal(std::string n, unsigned int pn,
09             unsigned int g, std::time_t z);
09     Personal(std::string n, unsigned int pn);
10     Personal(std::string n);
...
11     std::string get_datum() const;
...
12 };
```

Hier wurde in Zeile 05 eine konstante Eigenschaft `datum` vom Typ `std::time_t` aus der Headerdatei `<ctime>` hinzugefügt. `std::time_t` ist ein alter fundamentaler Typ,

welcher fähig ist, eine Zeit darzustellen, die von der Funktion `std::time()` zurückgegeben wird.

Im Konstruktor in Zeile 08 bieten wir auch noch ein Schnittstelle an, um `std::time_t` beim Erzeugen des Objekts manuell zu setzen. Natürlich bieten wir auch eine Getter-Zugriffsfunktion in Zeile 11 an, um das Einstellungsdatum als `std::string` zurückzugeben.

Ein `set_datum(std::time_t)` als Methode würde bei einer konstanten Eigenschaft nicht mehr funktionieren, weil ein nachträgliches Ändern nicht mehr möglich ist – und hier auch gar nicht erwünscht wäre.

time_t für einen Zeitstempel (Timestamp)?

Für portable Programme sollten Sie sich nicht auf `std::time_t` und die Funktion `std::time()` als Zeitstempel verlassen, weil der Standard nicht vorschreibt, dass `time_t(0)` gleich der 01.01.1970 um 00:00 Uhr darstellt. Aber für unser Beispiel zu Demonstrationszwecken ist es völlig ausreichend. Überhaupt ist die Verwendung von `time_t` hier nur Beiwerk. Es geht hier viel mehr um das Verständnis von konstanten Eigenschaften.

Wie bereits erwähnt, müssen Sie die Initialisierung von konstanten Eigenschaften in der Initialisierungsliste des oder der Konstruktoren durchführen – auch des Standardkonstruktors. Hier die Implementierung der konstanten Eigenschaft in unsere Quelldatei *personal.cpp*:

```
00 // kapitel07/7_8_1/personal.cpp
...
01 Personal::Personal() : name(""), pers_nr(0),
                        gehalt(GG), datum(std::time(0)) { }

02 Personal::Personal( std::string n, unsigned int pn,
                        unsigned int g, std::time_t t ) :
                        name(n), pers_nr(pn), gehalt(g), datum(t) { }

03 Personal::Personal(std::string n, unsigned int pn) :
                        name(n), pers_nr(pn), gehalt(GG), datum(std::time(0)) { }

04 Personal::Personal(std::string n) :
                        name(n), pers_nr(0), gehalt(GG), datum(std::time(0)) { }

05 void Personal::print_data() {
06     std::cout << "Name      : " << get_name() << std::endl;
07     std::cout << "Pers.Nr.: " << get_pers_nr() << std::endl;
```

```

08  std::cout << "Gehalt  : " << get_gehalt() << std::endl;
09  std::cout << "Datum   : " << get_datum() << "\n\n";
10  }
...
11  std::string Personal::get_datum() const {
12      return std::string( std::ctime(&datum) );
13  }
...

```

Sie sehen, dass in allen Konstruktoren die Initialisierungsliste erweitert wurde. Zusätzlich wurde hier auch noch in der Methode zur Ausgabe aller Eigenschaften in Zeile 09 die Methode `get_datum()` hinzugefügt, deren Definition Sie in den Zeilen 11 bis 13 vorfinden. Der Rückgabewert von `ctime(&datum)` liefert einen C-Style-String (`char*`) des »lesbaren« Einstellungsdatums zurück, welches Sie in den Konstruktoren mit `time_t` gesetzt haben. Mit `std::string()` konvertieren wir diesen C-Style-String in einen ordentlichen C++-String und geben diesen zurück.

Die Initialisierung von konstanten Eigenschaften muss allerdings in der Initialisierungsliste erfolgen und darf nicht im Anweisungsblock wie bspw. folgt geschehen:

```

Personal::Personal(std::string n) : name(n){
    pers_nr = 0;
    gehalt = GG;
    datum = std::time(0); // !!! Fehler !!!
}

```

Dieser Codeausschnitt funktioniert somit nur, wenn Sie die Initialisierung der konstanten Eigenschaft oben in die Initialisierungsliste stecken:

```

Personal::Personal(std::string n) :
    name(n), datum(std::time(0)) {

    pers_nr = 0;
    gehalt = GG;
}

```

Hierzu noch ein kurzes Beispiel, welches Sie zum Testen der neuen konstanten Eigenschaft verwenden können:

```

00  // kapitel07/7_8_1/main.cpp
...
01  Personal person01("Franz Mustersohn", 400455);
02  Personal person02("Maxi Musterfrau", 400567);

```



```

03 Personal person3("Max Mustermann",400345,4000, time(0));

04 std::cout << person01.get_datum() << std::endl;
05 person02.print_data();
06 person03.print_data();

```

Das Programm bei der Ausführung:

Thu Oct 14 17:38:25 2013

```

Name      : Maxi Musterfrau
Pers.Nr.: 400567
Gehalt    : 1900
Datum     : Thu Oct 14 17:38:25 2013

```

```

Name      : Max Mustermann
Pers.Nr.: 400345
Gehalt    : 4000
Datum     : Thu Oct 14 17:38:25 2013

```

Direkte Initialisierung in C++11

C++11 macht einem das Leben hier wesentlich einfacher, weil eine direkte Initialisierung der Eigenschaften in der Klassendefinition möglich ist. Sie können sich also ein wenig Tipparbeit sparen, wenn Sie das Einstellungsdatum gleich übergeben:

```

class Personal {
    std::string name;      // Name
    unsigned int pers_nr;  // Personalnummer
    unsigned int gehalt;    // monatliches Gehalt
    const std::time_t datum{std::time(0)}; // C++11-like
public:
    ...
};

```

7.8.2 Rohrer Zeiger als Element in einer Klasse

Enthält eine Klasse hingegen rohe Zeiger als Eigenschaften, steigt auch der Aufwand der Verwaltung solcher Eigenschaften von Klassen. Daher sollten Sie schon bei Erstellung von Klassen darauf achten, einfachere und sicherere Alternativen zu finden. In der Vergangenheit wurden rohe Zeiger in Klassen gerne mal für dynamische Eigenschaften verwendet. Im modernen C++ bieten sich heutzutage genügend Alternativen in Form von Standard-Containerklassen wie `std::string` oder `std::vector` an.

Leider hat man es nicht immer selber in der Hand, welchen Code man vor sich hat oder nacharbeiten muss. Auf diesen Fall will ich Sie hier vorbereiten.

Hierzu soll eine einfache Klasse `Zeitarbeiter` erstellt werden, welche auch unsere Klasse `Personal` verwendet, wo außerdem auch der Knappheit und Übersicht zuliebe alles gleich in der Klassendefinition `inline` definiert wird. Hierzu zunächst das Stückchen Quellcode, um unser Problem mit rohen Zeigern als Eigenschaften zu demonstrieren:

```
00 // kapitel07/7_8_2/zeitarbeiter.h
...
01 #include "personal.h"

02 class Zeitarbeiter {
03     char* firma;
04     Personal person;
05 public:
06     Zeitarbeiter(const char* f, const Personal p):person(p){
07         firma = new char[std::strlen(f) + 1];
08         std::strcpy(firma, f); // f nach firma kopieren
09     }
10     ~Zeitarbeiter() {
11         delete[] firma;
12     }
13     void print_All() {
14         std::cout << "Verliehen an : " << firma << std::endl;
15         person.print_data();
16     }
17 };
```

Der »Problemfall« in dieser Klasse ist die Eigenschaft `firma` in Zeile **03**, welche als `char*` implementiert wurde. Ganz klar, in der echten C++-Praxis verwendet man stattdessen `std::string`, und das Problem ist gelöst. Wenn Sie den Code betrachten, dürften Sie sich wohl die Frage stellen, wo hier das Problem ist. Es wird Speicher vom Heap reserviert, wenn Sie diesen benötigen, und dieser wird wieder über den Destruktor in den Zeilen **10** bis **12** freigegeben, wenn er nicht mehr benötigt wird. Hier wurde im Prinzip sauber nach dem RAII-Prinzip gearbeitet.

RAII

RAII steht für *resource acquisition is initialization* (= Ressourcenbelegung ist Initialisierung) und bezeichnet eine Programmiertechnik, welche die Belegung und Freigabe einer Ressource an den Gültigkeitsbereich heftet.

In C++ ist dies bspw. gegeben, wenn eine Ressource mit dem Konstruktor belegt wird und automatisch wieder mit dem Destruktor freigegeben wird, sobald der Gültigkeitsbereich verlassen wird. Die Freigabe von Ressourcen im Destruktor ist sehr wichtig, weil dieser auch aufgerufen wird, wenn eine Ausnahme geworfen wird. Ein weiterer typischer Anwendungsfall von RAII ist die Steuerung von Prozess- bzw. Thread-Sperren oder Datei-Operationen.

Und in der Tat ist es kein Problem, wenn Sie diese Klasse in der Praxis folgendermaßen einsetzen:

```
01 {
...
02   Personal person01("Max Mustermann", 400333);
03   Zeitarbeiter zeitarbeiter01("Elementaria Inc.", person01);
04   zeitarbeiter01.print_All();
05 }
```

In Zeile **02** wird ein Objekt mit dem Bezeichner `zeitarbeiter01` der Klasse `Zeitarbeiter` erzeugt und in Zeile **03** ausgegeben. Am Ende des Gültigkeitsbereichs hinter Zeile **05** wird auch der Destruktor aufgerufen und gibt den reservierten Speicher für die dynamische Eigenschaft `firma` wieder frei.

Zu Problemen kommt es erst, wenn Sie eine flache Kopie wie folgt erstellen:

```
00 // kapitel07/7_8_2/main.cpp
...
01   Personal person01("Max Mustermann", 400333);
02   Zeitarbeiter zeitarbeiter01("Elementaria Inc.", person01);
03   {
04       Zeitarbeiter zeitarbeiter02(zeitarbeiter01);
05       zeitarbeiter02.print_All();
06   }
07   zeitarbeiter01.print_All();
```

In Zeile **04** wird der Standard-Kopierkonstruktor verwendet, um den Inhalt von `zeitarbeiter01` in `zeitarbeiter02` zu kopieren. Bei dieser Kopie spricht man von einer **flachen Kopie** (engl. *shallow copy*), weil die Kopie des Objekts Daten mit dem anderen Objekt teilt. Während das Objekt `person` mit der Klasse `Personal` noch ordentlich kopiert wird, verwenden beide Objekte `zeitarbeiter01` und `zeitarbeiter02` den Zeiger `firma` auf dieselbe Adresse. Ein Problem tritt jetzt auf, wenn das Objekt aus Zeile **04** ab Zeile **06** seinen Gültigkeitsbereich verlässt und somit der Destruktor aufgerufen wird, wodurch die gemeinsame Ressource `firma` zerstört wird. Die weitere Verwendung von `zeitarbeiter01` wie in Zeile **07** ist jetzt undefiniert, weil die »gemeinsame«

Eigenschaft `firma` offiziell zerstört wurde. Abbildung 7.6 soll diesen Sachverhalt bildlich darstellen.

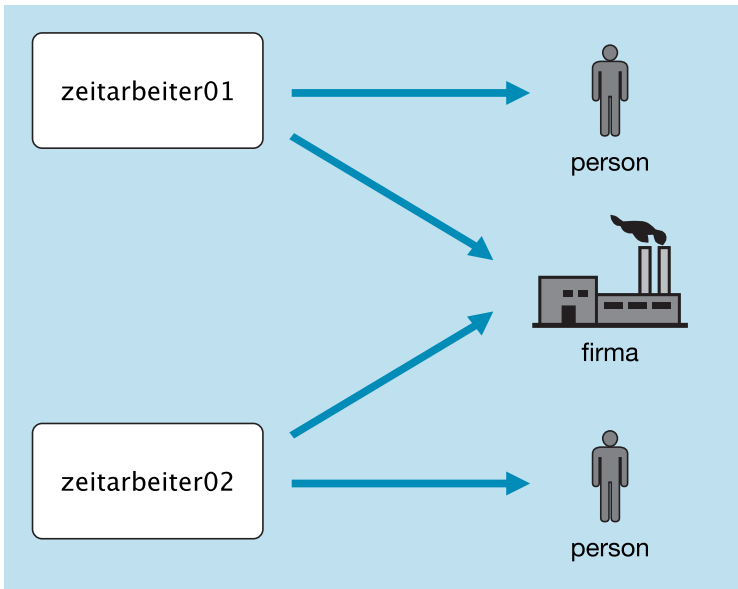


Abbildung 7.6 Das Problem bei einer flachen Kopie mit den Standardsachen ist, dass beide Objekte dieselbe Ressource teilen. Gefährlich wird dies dann, wenn eines der beiden Objekte zerstört wird und das andere Objekt auf einen nicht mehr gültigen Adressbereich zugreift.

Wie schon einmal im Buch bei den großen Drei (*The Big Three*) erwähnt, sollten Sie, wenn Sie mindestens einen der Konstruktoren selbst implementieren (also **Kopierkonstruktor**, **Zuweisungsoperator** oder **Desktruktor**), auch die anderen beiden hinzufügen und sich nicht mehr mit der vom Compiler zur Verfügung gestellten Version zufriedengeben. Denn wenn Sie einen der Konstruktoren selbst implementieren müssen, bedeutet dies ja auch, dass die Standardversionen nicht mehr ausreichend sind.

In unserem Beispiel müssen Sie daher den Kopierkonstruktor und den Zuweisungsoperator nachimplementieren, um hier die Eigenschaften manuell, Eigenschaft für Eigenschaft, zu kopieren. Hier spricht man dann von einer tiefen Kopie (engl. *deep copy*).

Hierzu wird nun der Quellcode von `zeitarbeiter.h` von oben um den Kopierkonstruktor und den Zuweisungsoperator erweitert, um eine tiefe Kopie zu erzeugen:

```
00 // kapitel07/7_8_2/zeitarbeiter.h
...
01 class Zeitarbeiter {
```

```

02  char* firma;
03  Personal person;
04  public:
05  Zeitarbeiter(const char* f, const Personal p):person(p) {
06      firma = new char[std::strlen(f) + 1];
07      std::strcpy(firma, f); // f nach firma kopieren
08  }

09  Zeitarbeiter(const Zeitarbeiter &za) {
10      firma = new char[std::strlen(za.firma) + 1];
11      std::strcpy(firma, za.firma);
12      person = za.person;
13  }

14  Zeitarbeiter& operator=(const Zeitarbeiter& za) {
15      if(this == &za) {
16          return *this;
17      }
18      firma = new char[std::strlen(za.firma) + 1];
19      std::strcpy(firma, za.firma);
20      person = za.person;
21      return *this;
22  }

23  ~Zeitarbeiter() {
24      delete[] firma;
25  }
...
26  };

```

Die Definition des selbst erstellten Kopierkonstruktors finden Sie in den Zeilen 09 bis 13. Zunächst fordern wir hier in Zeile 10 Speicherplatz für eine Kopie von `za.firma` an. Die Anzahl der Zeichen lassen wir uns von der alten C-Style-Funktion `std::strlen()` zählen. Das `+ 1` ist notwendig, weil das String-Endezeichen bei C-Style-Strings auch behandelt werden muss. Der Zeiger `firma` (genauer `this->firma`) verweist auf die Anfangsadresse des neu reservierten Speicherbereichs. Mit der Uralt-Funktion `std::strcpy()` in Zeile 11 kopieren Sie jetzt den kompletten Inhalt von `za.firma` nach `firma`. Jetzt haben Sie eine tiefe Kopie erstellt. Beide Objekte haben jetzt eine eigene Eigenschaft `firma` mit jeweils unterschiedlichen Adressen. Das Objekt `za.person` hingegen lässt sich fast problemlos nach `person` per Zuweisung kopieren (siehe Kasten), wie dies in Zeile 12 geschieht.

Optimierung des Kopierkonstruktors mit Element-Initialisierer

Wenn Sie den Kopierkonstruktor betrachten, sehen Sie hier ein (Teil-)Objekt `person`, wofür Sie den Element-Initialisierer verwenden sollten (siehe dazu den Abschnitt »Konstruktoren effektiver initialisieren mit Element-Initialisierer« in Abschnitt 7.4.2). Auch die Speicherreservierung können Sie per Element-Initialisierer erledigen. Optimierte sieht der Kopierkonstruktor dann wie folgt aus:

```
Zeitarbeiter(const Zeitarbeiter &za) :
    firma(new char[std::strlen(za.firma)+1]), person(za.person) {
        std::strcpy(firma, za.firma);
    }
```

Die Operatoren-Überladung ist erst Thema in Kapitel 8, »Operatoren überladen«, aber trotzdem will ich Ihnen das Beispiel hierzu nicht vorenthalten. In den Zeilen 15 bis 17 prüfen wir zunächst auf eine Selbstzuweisung, wie bspw. `zeitarbeiter01=zeitarbeiter01`, um ein Speicherleck zu verhindern. Bei einer Selbstzuweisung müssen wir keinen neuen Speicher anfordern (der niemals mehr freigegeben würde), sondern liefern einfach wieder das komplette Objekt mit `*this` zurück. Ansonsten läuft alles exakt wie schon beim Kopierkonstruktor ab, nur dass am Ende in Zeile 21 das neu initialisierte Objekt als Ganzes via `*this` zurückgegeben wird.

Zusätzlich sollte hier noch angemerkt werden, dass beim `operator=()` das `this`-Objekt schon existiert. Es handelt sich hierbei also nicht um einen Konstruktor, und daher muss man sich auch um den alten Zustand der Eigenschaften seine Gedanken machen. Mehr zum `operator=()` erfahren Sie in Abschnitt 8.2, »Zuweisungsoperator – `operator=`«.

Tiefe Kopie und const-Eigenschaften

Wenn Sie in diesem Beispiel die `const`-Eigenschaft `Personal::datum` aus Abschnitt 7.8.1 verwendet haben und eine Zuweisung im Kopierkonstruktor bzw. Zuweisungsoperator wie `person=za.person`; gemacht haben, würde dies nicht funktionieren, weil ja eine Zuweisung von `const`-Eigenschaften nicht möglich ist. Um eine weitere Codeanpassung in der Klasse `Personal` vorzunehmen, wurde hier der Einfachheit halber auf diese konstante Eigenschaft wieder verzichtet.

Ich denke, Sie haben in diesem Abschnitt selbst gesehen, dass rohe Zeiger als Eigenschaften einen ziemlichen Aufwand bedeuten können. Den können Sie sich auf jeden Fall sparen, wenn Sie das Klassendesign selbst entwerfen, indem Sie erst gar keinen rohen Zeiger dafür verwenden, wenn es möglich ist. Wenn Sie allerdings an ein Projekt geraten, in dem rohe Zeiger nun mal enthalten sind, wissen Sie, worauf es hierbei

ankommt. Besonders wichtig ist es, dass Sie zwischen einer flachen und einer tiefen Kopie unterscheiden können. Leider wurden hier nicht alle Feinheiten im Beispiel demonstriert. Daher nochmals eine kurze Checkliste, worauf Sie auf jeden achten sollten, wenn Sie dynamische Eigenschaften in einer Klasse verwenden wollen/müssen:

- ▶ Wenn Sie dynamische Eigenschaften verwenden, benötigen Sie häufig eine weitere Eigenschaft, welche die Anzahl bzw. Länge des oder der dynamischen Eigenschaft(en) enthält.
- ▶ Beim Standardkonstruktor sollten Sie dafür sorgen, dass gleich zu Beginn mindestens Platz für ein Element zu Verfügung steht – was auch zur Laufzeit mittels `new` geschehen muss.
- ▶ Häufig benötigen Sie eine weitere Methode, um bei Bedarf zur Laufzeit des Programms stets Speicherplatz für weitere Elemente reservieren zu können.
- ▶ Sie benötigen einen eigenen Kopierkonstruktor und einen eigenen Zuweisungsoperator, weil sonst zwei Objekte beim Kopieren mit derselben Adresse arbeiten würden. Sobald eines der Objekte zerstört wird, ist diese gemeinsame Adresse nicht mehr gültig und ein Zugriff auf diese Eigenschaft vom noch vorhandenen Objekt ist undefiniert.
- ▶ Wenn Sie den Speicher klassisch mit `new` bzw. `new[]` anfordern, müssen Sie diesen im Destruktor auch ordentlich mit `delete` bzw. `delete[]` wieder freigeben, um Speicherlecks (engl. Memory Leaks) zu vermeiden.

Verschieben implementieren

Wer jetzt noch C++11-mäßig den Verschiebe-Kopierkonstruktor implementieren will, den will ich hier auf Abschnitt 7.10, »Klassen für das Verschieben schreiben (C++11)«, verweisen.

7.8.3 Statische Eigenschaften in einer Klasse

Benötigen Sie Eigenschaften in einer Klasse, welche auch den anderen Objekten zur Verfügung stehen sollen – also eine gemeinsame Eigenschaft –, dann könnten Sie statische Eigenschaften in einer Klasse verwenden. Wenn Sie eine Eigenschaft in einer Klasse mit dem Schlüsselwort `static` kennzeichnen, ist diese Eigenschaft nur einmal im Speicher vorhanden

Solche Eigenschaften können hilfreich sein, wenn Sie Daten benötigen, die für alle Objekte von Interesse sein könnten. Dies können temporäre Werte sein, welche Sie dort zwischenspeichern, damit andere Objekte ebenfalls mit dieser Eigenschaft arbeiten können. Mithilfe solcher als `static` deklarierten Eigenschaften können Sie auf globale Speicherobjekte verzichten.

Statische Eigenschaften gehören nicht zum Objekt

Es ist daher sehr wichtig, dass Sie wissen, dass eine statische Eigenschaft nicht wirklich zum Objekt gehört und auch **nicht** über einen `this`-Zeiger erreichbar ist.

In unserem Beispiel mit der Klasse `Personal` könnten Sie bspw. einen Personalzähler implementieren, welcher sich mit jedem neu erzeugten Objekt bei allen Konstrukto-
ren um eins erhöht und mit jedem zerstörten Objekt beim Konstruktor wieder um eins reduziert. Hier ein Codeausschnitt, wo die statische Eigenschaft in der Klassen-
definition verwendet wird:

```
00 // kapitel07/7_8_3/personal.h
...
01 class Personal {
02     std::string name;      // Name
03     unsigned int pers_nr;  // Personalnummer
04     unsigned int gehalt;   // monatliches Gehalt
05     static unsigned int pers_counter;
06 public:
07     Personal();
08     Personal(std::string n, unsigned int pn, unsigned int g);
09     Personal(std::string n, unsigned int pn);
10     Personal(std::string n);
11     Personal(const Personal& p) = delete;
12     Personal& operator=(const Personal& p) = delete;
13     ~Personal();
...
14     unsigned int get_pers_counter() const;
15 };
```

In Zeile **05** finden Sie die statische Eigenschaft der Klasse `Personal`, welche jedem Objekt zur Verfügung steht. Wir verwenden diese Eigenschaft, um die Anzahl des Personals zu zählen. Daher muss in jedem Konstruktor dieser Wert um eins inkrementiert werden. Da eine Kopie des Personals eigentlich nicht wirklich Sinn macht und wir hier keine Lust haben, den Kopierkonstruktor und Zuweisungsoperator zu implementieren (und gegebenenfalls ebenfalls die statische Eigenschaft inkrementieren müssten), wurden diese beiden in den Zeilen **11** und **12** mit `delete` entfernt. Dies funktioniert allerdings nur mit C++11. In C++03 können sie diese beiden Zeilen gegebenenfalls auch in den privaten Bereich verschieben. Zusätzlich haben wir hier in Zeile **14** auch gleich noch eine Methode deklariert, um die Anzahl des Personals auch zurückgeben zu können.

Jetzt soll der Code geschrieben werden, in dem die statische Eigenschaft in den entsprechenden Konstruktoren inkrementiert und im Destruktor dekrementiert werden soll. Hier die Definition in *personal.cpp*:

```
00 // kapitel07/7_8_3/personal.cpp
...
01 #include "personal.h"

02 unsigned int Personal::pers_counter = 0;

03 Personal::Personal() : name(""), pers_nr(0), gehalt(GG) {
04     ++pers_counter;
05 }

06 Personal::Personal(std::string n, unsigned int pn,
07     unsigned int g) : name(n), pers_nr(pn), gehalt(g) {
08     ++pers_counter;

09 Personal::Personal(std::string n, unsigned int pn) :
10     name(n), pers_nr(pn), gehalt(GG) {
11     ++pers_counter;
12 }

12 Personal::Personal(std::string n) :
13     name(n), pers_nr(0), gehalt(GG) {
14     ++pers_counter;
15 }

15 Personal::~Personal() { --pers_counter; }
...
16 unsigned int Personal::get_pers_counter() const {
17     return pers_counter;
18 }
```

Hierbei müssen Sie unbedingt beachten, dass die statische Eigenschaft für alle Objekte nur einmal vorhanden ist. Daher müssen Sie diese explizit, wie in Zeile **02** zu sehen ist, außerhalb der Klasse definieren. Die Definition erfolgt wie bei einer gewöhnlichen Variablen, nur zusätzlich mit dem Bezug (Scope-Operator) zur Klasse und ohne (!) das Schlüsselwort `static`. Der restliche Code dürfte für sich selbst sprechen. Hierzu noch ein kurzer Codeausschnitt, welcher die statische Eigenschaft in der Praxis demonstriert:

```

00 // kapitel07/7_8_3/main.cpp
...
01 Personal person01("Max Mustermann", 400333);
02 Personal person02;
03 {
04     Personal person03("Maxi Mustersohn", 400345);
05     std::cout << person02.get_pers_counter()
                << std::endl; // =3
06 }
07 std::cout << person02.get_pers_counter()
            << std::endl; // =2
08 std::vector<Personal> timeworker(10);
09 std::cout << person02.get_pers_counter()
            << std::endl; // =12
...

```

Bei der Ausgabe in Zeile 03 werden, gemäß der Objekte, die erzeugt wurden, drei ausgegeben. In Zeile 07 hingegen sind es wieder nur noch zwei Objekte, weil `person03` von Zeile 04 hier nicht mehr gültig ist und vom Destruktor zerstört wurde. Natürlich funktioniert das auch mit ganzen Vektoren von Objekten, wie die Ausgabe in Zeile 09 demonstriert, wo in Zeile 08 zuvor noch zehn weitere Objekte erzeugt wurden.

7.8.4 Statische Methoden

In der Praxis ist allerdings die Objektbezogenheit der Eigenschaft `Personal::pers_counter`, wie wir diese hier im Beispiel in Listing *kapitel07/7_8_3/main.cpp* zuvor verwendet haben, doch etwas praxisfremd. Ein Zugriff wie `objekt.get_pers_counter()` macht doch recht wenig Sinn, weil die statische Eigenschaft ja im Grunde ohnehin nicht direkt zur Klasse gehört. Und gerade wenn man diese Informationen abrufen will, dann will man es nicht über ein Objekt tun, sondern ein wenig allgemeiner. Für solche Zwecke würde sich an dieser Stelle eine statische Methode eignen. In unserem Beispiel reicht es schon aus, wenn Sie in der Klassendefinition die Deklaration der Methode `Personal::get_pers_counter()` mit `static` kennzeichnen. Bei der Definition außerhalb der Klassendefinition darf das Schlüsselwort `static` nicht mehr verwendet werden. Hier der Codeausschnitt von *personal.h* mit der `static`-Methode:

```

00 // kapitel07/7_8_4/personal.h
...
01 class Personal {
02     std::string name;        // Name
03     unsigned int pers_nr;    // Personalnummer
04     unsigned int gehalt;     // monatliches Gehalt
05     static unsigned int pers_counter;
06 public:

```

```
...
07  static unsigned int get_pers_counter();
08  };
```

Hier wurde in Zeile 07 lediglich das Schlüsselwort `static` vor `get_pers_counter()` gesetzt. `const` müssen Sie allerdings bei diesen Methoden entfernen (auch in der Definition), weil dies mit `static`-Methoden nicht erlaubt ist. Da Sie jetzt eine statische Methode daraus gemacht haben, können Sie diese Methode jetzt auch ohne Bezug zu einem Objekt mit `Personal::get_pers_counter()` aufrufen. Natürlich funktioniert das nach wie vor in Verbindung mit einem Objekt. Hier ein Stückchen des Codes, der dies demonstrieren soll:

```
00  // kapitel07/7_8_4/main.cpp
...
01  Personal person01("Max Mustermann", 400333);
02  Personal person02;
03  {
04      Personal person03("Maxi Mustersohn", 400345);
05      std::cout << Personal::get_pers_counter() << std::endl; // =3
06  }
07  std::cout << person02.get_pers_counter() << std::endl; // =2
08  std::vector<Personal> timeworker(10);
09  std::cout << Personal::get_pers_counter() << std::endl; // =12
...
```

In den Zeilen 05 und 09 sehen Sie jetzt die Verwendung der statischen Methode ohne Bezug zu einem Objekt. Allerdings funktioniert es nach wie vor in Verbindung mit einem Objekt, wie Zeile 07 zeigt.

Neben der Einschränkung, dass statische Methoden nicht `const` (und auch nicht `volatile`) sein dürfen, gibt es noch weitere Einschränkungen für statische Methoden, die Sie kennen sollten:

- ▶ Statische Methoden enthalten keinen `this`-Zeiger.
- ▶ Da kein `this`-Zeiger vorhanden ist, dürfte klar sein, dass Sie mithilfe von statischen Methoden nur auf statische Eigenschaften einer Klasse zugreifen können. Ein Zugriff auf andere Eigenschaften wie `name`, `pers_nr` oder `gehalt` ist hiermit nicht möglich.

7.8.5 Andere Klassen als Eigenschaft in einer Klasse

Dass Sie Klassen als Eigenschaften innerhalb anderer Klassen verwenden können, kennen Sie mit `std::string` jetzt schon zur Genüge und auch von unserer eigenen Klasse `Personal`, wo wir diese als Eigenschaft bzw. (Teil-)Objekt in der Klasse `Zeitar-`

beiter implementiert haben. Im Grunde ist die Verwendung von Klassen als Eigenschaften nicht wesentlich komplexer als die von gewöhnlichen Basisdatentypen. Wichtig ist hierbei allerdings, dass Sie Klassen als Eigenschaften oder hier auch den (Teil-)Objekten beim Konstruktor ihren Wert über den Element-Initialisierer übergeben und nicht erst im Anweisungsblock. Darüber wurde bereits im Abschnitt »Konstruktoren effektiver initialisieren mit Element-Initialisierer«, in Abschnitt 7.4.2 ausführlich geschrieben.

7.8.6 constexpr bei Klassen (Objekte zur Kompilierzeit) (C++11)

Sie haben `constexpr` bereits im Abschnitt »constexpr, das stärkere `const`« in Abschnitt 6.4.3 näher kennengelernt. Gerade im Embedded-Bereich, wo man Daten gerne in den ROM-Speicher legt, kann ein `constexpr` sehr hilfreich sein. Und beim Multithreading muss man sich keine Gedanken mehr machen, ob die Daten multithreadingfest sind, weil diese schon zur Übersetzungszeit berechnet werden und somit endgültig vorliegen. Natürlich werden die Daten nicht unbedingt in das ROM gelegt, aber es kann doch sehr hilfreich sein, wenn alles bereits zur Übersetzungszeit initialisiert und fertig zur Verwendung vorliegt. Wenn gewissen Regeln eingehalten werden, können Sie mit `constexpr` auch fertige Instanzen von Klassen zur Übersetzungszeit fest in das Kompilat einbauen.

Für die Eigenschaften müssen Sie allerdings literale Typen verwenden. Sie können bspw. keinen `std::string` als Eigenschaft verwenden, weil dieser Typ keinen trivialen Destruktor enthält und dessen Eigenschaft auch nichtliterale Typen enthält. Bei den `constexpr`-Klassen müssen sowohl Kopierkonstruktor als auch Destruktor trivial sein – Sie müssen also einen leeren Anweisungsblock haben. Eine weitere Voraussetzung ist, dass mindestens ein Konstruktor mit `constexpr` spezifiziert wurde.

Literaler Typ

Ein Typ ist in diesem Fall literal, wenn alle Eigenschaften der Klasse (die Daten) ebenfalls literalen Typs sind und mindestens ein Konstruktor mit `constexpr` spezifiziert ist.

Zugegeben, es hört sich alles recht komplex an, und es sind schon eine Menge Dinge zu beachten, um ein Objekt schon zur Übersetzungszeit zur Verfügung zu haben, daher finden Sie hierzu einen kurzen und einfach gehaltenen Codeausschnitt, welcher demonstrieren soll, wie Sie ein Objekt schon zur Übersetzungszeit im Kompilat verwenden können:

```
00 // kapitel07/7_8_6/main.cpp
...
01 constexpr unsigned int MAX_X = 2550;
```

```

02 constexpr unsigned int MAX_Y = 1880;

03 constexpr unsigned int check_size_x(unsigned int s) {
04     return s < MAX_X ? s : MAX_X;
05 }

06 constexpr unsigned int check_size_y(unsigned int s) {
07     return s < MAX_Y ? s : MAX_Y;
08 }

09 class Window {
10     unsigned int size_x, size_y;
11     //std::string name = "test"; //Geht nicht! Nicht trivial
12 public:
13     constexpr Window(unsigned int x, unsigned int y) :
14         size_x(check_size_x(x)), size_y(check_size_y(y)) {}
15     constexpr unsigned int pixelsize() {
16         return check_size_x(size_x)*check_size_y(size_y);
17     }
18     constexpr unsigned int get_size_x() {return size_x;}
19     constexpr unsigned int get_size_y() {return size_y;}
20 };

21 int main() {
22     constexpr Window win01{5000, 5000};
23     constexpr unsigned int pixel = win01.pixelsize();
24     constexpr unsigned int y = win01.get_size_y();
25     constexpr unsigned int x = win01.get_size_x();
26     ...
27 }

```

Die Eigenschaften von `Window` sind hier zwei `unsigned int` (Zeile 10) und somit trivial genug. Mit einem Typ wie `std::string` (Zeile 11) würde es nicht mehr funktionieren. Auch mindestens einen Konstruktor mit `constexpr` haben wir spezifiziert und in Zeile 13 auch gleich an Ort und Stelle definiert. Der Konstruktor ist auch trivial und besitzt einen leeren Anweisungsblock. Die Eigenschaften füllen wir mit dem Element-Initialisierer hinter dem Doppelpunkt aus.

Auch die Methode `pixelsize()` ist `constexpr` und verwendet gleichfalls auch die `constexpr`-Funktionen `check_size_x()` und `check_size_y()`, um gegebenenfalls eine zu große Fenstergröße auf den maximal erlaubten Wert zurückzusetzen. Mehr ist im Grunde nicht nötig, um Objekte wie in Zeile 21 oder Methoden wie in Zeile 22 bereits zur Übersetzungszeit zur Verfügung zu haben.

Der Vorteil an dieser `constexpr`-Verwendung ist auch, dass der Compiler jetzt sicherstellt, dass `constexpr` eingehalten wird. Das beinhaltet auch Funktionen und Daten, welche nur deklariert, aber nicht definiert wurden.

constexpr – Übersicht

Sie haben `constexpr` jetzt über mehrere Kapitel verstreut in Verbindung mit Variablen, Funktionen und Klassen kennengelernt. Daher soll hier nochmals grob zusammengefasst werden, welche Voraussetzungen nötig sind, um Variablen, Funktionen oder Objekte gleich zur Übersetzungszeit fertigzustellen:

- ▶ **constexpr-Variablen und Objekte:** Einer `constexpr`-Variablen muss sofort ein Wert zugewiesen werden. Eigenschaften von Klassen, in denen ein `constexpr`-Konstruktor vorhanden ist, müssen daher ebenfalls sofort mit Werten instanziiert werden. Als Parameter für den Konstruktor oder als zugewiesener Wert für Variablen dürfen nur solche von literalem Typ, andere `constexpr`-Variablen oder `constexpr`-Funktionen verwendet werden. Der explizite Konstruktor, der das Objekt konstruiert, muss mit `constexpr` spezifiziert sein.
- ▶ **constexpr-Funktionen und Methoden:** Die Methode darf nicht `virtual` sein (`virtual` wird später im Buch erklärt). Der Rückgabewert muss literalen Typs sein, wie auch die einzelnen Parameter. Der Anweisungsblock kann entweder `deleted` (siehe Abschnitt 7.9.1, »default«) oder `defaulted` (siehe Abschnitt 7.9.2, »delete«) sein oder darf Folgendes enthalten:
 - ▶ eine `return`-Anweisung, welche nur literale Werte, `constexpr`-Variablen oder `constexpr`-Funktionen enthält
 - ▶ `static_assert()`
 - ▶ `typedef`-Deklarationen
 - ▶ `using`-Deklarationen und `using`-Direktiven
- ▶ **constexpr-Konstruktor:** Die Klasse darf keine virtuellen Basisklassen haben (wird später behandelt). Auch hier muss jeder Parameter ein literaler Typ sein. Für den Anweisungsblock gilt dasselbe, das schon eben bei den `constexpr`-Funktionen und Methoden erwähnt wurde, nur ohne die `return`-Anweisung. Für die Initialisierung der Eigenschaften wird die Initialisierungsliste (Element-Initialisierer) verwendet. Jeder Konstruktor muss logischerweise auch mit `constexpr` spezifiziert sein.

7.9 Erzeugen von Methoden steuern (C++11)

In C++11 haben Sie mit den neuen Schlüsselwörtern `delete` und `default` die Möglichkeit, einzelne Methoden per Definition zu verbieten oder den Compiler anzuweisen, eine Standardversion zu implementieren, ohne selbst die Implementierung zu erstellen.

7.9.1 default

Sie haben in diesem Buch jetzt schon des Öfteren von den großen Drei (The Big Three) gehört. Im neuen C++11-Standard sind ja mit den Verschiebeoperationen noch zwei weitere Versionen hinzugekommen. Daher wissen Sie, dass der Compiler Ihnen eine Reihe von Methoden automatisch zu Verfügung stellt, wenn diese im Programm irgendwo benötigt werden und Sie diese nicht selbst implementiert haben. Dies wären:

- ▶ der **Standardkonstruktor**, wenn kein eigener Konstruktor implementiert wurde
- ▶ der **Kopierkonstruktor** und der **Zuweisungsoperator**, wenn keine Verschiebeoperation implementiert wurde
- ▶ der **Verschiebekonstruktor** und der **Verschiebe-Zuweisungsoperator**, wenn keine Kopieroperation implementiert wurde
- ▶ der **Destruktor**

Default-Operatoren

Es wird eigentlich recht selten irgendwo erwähnt, aber hier sollte noch hinzugefügt werden, dass per Default auch diverse Operatoren von C++ zur Verfügung gestellt werden. Dazu gehören `operator&`, `operator*`, `operator->`, `operator->*`, `operator new` und `operator delete`.

Diese starke Verbindung zwischen der Standarderstellung der Konstruktoren bereitet allerdings auch Probleme. Sobald Sie einen eigenen Konstruktor implementiert haben, wird der Standardkonstruktor nicht mehr erzeugt. Ein einfaches Beispiel hierzu:

```
01 class SimpleClass {
02     unsigned int val{0};
03 public:
04     SimpleClass(unsigned int v) : val(v) {}
05 };
...
06 SimpleClass sc; // Fehler !!! Kein Standardkonstruktor
```

Der Versuch, das Objekt in Zeile **06** zu instanziierten, wird vom Compiler mit einer Fehlermeldung quittiert, weil in Zeile **04** bereits ein Konstruktor erzeugt wurde und somit kein Standardkonstruktor mehr generiert wird. Hier kommt jetzt das neue Schlüsselwort `default` ins Spiel, mit dem Sie den Compiler explizit anweisen können, den Standardkonstruktor selbst zu implementieren. Hierzu müssen Sie lediglich eine weitere Zeile hinzufügen:

```
01 class SimpleClass {
02     unsigned int val{0};
03 public:
```

```

04 SimpleClass(unsigned int v) : val(v) {}
05 SimpleClass() = default;
06 };
...
07 SimpleClass sc; // Standardkonstruktor ist jetzt vorhanden

```

Mit der Zuweisung von `=default` in Zeile 05 zum Standardkonstruktor erzeugt der Compiler jetzt explizit eine Standardversion, weshalb sich das Objekt in Zeile 07 jetzt instanziierten lässt.

Wie mit dem Standardkonstruktor funktioniert dies auch mit den anderen Konstruktoren, und spätestens wenn Sie die Kopier- und Verschiebekonstruktoren verwenden wollen, werden Sie den Vorteil von `=default` zu schätzen wissen. Wie bereits kurz erwähnt, werden vom Compiler keine Standard-Kopieroperationen erzeugt, wenn die Verschiebeoperationen implementiert wurden. Dasselbe gilt auch umgekehrt. Mit `=default` können Sie den Compiler explizit anweisen, trotzdem eine entsprechende Standardversion zu implementieren:

```

01 class SimpleClass {
02     unsigned int val{0};
03 public:
04     SimpleClass(unsigned int v) : val(v) {}
05     SimpleClass(const SimpleClass& sc) : val(sc.val) {}
06     SimpleClass(SimpleClass&& ) = default;
07     SimpleClass() = default;
08 };
...
09 SimpleClass sc(100);
10 SimpleClass xy = std::move(sc);
11 SimpleClass zw = xy;

```

Da hier in Zeile 05 ein eigener einfacher Kopierkonstruktor implementiert wurde, wird vom Compiler der Verschiebekonstruktor nicht mehr generiert, welcher in Zeile 10 verwendet wird. Mit einem einfachen `=default` in Zeile 06 weisen wir den Compiler an, den Verschiebekonstruktor explizit zu generieren und haben somit sowohl den Kopierkonstruktor als auch den Verschiebekonstruktor zur Verfügung.

Vom Compiler generierte Methoden in der Standardeinstellung

Die Standardversionen vom Compiler sind immer öffentlich erreichbar (`public`) und `inline` implementiert. Sie sind weder `explicit` noch als `virtual` generiert. Tatsächlich finden Sie hier noch weitere Vorteile, `=default` zu verwenden. So können Sie bspw. einen Standardkonstruktor (ohne Argumente) ganz einfach als `explicit` oder den Destruktor als `virtual` implementieren.

Die =default-Spezifikation muss übrigens nicht in der Klassendefinition erfolgen, sondern darf auch in der Implementation des Codes geschrieben werden. =default darf nur nicht in der Deklaration und Definition stehen. Folgendes ist bspw. ebenfalls möglich:

```
class SimpleClass {
    unsigned int val{0};
public:
    SimpleClass(unsigned int v) : val(v) {}
    SimpleClass(const SimpleClass& sc) : val(sc.val) {}
    SimpleClass(SimpleClass&& );
    SimpleClass();
};

// Implementationen
SimpleClass::SimpleClass(SimpleClass&& ) = default;
SimpleClass::SimpleClass() = default;
...
```

7.9.2 delete

Den Spezifizierer =delete haben Sie ja bereits des Öfteren im Buch verwendet. Dieser stellt im Grunde das Gegenstück zu =default dar. Mit delete können Sie quasi per Design bestimmte Methoden oder Funktionen verbieten. Im klassischen C++03 wurden solche Funktionen mit `privat` deklariert. Eine gängige Anwendung ist es, dass Kopieren von Klassen komplett zu verbieten. Ebenso lässt sich die =delete auch bei Methoden anwenden. Der folgende Codeausschnitt soll =delete etwas näher demonstrieren:

```
01 class SimpleClass {
02     unsigned int val{0};
03 public:
04     SimpleClass(unsigned int v) : val(v) {}
05     SimpleClass() = default;
06     SimpleClass(const SimpleClass&) = delete;
07     SimpleClass& operator=(const SimpleClass&) = delete;
08     SimpleClass(SimpleClass&& ) = default;
09     SimpleClass& operator=(SimpleClass&&) = default;

10     void set_val(unsigned int v) {val=v;}
11     void set_val(double) = delete;
12     void set_val(const unsigned int) const = delete;
13 };
```

```

...
14 SimpleClass sc{100};
15 SimpleClass xy = std::move(sc); // Ok
16 SimpleClass vw = xy; // Nicht erlaubt !!!
17 SimpleClass zw(sc); // Nicht erlaubt !!!
18 SimpleClass um{std::move(xy)}; // Ok
19 um.set_val(123u); // Ok
20 sc.set_val(123.123); // Nicht erlaubt !!!
21 const SimpleClass csc(200);
22 csc.set_val(123u); // Ohnehin nicht erlaubt !!!
...

```

In den Zeilen 06 und 07 wird das Kopieren und Zuweisen von Objekten der Klasse SimpleClass per Design verboten. Die Zeilen 16 und 17 werden daher vom Compiler mit der Fehlermeldung quittiert, dass diese Konstruktoren `deleted` sind und nicht verwendet werden können. Mit unserer Klasse ist daher nur das Verschieben von Objekten mit dem Verschiebekonstruktor und der Verschiebezuweisung erlaubt. Beide Funktionen fordern wir vom Compiler in den Zeilen 08 und 09 explizit mit `=default` an, so dass er hier die Standardversionen implementiert. In den Zeilen 15 und 18 sehen Sie die Verschiebeoperationen im Einsatz.

Natürlich können Sie auch Methoden verbieten, wie Sie dies in den Zeilen 11 und 12 sehen können. In Zeile 11 verbieten wir, der Überladung einer Zugriffsfunktion einen Wert mit `double` zu übergeben, weshalb die Anweisung in Zeile 20 mit einem Fehler des Compilers quittiert wird. In Zeile 12 hingegen verbieten wir, dass ein `const`-Objekt die Zugriffsfunktion `set_val()` verwendet, was ja ohnehin verboten ist. Die Ausführung in Zeile 22 wird daher ebenfalls vom Compiler mit der Fehlermeldung quittiert, dass diese Methode `deleted` ist. Der Vorteil der Version in Zeile 12 gegenüber keinerlei Aktion dürfte allerdings wohl die angenehmere und lesbare Fehlermeldung sein, welche der Compiler zurückgibt.

Implizite Konvertierung per Design verbieten

Ebenfalls sehr hilfreich kann `=delete` sein, wenn Sie eine implizite Konvertierung von Typen vermeiden wollen. Wollen Sie bspw., dass eine bestimmte Funktion nur mit einem bestimmten Typ aufgerufen werden kann, brauchen Sie lediglich diese Funktion mit dem Typ zu definieren und ein Funktions-Template zu schreiben, welches alle anderen Typen als Argument verbietet. Auch wenn Funktions-Templates erst in Abschnitt 11.1, »Funktions-Templates«, beschrieben werden, will ich Ihnen das Beispiel hierzu nicht vorenthalten:

```

...
01 void output(unsigned int val) {
02     std::cout << val << std::endl;
03 }

04 template <typename T>
05 void output(T val) = delete;
...
06 output(100u);    // Ok -> unsigned int
07 output(123);     // Fehler -> int
08 output(123.123); // Fehler -> double
09 output('A');     // Fehler -> char
...

```

Durch die Zuweisung von `delete` zum Funktions-Template in den Zeilen **04** bis **05** kann die Funktion `output()` nur noch mit `unsigned int` aufgerufen werden, wie in den Zeilen **01** bis **03** definiert wurde. Ohne das gelöschte Funktions-Template in den Zeilen **04** bis **05** hätten alle anderen Funktionsaufrufe in den Zeilen **06** bis **09** aufgrund der impliziten Konvertierung funktioniert. So aber meldet der Compiler die Funktionsaufrufe in den Zeilen **07** bis **09** als nicht erlaubte Aufrufe, weil diese Funktionsaufrufe mit `delete` gekennzeichnet sind.

7.10 Klassen für das Verschieben schreiben (C++11)

In Abschnitt 7.8.2, »Rohrer Zeiger als Element in einer Klasse«, haben Sie gesehen, wie Sie gefährliche flache Kopien verhindern, indem Sie den Kopierkonstruktor und den Zuweisungsoperator selbst implementieren, um so tiefe Kopien zu erzeugen. In der gängigen C++-Praxis dürften Sie selten noch in Verlegenheit kommen, rohe Zeiger als Eigenschaft einer Klasse verwenden zu müssen. Neben den neuen Smart Pointern können Sie auch auf die vielen Standardcontainer zurückgreifen.

Stilfrage

Auch bei den folgenden Beispielen implementieren wir den Code gleich in der Klassendefinition, um Platz zu sparen. In der Praxis lagern Sie diese Definitionen außerhalb der Klassendefinition in eine extra Quelldatei aus.

Kopieren verbieten

Wie dem auch sei, um hier einen eigenen Verschiebekonstruktor (Move-Konstruktor) und Verschiebe-Zuweisungsoperator (Move-Zuweisungsoperator) etwas sinnge-

mäßer demonstrieren zu können, verwende ich hierfür nochmals eine Klasse mit einem rohen Zeiger. Wir wollen dieses Mal die gefährlichen flachen Kopien einfach per Design via `=delete` verbieten. Hierzu der Ausgangscode:

```

00 // kapitel07/7_10a/move01.cpp
01 #include <iostream>
02 #include <vector>
03 #include <random>

04 class Zufallszahlen {
05     int *zahlen{nullptr}; // Daten
06     int n{0};             // Anzahl der Daten
07 public:
08     Zufallszahlen(int _n, int range) : zahlen(new int[_n]),n(_n){
09         std::random_device rn;
10         std::mt19937 engine(rn());
11         std::uniform_int_distribution<int> dice(1, range);
12         for(int i=0; i < _n; ++i) {
13             zahlen[i] = dice(engine);
14         }
15     }
16     ~Zufallszahlen() {
17         delete[] zahlen;
18     }
19     Zufallszahlen(const Zufallszahlen&) = delete;
20     Zufallszahlen& operator=(const Zufallszahlen&)=delete;
21     void print_All() {
22         for(int i=0; i<n; ++i) {
23             std::cout << zahlen[i] << std::endl;
24         }
25     }
26 };

```

Die Klasse ist im Prinzip recht einfach und erzeugt ein Objekt mit `n` Zufallszahlen im Bereich 1 bis `range`. Für die `n`-Daten wird zur Laufzeit Speicher vom Heap angefordert und dem rohen Zeiger `daten` zugewiesen. Im echten C++11-Leben verwenden Sie hier idealerweise `std::vector`. Wenn Sie den Code, in dem die Zufallszahlen erzeugt werden, in den Zeilen 09 bis 14 nicht verstehen, sollten Sie zu Abschnitt 2.8, »Zufallszahlen (neu in C++11)«, vorblättern. Dort wird dieser Code ebenfalls verwendet und beschrieben. Hier haben wir außerdem auch gleich per Design böse flache Kopien und Zuweisungen in den Zeilen 19 und 20 mit `=delete` verboten.

Leider hat das Verbot des Kopierkonstruktors jetzt den Seiteneffekt, dass Sie Objekte der Klasse `Zufallszahlen` nicht mehr in einen Standardcontainer, wie bspw. einen `std::vector`, packen können. Ein Beispiel:

```
01 Zufallszahlen lottozahlen(6, 49);
02 lottozahlen.print_All();
...
03 std::vector<Zufallszahlen> zvec;
04 zvec.push_back(lottozahlen); // Kopierkonstruktor = delete
```

Der Compiler wird die Übersetzung mit dem Hinweis quittieren, dass `Zufallszahlen::Zufallszahlen(const Zufallszahlen&)` hier `deleted` ist, wenn `push_back()` ausgeführt wird.

Verschieben implementieren

Im Vergleich mit dem Kopieren der Eigenschaften ist das Verschieben ohnehin wesentlich effizienter. Im Beispiel muss hierbei nicht mehr erneut ein Speicherplatz für die Kopie angefordert werden, sondern es wird lediglich die Adresse der Eigenschaft vom alten Objekt an das neue Objekt übergeben (bzw. verschoben). Der Inhalt des alten Objekts ist hiermit auch hinfällig.

Jetzt soll die Klasse `Zufallszahlen` um die Verschiebeoperationen erweitert werden. Hier der Quellcode dazu:

```
00 // kapitel07/7_10b/move02.cpp
...
01 class Zufallszahlen {
02     int *zahlen{nullptr};
03     int n{0};
04 public:
05     Zufallszahlen(int _n, int range) :
06         zahlen(new int[_n]), n(_n) {
07         ...
08     }
09     ~Zufallszahlen() {
10         delete[] zahlen;
11     }
12     Zufallszahlen(const Zufallszahlen&) = delete;
13     Zufallszahlen& operator=(const Zufallszahlen&)=delete;

12     Zufallszahlen(Zufallszahlen&& a) :
13         zahlen(std::move(a.zahlen)), n(std::move(a.n)) {
14         a.zahlen=nullptr;
15     }
```

```

14     a.n=0;
15 }
16 Zufallszahlen& operator=(Zufallszahlen &&a) {
17     n = std::move(a.n);
18     a.n=0;
19     delete[] zahlen;
20     zahlen=std::move(a.zahlen);
21     a.zahlen = nullptr;
22     return *this;
23 }

24 void print_All() {
25     ...
26 };

```

In den Zeilen 12 bis 15 wurde hier der Verschiebekonstruktor (auch Move-Konstruktor) definiert. Klar sein sollte natürlich, dass hier der Parameter in Zeile 12 nicht mehr `const` sein darf, weil ja sonst nicht »verschoben« werden kann. Selbiges gilt auch für den Verschiebe-Zuweisungsoperator (bzw. Move-Zuweisungsoperator), welcher in den Zeilen 16 bis 23 definiert wurde.

Bei beiden Verschiebeoperationen ist der Vorgang ähnlich. Zuerst werden die Eigenschaften vom alten dem neuen Objekt per Verschiebezuweisung zugewiesen, und die alten Ressourcen werden dann als freigegeben (aufgeräumt) markiert. Also im Prinzip verschieben und aufräumen.

Beim Verschiebekonstruktor in Zeile 12 verschieben wird die Eigenschaften gleich über den Element-Initialisierer hinter dem Doppelpunkt. In den Zeilen 13 und 14 räumen wird dann die alten Ressourcen auf.

Dasselbe wird auch beim Verschiebe-Zuweisungsoperator gemacht. Nur verwenden wird hier gleich `std::move()` (Zeile 17 und 20), um die Eigenschaften von einem Objekt zum anderen zu verschieben. Damit wird sichergestellt, dass die Eigenschaft in `&&` umgewandelt und die korrekte Überladung von `operator=(&&)` aufgerufen wird – genauer wird auch hier wirklich verschoben. Ohne `std::move()` hätten Sie lediglich wieder eine kopierende Zuweisung (`operator=(&)`). Das ist ja schließlich das Prinzip von Verschiebeoperationen: möglichst wenig Ressourcen neu anzulegen. Aufgeräumt wird hier in den Zeilen 18 und 21.

Wichtig ist hier auch Zeile 19, in der wir bei der Verschiebezuweisung die selbst belegte Ressource freigeben müssen, die sonst durch eine Zuweisung verloren ist.

Reihenfolge, um Verschieben zu implementieren

Wenn Sie einen eigenen Verschiebekonstruktor oder Verschiebe-Zuweisungsoperator implementieren wollen, sollten Sie dies in folgenden drei Schritten realisieren:

- ▶ den Inhalt des Zielwertes löschen
- ▶ den Quellwert in den Zielwert übertragen
- ▶ den Quellwert auf einen definierten Zustand setzen

Mehr war nicht nötig für unsere Klasse `Zufallszahlen`, um Objekte nur noch verschieben und nicht mehr Kopieren zu können. Jetzt klappt dies auch wieder mit den Standardcontainern, wie der folgende Codeausschnitt verdeutlichen soll:

```
00 // kapitel07/7_10b/move02.cpp
...
01 Zufallszahlen lottozahlen{6, 49};
02 std::vector<Zufallszahlen> zvec;

03 zvec.push_back(std::move(lottozahlen));
04 zvec.emplace_back( 3, 5 );
05 zvec.push_back(Zufallszahlen{6, 10});

06 for( auto &it : zvec ) {
07     it.print_All();
08 }
...
```

In diesem Beispiel sehen Sie in den Zeilen **03**, **04** und **05** drei Möglichkeiten, ein Objekt in den Standardcontainer zu **verschieben**:

- ▶ **std::move()**: Die erste Möglichkeit mit `std::move()` in Zeile **03** stellt so etwas wie eine Typumwandlung dar, um auch nichttemporäre Objekte, wie in Zeile **01** definiert, in einen Standardcontainer verschieben zu können. Das Objekt `lottozahlen` von Typ `Zufallszahlen` alleine passt ja nicht in den Standardcontainer und muss daher explizit mit `std::move()` in ein `Zufallszahlen&&` umgewandelt werden. Mit einer Umwandlung von `Zufallszahlen` nach `Zufallszahlen&&` geben Sie allerdings auch den Inhalt von `lottozahlen` auf, und in `push_back()` ist es nun möglich, die verschiebende Zuweisung statt der kopierenden Zuweisung zu verwenden (dank der passenden Überladung im Standardcontainer). Nach der `std::move()`-Anweisung ist das `Zufallszahlen`-Objekt `lottozahlen` zwar leer, aber immer noch ein gültiges Objekt(!). Es ist quasi ein `Zufallszahlen`-Objekt, dem seine Eigenschaften genommen wurden.

Tipp: `std::swap()`

Viele Algorithmen und Funktionen wurden mit C++11 so implementiert, dass sie auch mit der Verschiebesemantik (Move-Semantik) verwendet werden können. `std::swap()` ist bspw. ein solcher Fall, dessen Aufrufe mit temporären Objekten extrem effizient und günstig sind.

- **Temporäres Objekt:** In Zeile 05 übergeben wir in `push_back()` ein temporäres Objekt als Argument. Intern wird hierbei der Verschiebe-Zuweisungsoperator `operator=(Zufallszahlen&&)` verwendet, welcher auf das temporäre Objekt passt. Vom temporären Objekt werden jetzt die Eigenschaften durch die Verschiebeoperation in ein Vektor<Zufallszahlen>-Element verschoben und hinten angehängt. Das temporäre Objekt wird anschließend ohnehin verworfen.

Zum Weiterlesen

Etwas tiefer in diese neue Verschiebesemantik (oder Move-Semantik) wird in Abschnitt 16.1, »Move-Semantik und Perfect Forwarding«, eingestiegen. Hier wird dann auch auf *LValue* und *RValue* bzw. *LValue-Referenzen* und *RValue-Referenzen* eingegangen, worauf in diesem Abschnitt noch bewusst verzichtet wurde, um die Beschreibungen nicht noch komplizierter zu machen. Aber wenn Sie sich ernsthaft mit der neuen Verschiebesemantik von C++11 befassen wollen, sind Kenntnisse über LValue und RValue unverzichtbar.

- **`emplace_back()`:** Mithilfe von `emplace_back()` kommen Sie sogar ohne ein temporäres Objekt aus. Hier wird das Objekt direkt dort konstruiert, wo es benötigt wird. Alle Standardcontainer liefern seit dem neuen C++11-Standard solche `emplace`-Methoden mit. Diese Methode verwendet alle Argumente und sucht den passenden Konstruktor unserer Zielklasse (hier `Zufallszahlen::Zufallszahlen(int,int)`).

Wann Move-Semantik implementieren?

Der Hauptgrund, eine Move-Semantik (mit Move-Konstruktor und Move-Zuweisungsoperator) für eigene Typen zu implementieren, ist, wenn Sie unnötige tiefe Kopien vermeiden wollen. Der Move-Konstruktor ist besonders nützlich, wenn die Übergabe als Referenz und als Wert benötigt wird.

Standardversionen vom Compiler

Die Implementierung und Verwendung der Verschiebeoperationen ist im Grunde gar nicht so komplex. Schwieriger ist es, den Überblick zu bewahren, was ein Compiler von

Haus aus erstellt und was nicht. Dass in C++11 neben den großen Drei (Kopierkonstruktor, Zuweisungsoperator und Destruktor) jetzt auch noch der Verschiebekonstruktor und der Verschiebe-Zuweisungsoperator dazukommen, macht es nicht unbedingt einfacher, zu entscheiden, wann man etwas selbst definieren muss und wann nicht.

Turbo mit Verschiebeoperationen

Zwar wurden in diesem Beispiel die Verschiebeoperationen in Verbindung mit rohen Zeigern demonstriert, aber auch ohne solche Eigenschaften wie rohe Daten sind die Verschiebeoperationen extrem effizient (und billig), da auf aufwendigeres Kopieren verzichtet werden kann.

Folgende Voraussetzung müssen erfüllt werden, damit der Compiler Standardversionen der Verschiebeoperationen erstellen kann:

- Es sind keine eigenen Verschiebe- oder Kopieroperationen vorhanden. Sobald mindestens eine der Verschiebe- oder Kopieroperationen (Konstruktoren oder Zuweisungen) implementiert wurde, erstellt der Compiler auch keine Standardversionen mehr.
- Es ist kein eigener Destruktor vorhanden. Auch hier geht der Compiler davon aus, wenn ein eigener Destruktor implementiert wird, dass etwas Komplexeres gemacht wurde und eine Standardversion hier nicht ausreichend ist.
- Und zu guter Letzt muss es dem Compiler natürlich möglich sein, die entsprechenden Eigenschaften zu verschieben. Dies klappt in der Regel mit allen eingebauten Typen, rohen Zeigern (allerdings ist hier Vorsicht angebracht; siehe auch den Abschnitt »Verschieben implementieren« in Abschnitt 7.10) und vielen Standardcontainern bzw. Typen der Standardbibliothek (wie bspw. den Containertypen `std::vector`, `std::string` usw.).

Standardversion explizit anfordern

Wenn Sie wissen, was Sie tun, und der Compiler sich irrt, keine Standardversion zu erstellen, können Sie diesen ja immer noch mit `=default` (siehe Abschnitt 7.9.1, »default«) explizit anweisen, eine Standardversion zu erstellen.

Sind alle Voraussetzungen erfüllt, stellt Ihnen der Compiler die Verschiebeoperatoren (A sei die Klasse) `A::A(&&)` und `A& A::operator=(T&&)` zur Verfügung.

7.11 Friend-Klassen

Aus Abschnitt 7.7.9, »Globale friend-Funktion«, kennen Sie das Schlüsselwort `friend` ja bereits in Verbindung mit Funktionen. Sie wissen, dass Sie hiermit die Datenkapselung aufbrechen können, um auf `private` deklarierte Eigenschaften oder Methoden zugreifen zu können. Dasselbe Prinzip funktioniert auch mit Klassen, die Sie mit `friend` kennzeichnen können. Bei Klassen, die Sie mit `friend` kennzeichnen, haben Sie Zugriff auf die privaten Elemente der anderen Klasse. Der folgende Codeausschnitt soll dies demonstrieren:

```
00 // kapitel07/7_11/friend_class.cpp
...
01 class B; // Vorausdeklaration

02 class A {
03     int wert_A;
04 public:
05     A(int val) : wert_A(val) {}
06     friend class B;
07 };

08 class B {
09 public:
10     void change_A(A& a) {
11         a.wert_A++;
12     }
13 };
...
14 A aval(100);
15 B bval;
16 bval.change_A(aval);
```

Hier wird in Zeile 14 zunächst ein Objekt der Klasse A instanziiert, und in Zeile 15 wird eine Instanz der Klasse B erzeugt. Anschließend in Zeile 16 übergeben wir in der Methode der Klasse B die Klasse A als Parameter. In der Methode (Zeile 10 bis 12) wird in Zeile 11 die `private` Eigenschaft `wert_A` von Zeile 03 direkt geändert. Dies ist hier allerdings nur daher möglich, weil Sie in Zeile 06 die Klasse B in der Klasse A zum Freund erklärt haben.

Die Freundschaft ist allerdings nur einseitig. Mit einer Instanz der Klasse B könnten Sie jetzt nicht auf `private` Eigenschaften der Klasse A zugreifen (wenn es welche gäbe). Soll die Freundschaft gegenseitig sein, müssen Sie auch in der Klasse B die Klasse A zum Freund erklären.

Kapitel 8

Operatoren überladen

Viele (Standard-)Klassen und Standardcontainer bieten die Möglichkeit an, einen einfachen Vergleich mit == oder != zwischen zwei Objekten durchzuführen. Oft will man auch nur einfache arithmetische Berechnungen zweier Objekte (genauer zweier Eigenschaften) durchführen. Dies können Sie jetzt bspw. mithilfe von einfachen Methoden realisieren. In C++ haben Sie aber auch die Option, Operatoren für Ihre persönlichen Bedürfnisse der Klasse anzupassen und zu implementieren. Wenn Sie somit einen eigenen Operator in Ihrer Klasse implementieren, so wird dies als Operatorüberladung bezeichnet.

Sie haben bereits mit dem `operator=()` (Stichwort *The Big Three*) kurz gesehen, wie eine solche Operatorüberladung aussieht. Zugegeben, auf den ersten Blick sieht die Syntax in Form von `operator@` etwas seltsam aus, und nicht alle Programmierer halten die Operatorüberladung, wo es nur geht, für gut. Die Freiheit, einzelne Operatoren nach eigenen Vorstellungen zu überladen, bietet leider auch die Möglichkeit, ziemlich überraschende Effekte zu implementieren. Um bspw. nochmals auf unsere Klasse `Personal` aus Kapitel 7 zurückzukommen, so würde doch folgende Verwendung eher für Stirnrunzeln und Verwunderung als für Klarheit sorgen:

```
01 Personal person01("Max Mustermann");  
02 ++person01; // Was hier wohl geschieht???
```

Unabhängig (noch) von der Implementierung von `operator++()` stellt sich hier in Zeile 02 als Erstes die Frage: Was geschieht hier? Wird das Gehalt erhöht oder die Personalnummer? Oder geschieht etwas ganz anderes? Sie können solche Operatorüberladungen zwar in Ihren Klassen implementieren, aber so ein Beispiel stellt eher ein schlechtes und unlogisches Codedesign dar und sollte vermieden werden. Sollte bspw. in Zeile 02 eine Lohnerhöhung beabsichtigt gewesen sein, so sollte man hier stattdessen auf Methoden zurückgreifen. Eine Methode wie `Personal::lohn_erhoe-hung()` lässt auch schon allein an der Schnittstelle der Klasse erkennen, was diese Methode macht.

Operatoren, die Sie fast immer überladen wollen/können und zudem weitaus intuitiver, dürften die Vergleichsoperatoren `==` und `!=` sein, um zwei Objekte miteinander zu vergleichen. So wird die Logik und die Absicht der Operatorüberladung `==` und `!=`

(ohne die Implementierung davon zu kennen) vom folgenden Codeausschnitt recht schnell erfasst:

```
01 Personal person01("Max Mustermann");
02 Personal person02("Max Musterfrau");

03 if( person01 == person02 ) {
04     // Beide Objekt sind gleich
05 }
06 if( person01 != person02 ) {
07     // Verschiedene Objekte
08 }
```

Hier lässt sich ohne Umschweife in den Zeilen **03** und **06** erkennen, was der Sinn dieser Operatorüberladung ist. Hier werden zwei Objekte auf Gleichheit getestet. Natürlich ist es auch hier möglich, das Design kaputt zu machen und per Implementierung den beiden Vergleichsoperatoren eine ganz andere Aufgabe zuzuweisen.

Sie haben aber jetzt verstanden, worauf dieser Abschnitt abzielt. Wenn Sie Operatoren überladen, dann sollten Sie nichts Unerwartetes oder Überraschendes machen. Wenn Sie bspw. zwei Integer miteinander mit dem `++`-Operator addieren, erwarten Sie ja auch die Summe der beiden Integer und nichts anderes. So soll es künftig auch bei Ihren Klassen sein, wenn Sie Operatoren überladen. Überladen Sie also die Operatoren nicht auf Teufel komm heraus, sondern verwenden Sie diese Technik nur, wenn es sinnvoll (und intuitiv) erscheint. Ein gewaltsames Überladen und Anbieten aller Operatoren ist in den seltensten Fällen sinnvoll und häufig auch gar nicht nachvollziehbar. In allen anderen Fällen schreiben Sie stattdessen einfach eine Methode. Ich werde in den anschließenden Codebeispielen das eine oder andere Mal nochmals darauf hinweisen, wo es sinnvoller wäre, eine Methode statt einer Operatorüberladung zu verwenden.

Ebenfalls erwähnt werden muss: Dass etwas im Code (der Implementierung) des zu überladenen Operators geschieht, ist ebenfalls Ihre Entscheidung. Sie entscheiden, welche Eigenschaften Sie wie und ob verändern und was passieren soll, wenn der überladene Operator aufgerufen wird. Sie haben quasi in C++ die komplette Freiheit, den Code für den Operator zu schreiben. Allerdings wie im letzten Absatz bereits erwähnt, tun Sie nichts Überraschendes, wenn Sie die Überladung eines Operators verwenden, sondern machen Sie das, was man dabei auch erwarten würde.

Beispiele in diesem Kapitel

Die Beispiele in diesem Kapitel sind meistens relativ einfach gehalten und dienen nur der Demonstration. In vielen Fällen würde ich eine bestimmte Operatorüberladung gar nicht implementieren.

Ziel und Zweck der Beispiele ist es lediglich, die Überladung von Operatoren zu zeigen, ohne gleich komplizierten Code oder eine umfangreiche Klasse zu erstellen. Der Fokus liegt hier ganz klar auf der Operatorüberladung!

8.1 Grundlegendes zum Überladen von Operatoren

Operatorüberladungen sind im Grunde wie gewöhnliche Funktionen oder Methoden. Im Gegensatz zu gewöhnlichen Methoden bzw. Funktionen benötigen Sie bei der Operatorüberladung allerdings statt des Bezeichners das Schlüsselwort `operator`, gefolgt vom Operatorzeichen.

Kurze Begriffserklärung: Binäre Operatoren und unäre Operatoren

Binäre Operatoren (Bi = zwei) stehen zwischen zwei Werten (auch als Operanden bezeichnet) und verknüpfen diesen Ausdruck zu einem neuen Wert. Einfaches Beispiel wäre eine Addition wie $10 + 11$, bei der der `+`-Operator der binäre Operator ist und 10 der linke bzw. 11 der rechte Operand.

Unäre Operatoren hingegen beziehen sich nur auf einen einzelnen Wert (auch hier als Operand bezeichnet). Ein unärer Operator hat somit nur eine einstellige Verknüpfung mit einem Operanden.

8.1.1 Binäre Operatoren

Klassische binäre Operatoren (wie bspw. `+` oder `-`) lassen sich mit folgender Syntax als Methode einer Klasse überladen:

```
Rückgabewert [Klassenname::]operator @ ( Typ )
```

Der Rückgabewert spricht für sich. Das Schlüsselwort `operator` ist unbedingt nötig für die Operatorüberladung und das Zeichen `@` hinter dem Schlüsselwort `operator` muss durch das von Ihnen gewünschte Operatorzeichen ersetzt werden, welches Sie für die Klasse `Klassenname` überladen wollen. Der linke Operand wird auch hier wie bei den klassischen Methoden im Kontext (`objekt01.operator@(objekt02)`) entsprechend aufgerufen. Den rechten Operanden hingegen geben Sie über `TYP` an (`objekt01.operator@(objekt02)`).

Den so überladenen binären Operator einer Klasse können Sie jetzt auf zwei unterschiedliche Weisen verwenden:

```
01 Klassenname objekt01, objekt02;
02 // Möglichkeit 1:
```

```

03 Rückgabewert bez01 = objekt01.operator@(objekt02);
04 // Möglichkeit 2:
05 Rückgabewert bez02 = objekt01 @ objekt02;

```

Das Zeichen @ in den Zeilen 03 und 05 sollte natürlich gegen den Operator ausgetauscht werden, der überladen werden soll. In Zeile 03 verwenden Sie den klassischen Aufruf einer Methode, womit Sie die Umwandlung auch gleich schon selbst vorgenommen haben. Die Schreibweise in Zeile 05 entspricht der Methodenschreibweise von Zeile 03, nur dass hier der Aufruf noch implizit (und automatisch) vom Compiler in einen Methodenaufruf wie in Zeile 03 umgewandelt wird. Beide Schreibweisen (Zeilen 03 und 05) sind somit erlaubt. Allerdings ist die Verwendung der Schreibweise aus Zeile 05 die gängigere Möglichkeit, eine Operatorüberladung zu verwenden.

Hierzu ein einfaches Beispiel, in dem der binäre Operator + gleich zweimal für die Klasse Sales überladen wurde:

```

00 // kapitel08/overloading01.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
        paperback(p), ebook(e) {}
06     Sales operator + (const Sales& op2) const;
07     Sales operator + (unsigned int val) const;
...
08 };

09 Sales Sales::operator + (const Sales& op2) const {
10     Sales tmp(*this);
11     tmp.paperback += op2.paperback;
12     tmp.ebook += op2.ebook;
13     return tmp;
14 }

15 Sales Sales::operator + (unsigned int val) const {
16     Sales tmp(*this);
17     tmp.paperback += val;
18     return tmp;
19 }

```

```

...
20 Sales book01(100, 200);
21 Sales book02(300, 100);

22 Sales gesamt = book01 + book02;
23 Sales book01_neu = book01 + 200;

```

Die einfach gehaltene Klasse enthält lediglich zwei Eigenschaften, welche die Verkaufszahlen von Büchern in Paperback-Form und als eBook enthalten. Die Deklarationen der beiden Operatorüberladungen finden Sie in den Zeilen 06 und 07. Beide Operatorüberladungen werden als `const` implementiert, weil sinngemäß der linke Operand bei dem `++`-Operator logischerweise nicht verändert wird. Als Rückgabewert wurde ebenfalls ein `Sales`-Objekt verwendet, weil eine Addition von zwei `Sales`-Objekten sinnvollerweise ein neues `Sales`-Objekt ergibt.

Die Implementierung der Operatorüberladung aus Zeile 06 finden Sie in den Zeilen 09 bis 14. Zuerst erstellen wir hier ein lokales `Sales`-Objekt aus dem linken Operanden des `++`-Operators in Zeile 10 mithilfe des Kopierkonstruktors. In den Zeilen 11 und 12 addieren Sie dann zu diesem lokalen Objekt die Eigenschaften des zweiten Objekts (= rechter Operand des `++`-Operators) hinzu. Damit haben Sie sinngemäß die Eigenschaften beider Objekte addiert, ohne, wie es bei einer Addition auch der Fall ist, den linken bzw. den rechten Operanden zu ändern. Dank dieser Implementierung lässt sich jetzt die Addition zweier Objekte durchführen, wie dies in Zeile 22 geschieht.

Mehrfache Verwendung möglich

Die erstellte Operatorüberladung lässt sich übrigens durchaus mehrfach anwenden. So können Sie die Verkaufszahlen mehrerer Bücher wie folgt berechnen lassen:

```
Sales book_all = book01 + book02 + book03;
```

Intern wird hierbei allerdings die `++`-Operatorüberladung zweimal aufgerufen.

Die zweite Implementierung von Zeile 07 finden Sie dann in den Zeilen 15 bis 19. Im Gegensatz zur anderen Implementierung wurde jetzt der Datentyp des Parameters (`unsigned int`) geändert (was der rechte Operand ist). Auch hier wird zunächst ein neues Objekt in Zeile 16 aus dem linken Operanden über den Standard-Kopierkonstruktor erstellt (weil ja auch hier keiner der Operanden geändert wird). Dann addieren Sie den Wert des Parameters in Zeile 17 zur Eigenschaft `Sales::paperback` hinzu und geben das Objekt zurück. Diese Implementierung der Operatorüberladung können Sie in Zeile 23 bei der Ausführung sehen.

Design-Verbesserungsvorschlag

Die zweite Überladung von `operator+(unsigned int)` und anschließend dann die »freie« Operatorüberladung von `operator+(unsigned int, Sales&)` in Abschnitt 8.1.2 würde ich allerdings in der Praxis so nicht empfehlen, weil hierbei wieder nicht sofort erkennbar ist, zu welcher der beiden Eigenschaften (`paperback` oder `ebook`) der Wert hinzuaddiert wird. Hierfür würde ich eher zu zwei Methoden wie `Sales::add_paperback(unsigned int)` und `Sales::add_ebook(unsigned int)` tendieren. Aber zu Demonstrationszwecken der Operatorüberladung ist *overloading01.cpp* und das im nächsten Abschnitt folgende Beispiel *overloading02.cpp* in Ordnung und einfach gehalten, und letztendlich entscheiden nach wie vor Sie, wie (oder auch ob) Sie eine Operatorüberladung implementieren.

8.1.2 Binäre Operatoren als (friend-)Funktion überladen

Durch die Überladung von `operator+` sind Sie jetzt für `objekt01+objekt02` und `objekt01+100` gerüstet. Aber über einen Aufruf wie `100+objekt01` würde sich der Compiler beschweren, weil der linke Operand ja klassentypisch als Objekt einer Klasse implementiert ist. In solch einem Beispiel kommen Sie nicht um eine Operatorüberladung als freie (friend-)Funktion herum.

Die Syntax einer freien Hilfsfunktion, um einen binären Operator zu überladen, sieht wie folgt aus:

```
Rückgabewert operator @ (TYP1 op1, TYP2 op2)
```

Der Rückgabewert spricht wieder für sich. Das Symbol `@` ersetzen Sie für den zu überladenden binären Operator. Da Funktionen ja nicht mehr zur Klasse gehören, müssen Sie jetzt in der Parameterliste zusätzlich den linken Operanden `op1` vom Datentyp `TYP1` angeben. Als zweiten Parameter geben Sie den rechten Operanden `op2` vom Datentyp `TYP2` an. Bei der Definition der Hilfsfunktion wird **kein** Scope wie `Klassenname::` mehr benötigt. Dafür müssen Sie jetzt die zu überladende Hilfsfunktion in der Klasse mit `friend` deklarieren.

Zum Nachlesen

Die `friend`-Funktionen werden in Abschnitt 7.7.9, »Globale friend-Funktion«, behandelt.

Den so überladenen binären Operator als freie Hilfsfunktion können Sie jetzt auf zwei unterschiedliche Arten verwenden:

```

01 Klassenname objekt01, objekt02;
02 // Möglichkeit 1:
03 Rückgabewert bez01 = operator@(objekt01, objekt02);
04 // Möglichkeit 2:
05 Rückgabewert bez02 = objekt01 @ objekt02;

```

Das Zeichen @ in den Zeilen 03 und 05 sollte auch hier wieder gegen den Operator ausgetauscht werden, der überladen werden soll. In Zeile 03 verwenden Sie den klassischen Funktionsaufruf, womit Sie die Umwandlung auch gleich schon selbst vorgenommen haben. Die Schreibweise in Zeile 05 entspricht der Funktionsschreibweise von Zeile 03, nur dass hier der Aufruf noch implizit (und automatisch) vom Compiler in einen Funktionsaufruf wie in Zeile 03 umgewandelt wird. Beide Schreibweisen (Zeilen 03 und 05) sind somit erlaubt. Allerdings ist auch hier die Verwendung in Zeile 05 die gängigere Möglichkeit, eine Operatorüberladung als freie Hilfsfunktion zu verwenden.

Bezogen auf unsere Klasse `Sales` können Sie jetzt wie folgt eine freie Funktion als Operatorüberladung realisieren:

```

00 // kapitel08/overloading02.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
        paperback(p), ebook(e) {}
06     Sales operator + (const Sales& op2) const;
07     Sales operator + (unsigned int val) const;
08     friend Sales operator+( unsigned int val,
        const Sales& op2 );
...
09 };
...
10 Sales operator + (unsigned int val, const Sales& op2) {
11     Sales tmp(op2);
12     tmp.paperback += val;
13     return tmp;
14 }
...
15 Sales book01(300, 100);
16 Sales book01_neu = 400 + book01;

```

Die Definition der freien Hilfsfunktion finden Sie in den Zeilen 10 bis 14. Im Funktionskopf in Zeile 10 finden Sie in der Parameterliste auf der linken Seite mit `val` den linken Operanden und auf der rechten Seite mit `Sales` (hier ist eine Referenz effizienter) den rechten Operanden. Natürlich muss auch diese Funktion ein `Sales`-Objekt zurückliefern. Ansonsten finden Sie nicht viel Neues im Ablauf der Funktion. Zuerst wird wieder mit dem Standard-Kopierkonstruktor eine Kopie des rechten Operanden (`Sales`-Objekt) erzeugt, der Wert von `val` zur Kopie `tmp.paperback` hinzuaddiert (man hätte auch `ebook` stattdessen verwenden können) und das Objekt zurückgegeben. Damit Sie diese freie Hilfsfunktion dann auch wirklich wie in Zeile 16 verwenden können, müssen Sie diese in der Klasse als `friend` deklarieren, wie es hier in Zeile 08 geschehen ist.

Statt des Aufrufes wie in Zeile 16 könnten Sie hier auch folgendes gleichwertige Gegenstück (wenn auch eher unüblich) verwenden, um die freie Hilfsfunktion aufzurufen:

```
16 Sales book02_neu = operator+(400, book02);
```

8.1.3 Unäre Operatoren

Um Operatorüberladungen für unäre Operatoren mit einzelnen Operanden zu implementieren, wird geringfügig anders vorgegangen. Die Syntax, um einen unären Operator (wie bspw. `++` oder `--`) als Methode zu überladen, sieht wie folgt aus:

```
Rückgabewert [Klassenname::]operator @ ()
```

Auch hier spricht der Rückgabewert für sich. Hinter dem Schlüsselwort `operator` müssen für `@` den gewünschten unären Operator angeben, den Sie beabsichtigen, für die Klasse `Klassenname` zu überladen. Ansonsten benötigen Sie keinen weiteren Parameter.

Den so überladenen unären Operator einer Klasse können Sie jetzt auf zwei unterschiedliche Arten verwenden:

```
01 Klassenname objekt01;
02 // Möglichkeit 1:
03 Rückgabewert bez01 = objekt01.operator@();
04 // Möglichkeit 2:
05 Rückgabewert bez02 = @objekt01;
```

Das Zeichen `@` in den Zeilen 03 und 05 wird auch hier wieder gegen den Operator ausgetauscht, der überladen werden soll. In Zeile 03 verwenden Sie den klassischen Aufruf einer Methode. Die gängigere Schreibweise in Zeile 05 entspricht der Methodenschreibweise von Zeile 03, nur dass hier der Aufruf noch implizit (und automatisch) vom Compiler in einen Methodenaufruf wie in Zeile 03 umgewandelt wird.

Als Beispiel wollen wir hier die Präfixschreibweise von ++objekt und --objekt für die Klasse Sales zur Demonstration überladen. Hier das Beispiel dazu:

```

00 // kapitel08/overloading03.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
        paperback(p), ebook(e) {}
06     Sales& operator++();
07     Sales& operator--();
...
08 };

09 Sales& Sales::operator++() {
10     paperback++;
11     ebook++;
12     return *this;
13 }

14 Sales& Sales::operator--() {
15     if(paperback > 0 && ebook > 0) {
16         paperback--;
17         ebook--;
18     }
19     return *this;
20 }
...
21 Sales book01(100, 200);
22 Sales book02(300, 100);
23 --book01;
24 ++book02;

```

An der Deklaration der beiden Operatorüberladungen in den Zeilen 06 und 07 können Sie schon erkennen, dass wir hier kein neues temporäres Objekt zurückgeben, sondern das aktuelle Objekt als Referenz zurückliefern. Reicht ja auch aus, weil man von ++objekt und --objekt auch nichts anderes erwartet, als dass der Wert des aktuellen Speicherobjekts erhöht oder reduziert wird.

Entsprechend einfach sind daher auch die Implementierungen in den Zeilen 09 bis 13 mit der Präfixversion von ++ und 14 bis 20 mit der Präfixversion von -- gehalten. Bei der Implementierung der Präfixschreibweise von ++ wurden beide Eigenschaften

paperback und ebook um eins erhöht und am Ende als this-Zeiger wieder zurückgegeben. Gleiches geschieht auch bei der Implementierung der Präfixschreibweise von --, nur überprüfen wir hier vorher noch, ob die Anzahl der verkauften Bücher nicht schon 0 ist, bevor wir beide Eigenschaften dekrementieren. Beide Versionen sehen Sie dann in den Zeilen 23 und 24 im Einsatz.

Spezialfall: Postfixschreibweise von ++ und --

Die eben gezeigte Version der Operatorüberladung für die unären Operatoren ++ und -- funktioniert allerdings nur, solange der Operand rechts vom Operator steht (also ++objekt oder --objekt). Sobald der Operator auf der rechten Seite steht und der Operand links (wie bspw. in der Postfixschreibweise von objekt++ oder objekt--), benötigen Sie einen extra Dummy-Parameter, um diese Schreibweise realisieren zu können. Das folgende Beispiel zeigt diesen Sonderfall und wie Sie objekt++ und objekt-- implementieren können:

```
00 // kapitel08/overloading04.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
06         paperback(p), ebook(e) {}
07     Sales& operator++();
08     Sales& operator--();
09     Sales operator++(int);
10     Sales operator--(int);
...
10 };
...
11 Sales Sales::operator++(int) {
12     Sales tmp(*this);
13     ++paperback;
14     ++ebook;
15     return tmp;
16 }

17 Sales Sales::operator--(int) {
18     if(this->paperback <= 0 || this->ebook <= 0) {
19         return *this;
20     }
21     Sales tmp(*this);
```

```

22    --paperback;
23    --ebook;
24    return tmp;
25 }
...
26 Sales book01(100, 200);
27 Sales book02(300, 100);
28 book01--;
29 book02++;

```

Anhand der Deklaration in den Zeilen **08** und **09** können Sie schon sehen, dass neben dem Dummy-Parameter vom Typ `int` als Rückgabewert jetzt auch ein Objekt der Klasse verwendet wird. Warum das so ist, erfahren Sie gleich.

Die Implementierungen der beiden Postfixnotationen von `++` und `--` finden Sie in den Zeilen **11** bis **16** und **17** bis **25**. Betrachten Sie Zeile **12** bzw. **21**, wo wir mithilfe des Standard-Kopierkonstruktors eine Kopie vom aufrufenden Objekt anlegen. Anschließend inkrementieren bzw. dekrementieren wir beide Eigenschaften der Klasse `Sales` und geben die Kopie an den Aufrufer zurück. Dies tun wir daher, weil wir hiermit die Regel der Postfixschreibweise einhalten wollen, die gewöhnlich so erwartet wird. Und die besagt, dass der Wert (hier die Eigenschaften) erst nach der Auswertung des Objekts geändert wird. Den Unterschied zwischen `objekt++` und `++objekt` haben Sie ja bereits in Abschnitt 3.3.1, »Zähloperatoren (Inkrement und Dekrement)«, kennengelernt. Bei der Postfixversion von `--` überprüfen wir außerdem noch in Zeile **18**, ob eine der beide Eigenschaften nicht bereits 0 ist. In dem Fall wird nicht mehr dekrementiert und wir sparen uns das Kopieren und geben gleich den `this`-Zeiger zurück. Ordentlicher wäre es vielleicht noch, wenn Sie hier beide Eigenschaften einzeln behandeln. Beide Postfixversionen finden Sie in den Zeilen **28** und **29** im Einsatz.

8.1.4 Unäre Operatoren als (friend-)Funktion überladen

Hierzu die Syntax, um einen unären Operator als Funktion zu verwenden:

```
Rückgabewert operator @ (Klassenname& op1)
```

Auch in diesem Fall spricht der Rückgabewert wieder für sich. Hinter dem Schlüsselwort `operator` ersetzen Sie `@` durch den unären Operator, den Sie überladen wollen. Da die Funktion nicht mehr Bestandteil der Klasse ist und daher gewöhnlich als friend-Funktion der Klasse implementiert wird, benötigen Sie einen Parameter, welcher auch dem Typ der Klasse entspricht, für die der Operator überladen wird.

Den so überladenen unären Operator als freie Hilfsfunktion können Sie jetzt ebenfalls wieder auf zwei unterschiedliche Arten verwenden:

```

01 Klassenname objekt01;
02 // Möglichkeit 1:
03 Rückgabewert bez01 = operator@(objekt01);
04 // Möglichkeit 2:
05 Rückgabewert bez02 = @objekt01;

```

Das Zeichen @ in den Zeilen 03 und 05 sollte hier wieder gegen den unären Operator ausgetauscht werden, der überladen wurde. In Zeile 03 verwenden Sie den klassischen Funktionsaufruf, womit Sie die Umwandlung auch gleich schon selbst vorgenommen haben. Die Schreibweise in Zeile 05 entspricht der Funktionsschreibweise von Zeile 03, nur dass hier der Aufruf noch implizit (und automatisch) vom Compiler in einen Funktionsaufruf wie in Zeile 03 umgewandelt wird.

Zur Demonstration soll hier der !-Operator mit der Klasse Sales als freie Hilfsfunktion überladen werden:

```

00 // kapitel08/overloading05.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
        paperback(p), ebook(e) {}
06     friend bool operator ! (const Sales& op);
...
07 };

08 bool operator ! (const Sales& op) {
09     return ( op.paperback == 0 ) && (op.ebook == 0) ;
10 }
...
11 Sales book01;
12 if( !book01 ) {
13     std::cout << "Noch keine Bücher verkauft\n";
14 }

```

In den Zeilen 08 bis 10 finden Sie die Implementierung der freien Hilfsfunktion für den NOT-Operator, der C++-mäßig einen bool-Wert zurückgibt, je nachdem, ob die Bedingung wahr (true) oder falsch (false) ist. Um die Bedingung überprüfen zu können, mussten wir das Objekt als Referenz mit übergeben. Die Bedingung gibt true zurück, wenn beide Eigenschaften (paperback und ebook) des Objekts in Zeile 09 gleich 0 sind. Ansonsten wird false zurückgegeben. Damit diese freie Hilfsfunktion auch mit unserer Klasse Sales zusammenarbeitet, müssen wir diese selbstverständ-

lich wieder mit `friend` kennzeichnen, wie Sie es in Zeile 06 sehen können. Die freie Hilfsfunktion finden Sie in Zeile 12 im Einsatz.

8.1.5 Unterschied zwischen Operatorüberladung und Methoden

Wenn Sie sich jetzt die Beispiele zu den Operatorüberladungen so betrachtet haben, werden Sie sicherlich die Ähnlichkeit mit Methoden und Funktionen darin erkannt haben. Und tatsächlich ist ein selbst implementierter Operator nichts anderes als eine Methode bzw. Funktion, welche lediglich in einer etwas anderen Syntax verpackt wurde. Anstelle eines Bezeichners wie `methoden_name()` oder `funktions_name()` besteht der Name einer Operatorfunktion aus dem Schlüsselwort `operator`, gefolgt vom gewünschten zu überladenden Operator. Und in der Verwendung bzw. dem Aufruf können Sie statt eines methoden- bzw. funktionsüblichen Aufrufes wie `operator@(val1, val2)` eben `val1 @ val2` schreiben. In dem Fall (wenn Sie bspw. `val1 @ val2`) verwenden, übernimmt der Compiler die Umwandlung nach `operator@(val1, val2)` für Sie. Das waren sie aber auch schon, die Unterschiede zwischen Operatorüberladung und Methoden bzw. Funktionen.

8.1.6 Regeln für die Operatorüberladung

Die Operatorüberladung bietet Ihnen eine Menge an Freiraum. Trotzdem gibt es einige Regeln, die hierbei noch beachtet werden müssen:

- ▶ Sie können keine neuen Operatoren erzeugen. Sie können nur Operatoren verwenden, die bereits in C++ existieren. Aber selbst hier gibt es Operatoren, die nicht überladen werden können (Tabelle 8.2).
- ▶ Die Anzahl an Operanden kann ebenfalls nicht geändert werden. Ein binärer Operator bekommt auch weiterhin zwei Operanden, und ein unärer Operator bleibt bei einem Operanden.
- ▶ Auch die Rangfolge (Priorität) der Operatoren bleibt unverändert. Also auch bei einer Operatorüberladung gilt bspw. die Punkt-vor-Strich-Regelung.
- ▶ Sie können keine Standardargumente verwenden, und auch die Anzahl der Argumente, welche ein Operator in C++ ursprünglich hat, kann nicht geändert werden.
- ▶ Operatorüberladungen finden immer im Zusammenhang mit Klassen statt. Es muss somit immer mindestens ein Argument in der Operatorüberladung ein Klassenobjekt oder eben eine Methode sein. Es ist nicht möglich, die vorgegebene Operatorfunktion von Basisdatentypen (wie `int`, `double` etc.) zu ändern.
- ▶ Einige Operatoren dürfen nur als klasseneigene Methode (und nicht als freie Hilfsfunktion) überladen werden. Dies sind die Operatoren `=`, `->`, `()`, `[]`, die Konvertierungsoperatoren sowie eine klasseneigene Speicherverwaltung mit `new` und `delete`.

8.1.7 Überladbare Operatoren

Jetzt fehlt Ihnen nur noch eine kurze Übersicht darüber, welche Operatoren Sie alles überladen können (siehe Tabelle 8.1).

Operatoren	Beschreibung
+ - * / % (unär) ++ -- += -= *= %=	arithmetische Operatoren
&& !	logische Operatoren
== != < <= > >=	Vergleichsoperatoren
& ^ ~ >> << &= = ^= >>= <<=	Bit-Operatoren
* -> ->* () [] & new delete new[] delete[]	sonstige Operatoren

Tabelle 8.1 Operatoren, die Sie überladen können

Es gibt aber auch Operatoren, die nicht überladen werden können. Eine Auflistung der nicht überladbaren Operatoren finden Sie in Tabelle 8.2.

Operatoren	Beschreibung
.	Zugriffsoperator
.*	Zugriffsoperator (Elementzeiger)
::	Bereichsoperator
?:	bedingte Auswertung
sizeof	Größe von Objekten

Tabelle 8.2 Operatoren, die Sie nicht überladen können

Im Grunde sind Sie jetzt für jegliche Operatorüberladung gerüstet. Sie kennen jetzt die Grundlagen, und da Sie wissen, dass Operatorüberladungen letztendlich auch nur Methoden bzw. freie (friend-)Funktionen sind, sollten Sie keine Probleme mehr damit haben. In den folgenden Abschnitten soll daher noch kurz auf die einzelnen Operatoren und deren typische Semantik eingegangen werden.

8.2 Zuweisungsoperator – operator=

Die Überladung des Zuweisungsoperators wurde bereits im Abschnitt »(Standard-) Kopierkonstruktor« in Abschnitt 7.4.2 beschrieben, wo Sie erfahren haben, dass der Compiler eine Standardversion zur Verfügung stellt. Dann wurde das Thema nochmals behandelt in Abschnitt 7.8.2, »Roher Zeiger als Element in einer Klasse«, wo Sie erfahren haben, wie und warum Sie eine eigene Version implementieren können bzw. sollen. Dort haben Sie auch den Zusammenhang mit *The Big Three* erfahren. Ich denke, nachdem Sie die Einführung zu den Operatoren gelesen haben, dürften Ihnen die vorangegangenen Abschnitte, in denen wir den Zuweisungsoperator bereits implementiert haben, nicht mehr kompliziert erscheinen.

Die großen Drei nochmals in Kurzfassung

Sobald Sie den **Kopierkonstruktor**, **Destruktor** oder den überladenen **Zuweisungsoperator** selbst implementiert haben, bedeutet dies, dass die Standardversionen des Compilers nicht mehr ausreichend sind, daher sollten Sie jeweils immer die beiden anderen Methoden ebenfalls implementieren.

Da der Zuweisungsoperator in C++ doch etwas spezieller und sehr wichtig ist, soll dieser hier nochmals kurz aufgerollt werden. Die Syntax eines Zuweisungsoperators, sieht wie folgt aus:

```
Klassenname& [Klassenname::]operator = (const TYP& p)
```

Für **Klassenname** verwenden Sie die Klasse, für die Sie den Zuweisungsoperator überladen wollen. Anschließend folgen das Schlüsselwort `operator` und der Zuweisungsoperator. Als Parameter geben Sie den TYP des rechten Operanden an, welcher der Klasse als linker Operand zugewiesen wird. Als Rückgabewert wird generell eigentlich immer eine Referenz auf das aktuelle Objekt zurückgegeben. Nur durch eine Referenz als Rückgabewert und die anschließende Dereferenzierung des Zeigers `this` ist sichergestellt, dass eine Verkettung wie `a=b=c` möglich wird. Dazu aber gleich mehr. Der Aufruf des Zuweisungsoperators kann wieder auf zwei unterschiedliche Arten erfolgen:

```
01 Klassenname objekt01, objekt02;
02 // Möglichkeit 1:
03 objekt02.operator=(objekt01);
04 // Möglichkeit 2:
05 objekt02 = objekt01;
```

Wann sollte operator= überladen werden?

Den Zuweisungsoperator überlädt man in der Regel nur dann, wenn eine Klasse dynamische Eigenschaften enthält (Zeiger auf Speicherbereich). Was passieren kann,

wenn man dies nicht tut, wurde bereits in Abschnitt 7.8.2, »Rohe Zeiger als Element in einer Klasse«, erklärt. Bezogen auf unsere Klasse `Sales` ist eine Überladung absolut unnötig. Ein Beispiel:

```
01 Sales book01(2000, 2000);
02 Sales book02;
03 book02 = book01; // Zuweisung
```

In diesem Beispiel reicht in Zeile **03** die vom Compiler zur Verfügung gestellte Standardversion des Zuweisungsoperators völlig aus und kopiert hier alle Eigenschaften der Klasse. Eine Überladung ist hier daher überhaupt nicht nötig.

Zuweisungsoperator oder Kopierkonstruktor

Wie im Abschnitt »(Standard-)Kopierkonstruktor« in Abschnitt 7.4.2 schon erwähnt wurde, wird mit

```
01 Sales book01;
02 Sales book02 = book01 // Kopierkonstruktor !!!
```

der Kopierkonstruktor verwendet, und nicht, wie man vielleicht annehmen würde, der Zuweisungsoperator. Genau genommen handelt es sich in Zeile **02** um eine Initialisierung. Um etwas zuweisen zu können, muss schon ein fertiges Objekt existieren, wie dies nun mal bei den folgenden Codezeilen der Fall ist:

```
01 Sales book01(2000, 2000);
02 Sales book02;
03 book02 = book01; // Zuweisungsoperator
```

Anders sieht dies, wie eben erwähnt wurde, bei dynamischen Eigenschaften einer Klasse aus. Hier müssen Sie die großen Drei implementieren, wie bereits in Abschnitt 7.8.2 demonstriert wurde.

Hierbei will ich Ihnen aber noch eine etwas andere Technik zeigen, mit der Sie sich eine Menge Kopierarbeit und Speichermanagement sparen können. Eine beliebte Technik der Implementierung des Zuweisungsoperators stellt das Kopieren und Tauschen (*Copy & Swap*) dar. Der folgende Codeausschnitt soll Ihnen diese Technik demonstrieren:

```
00 // kapitel08/copy_and_swap.cpp
...
01 class Dynamic {
02     int* cPtr;
03 public:
04     Dynamic(int n=1) { cPtr=new int(n); }
05     Dynamic(const Dynamic& org) {
06         cPtr = new int(org.get_cPtr());
```

```

07     }
08     ~Dynamic() { delete[] cPtr; }
09     Dynamic& operator=(const Dynamic & rhs) {
10         if(this != &rhs) {
11             Dynamic tmp(rhs);
12             std::swap(cPtr, tmp.cPtr);
13         }
14         return *this;
15     }
16     int get_cPtr() const { return *cPtr; }
17 };
...
...
18 Dynamic d01;
19 {
20     Dynamic d02(5);
21     d01 = d02;
22 }
23 std::cout << d01.get_cPtr() << std::endl;

```

In der folgenden Klasse `Dynamic` wurden die großen Drei implementiert. Der Kopierkonstruktor in den Zeilen 05 bis 07 muss in diesem Beispiel nur neuen Speicher besorgen und initialisieren. Der Destruktor in Zeile 08 muss hinterher wieder aufräumen, wenn der Gültigkeitsbereich eines Objekts endet.

Den Zuweisungsoperator sehen Sie hier in den Zeilen 09 bis 15 ebenso gleich `inline` definiert. Hier dürften Sie gleich feststellen, dass dieser jetzt etwas anders implementiert wurde, als dies in Abschnitt 7.8.2 gemacht wurde. Wichtig bei einer Zuweisung ist zunächst immer, dass Sie eine versehentliche Selbstzuweisung von `obj01=obj01` überprüfen und vermeiden, was wir in Zeile 10 auch machen. Im Falle einer Selbstzuweisung wird gleich der `this`-Zeiger in Zeile 14 zurückgeliefert.

Prüfung auf Selbstzuweisung wirklich nötig?

Wer hier etwas länger auf den Code blickt, wird feststellen, dass wir ja eine Kopie vom Objekt selbst machen, mit dem wir tauschen, was dann eigentlich ungefährlich sein sollte. Und in der Tat können Sie beim Copy-&-Swap-Idiom auf eine Selbstüberprüfung verzichten. Wir haben es aber trotzdem aus Performance-Gründen verwendet, weil wir dem Rechner mit den Zeilen 11 und 12 im Falle einer Selbstzuweisung unnötige Arbeit ersparen können.

Der eigentliche Code der Zuweisung besteht jetzt nur noch aus zwei Zeilen (11 und 12). In Zeile 11 erzeugen wir ein neues `tmp`-Objekt mithilfe des Kopierkonstruktors. Mit

der Standardfunktion `std::swap()` der C++-Standardbibliothek werden jetzt **ALLE** Eigenschaften der Klasse getauscht. Diese `std::swap`-Funktion zum Vertauschen von Objekten bzw. ihren Eigenschaften ist gerade bei dynamisch reserviertem Speicher sehr hilfreich, weil man sich hiermit eine Menge Kopiarbeit und das Speichermanagement sparen kann, gerade weil hiermit auch die Zeiger auf den Speicherbereich zwischen beiden Objekten getauscht werden. Damit auch fortlaufende Zuweisungen wie `a=b=c` funktionieren, wird am Ende der `this`-Zeiger zurückgegeben, womit bspw. `c` sich selbst an `b` und das Ergebnis davon sich selbst an `a` weiterreicht.

Die Klasse bei der Ausführung wird in den Zeilen **18** bis **23** demonstriert. Zunächst wird mit Zeile **18** der Konstruktor aus Zeile **04** aktiv und reserviert Speicher für ein Element. Jetzt öffnen wir einen neuen Gültigkeitsbereich in Zeile **19** und legen dort ein weiteres Objekt in Zeile **20** mit Platz für fünf Elemente an.

Jetzt führen wir in Zeile **21** unseren selbst implementierten Zuweisungsoperator (Zeilen **09** bis **15**) aus. Im Zuweisungsoperator in Zeile **11** rufen wir den Kopierkonstruktor (Zeilen **05** bis **07**) auf und erstellen ein neues temporäres Objekt aus dem rechten Operanden. In Zeile **12** tauschen wird jetzt die Objekte bzw. die Eigenschaften der Objekte und geben diese Daten in Zeile **14** weiter. Am Ende von Zeile **15** (Ende des Gültigkeitsbereichs von `tmp`) werden die alten Daten von `tmp` mithilfe des Destruktors freigegeben. Fertig, das war es mit der Zuweisung in Zeile **21**. Am Ende des Gültigkeitsbereichs in Zeile **22** wird auch der Destruktor für das Objekt `d02` von Zeile **20** aufgerufen.

Das Objekt `d01` von Zeile **18** hat somit alle Eigenschaften vom lokaleren Objekt `d02` (Zeile **20**) per Kopieren und Tauschen erhalten, und wir haben auch keinerlei »Speicherleichen« liegen lassen.

Sie kennen somit die manuelle Implementierung des Zuweisungsoperators aus Abschnitt 7.8.2 und jetzt noch die Copy-&Swap-Implementierung. In der Praxis sollten Sie, wenn es die Umstände erlauben, die Copy-&Swap-Implementierung bevorzugen.

Standardcontainer machen das Leben einfacher

Um hier zwischendurch wieder mein Mantra einzubringen: Mit Standardcontainern, wie in diesem Beispiel `vector<int>`, können Sie sich das Leben im Gegensatz zu dynamischen Eigenschaften erheblich einfacher machen.

8.2.1 Zuweisungsoperator mehrfach überladen

Der Zuweisungsoperator für eine Klasse wird allerdings nicht nur für dynamische Eigenschaften verwendet. In der Praxis wird dieser Operator auch gerne verwendet, um einfache Datentypen einer bestimmten Eigenschaft im Objekt zuzuweisen, bspw.:

```
01 Sales book01;
02 book01 = 1000;
```

Hier wird dem Objekt `Sales` der Wert 1000 zugewiesen. Hierfür reicht es aus, den Zuweisungsoperator entsprechend zu implementieren oder, wenn bereits eine Version vorhanden ist, zu überladen. Der folgende Codeausschnitt soll dies demonstrieren:

```
00 // kapitel08/overloading06.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
06         paperback(p), ebook(e) {}
07     Sales& operator=(const Sales& rhs);
08     Sales& operator=(unsigned int val);
...
08 };

09 Sales& Sales::operator=(const Sales& rhs) {
10     if(this != &rhs) {
11         paperback=rhs.paperback;
12         ebook=rhs.ebook;
13     }
14     return *this;
15 }

16 Sales& Sales::operator=(unsigned int val) {
17     paperback=val;
18     ebook=0;
19     return *this;
20 }
...
21 Sales book01(200, 10);
22 Sales book02, book03;
23 book02 = book01; // operator=(const Sales&)
24 book03 = 300;    // operator=(unsigned int)
```

In Zeile **06** sehen Sie die erste Überladung des Standard-Zuweisungsoperators, die Sie sich allerdings in diesem Beispiel auch sparen könnten, weil die Standardversionen des Compilers hier schon gute Arbeit verrichten und unsere Klasse keine speziellen

Eigenschaften enthält, welche diese Implementierung rechtfertigen. Trotzdem wurde diese hier zur Demonstration implementiert. Die Definition finden Sie in den Zeilen 09 bis 15.

In Zeile 07 finden Sie eine erneute Überladung des Zuweisungsoperators, womit jetzt eine Zuweisung wie `book01=1000` möglich wird. Die Implementierung davon finden Sie in den Zeilen 16 bis 20, und es sollte Ihnen eigentlich keine Probleme mehr bereiten, sie zu verstehen.

In den Zeilen 23 und 24 finden Sie jeweils ein Beispiel, wo der überladene `operator=()` einmal mit `Sales` und einmal mit `unsigned int` verwendet wird.

8.2.2 Zuweisung verbieten (C++11)

Nicht immer ist erwünscht, eine Zuweisung von Objekten zu einem anderen Objekt zu erlauben. Auch ohne eine manuelle Implementierung stellt Ihnen der Compiler eine Standardversion zur Verfügung, welche dann Eigenschaft für Eigenschaft kopiert. Im alten C++03-Standard können Sie dieses Verhalten unterbinden, indem Sie die Deklaration des zu überladenden Zuweisungsoperators in den `private`-Bereich der Klassendefinition schreiben. Private Mitglieder sind von außen nicht zugänglich, und somit können auch keine Objekte dieser Klasse anderen Objekten derselben Klasse mit dem Zuweisungsoperator zugewiesen werden.

Im neuen C++11-Standard können Sie es sich einfacher machen und am Ende der Deklaration diese Verwendung einfach mit `=delete` verbieten (siehe auch Abschnitt 7.9.2, »delete«), bspw. so:

```
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
06         paperback(p), ebook(e) {}
07     Sales& operator=(const Sales& rhs) = delete;
08     Sales(const Sales& rhs) = delete;
09     ...
10 };
```

In der C++11-Praxis können die neue und effizientere verschiebende Zuweisung und der neue verschiebende Kopierkonstruktor statt der kopierenden Versionen verwendet werden. Hier macht es dann häufig Sinn, die kopierende Zuweisung (so auch den Kopierkonstruktor) mit `delete` zu verbieten (wie in den Zeilen 06 und 07 zu sehen ist), um den Anwender zu zwingen, die verschiebenden Versionen zu verwenden.

Die verschiebende Zuweisung (C++11)

Die (Standard-)Verschiebezweisung von C++11 haben Sie bereits im Abschnitt »Verschiebekonstruktor (C++11)« in Abschnitt 7.4.2 kennengelernt. Auch in der Praxis haben Sie diesen verschiebenden Zuweisungsoperator bereits in Abschnitt 7.10, »Klassen für das Verschieben schreiben (C++11)«, gesehen. Mehr über das neue Konzept können Sie außerdem in Abschnitt 16.1, »Move-Semantik und Perfect Forwarding«, nachlesen.

8.3 Die Operatoren im Schnelldurchlauf

Vom Prinzip her haben Sie jetzt erfahren, wie Sie überladbare Operatoren implementieren können. In den folgenden Abschnitten werden alle überladbaren Operatoren nochmals kurz vorgestellt und deren übliche Verwendungen (Semantik), Deklarationen, mögliche Implementierungen und Besonderheiten beschrieben. Wohlgermerkt, hierbei handelt es sich lediglich um Empfehlungen, wie diese Operatoren überladen und implementiert werden sollten/könnten. Letztendlich steht es Ihnen frei, den Operator mit einer anderen Semantik zu überladen. Am Ende des Kapitels finden Sie noch eine umfassende Tabelle, mit allen Operatoren und deren Möglichkeiten im Überblick.

8.3.1 Der Zuweisungsoperator operator=

Der Zuweisungsoperator hat nun schon besonders viel Aufmerksamkeit in diesem Buch erhalten, weshalb er hier noch nur steckbriefartig behandelt wird (7.7.9). Mehr über diesen Operator können Sie im Abschnitt »(Standard-)Kopierkonstruktor« in Abschnitt 7.4.2, in den Abschnitten 7.8.2, »Roher Zeiger als Element in einer Klasse«, und 8.2, »Zuweisungsoperator – operator=«, nachlesen.

Typische Verwendung: `obj02 = obj01`

Hiermit werden gewöhnlich die Eigenschaften von `obj01` nach `obj02` kopiert.

Deklaration:

```
C& C::operator=(
    C const& op2 );
```

Definition:

```
C& C::operator=(C const& op2)
{
    if(this != &op2)
    {
        // Elementweise kopieren
        // oder (besser)
        // Kopieren und Tauschen
    }
    return *this;
}
```


Anmerkungen:

- ▶ Der Operator darf nur als Methode einer Klasse implementiert werden.
- ▶ Sollte überladen werden, wenn die Klasse Eigenschaften wie einen Zeiger oder andere spezielle Ressourcen enthält. Muss dieser Operator überladen werden, müssen häufig auch noch der Kopierkonstruktor und der Destruktor implementiert werden (Stichwort *die großen Drei*).
- ▶ Wird dieser nicht überladen, stellt der Compiler eine Standardversion zur Verfügung, welche ausreicht, wenn keine speziellen Eigenschaften (wie bspw. Zeiger) in der Klasse vorhanden sind. Allerdings wird keine automatische Version erstellt, wenn die Klasse konstante, nichtstatische Eigenschaften enthält.
- ▶ Wollen Sie eine Zuweisung verbieten, können Sie die Deklaration entweder im `private`-Bereich schreiben oder in C++11 mit der Zuweisung von `=delete` bei der Deklaration verbieten.
- ▶ Kann entweder »manuell« implementiert werden, um Eigenschaft für Eigenschaft zu kopieren, oder mit einer Kopieren-&-Tauschen-Semantik. Wenn möglich sollten Sie Kopieren & Tauschen bevorzugen.

8.3.2 Binäre Operatoren – `operator+`, `-`, `*`, `/`, `%`

Auf die allgemeine Überladung von binären Operatoren wurde bereits in den Abschnitten 8.1.1, »Binäre Operatoren«, und 8.1.2 »Binäre Operatoren als (friend-)Funktion überladen«, umfassend eingegangen. Zwar wurde hier ausschließlich der binäre `operator+` behandelt, aber alles dort Beschriebene gilt auch für die Operatoren `operator-`, `operator*`, `operator/` oder `operator%`.

Im Grunde wird eben nach einer Addition, Subtraktion, Multiplikation, Division oder einem Modulo ein neues Objekt mit dem Rückgabergebnis beider Objekte erzeugt. Der rechte Operand wird gewöhnlich nicht verändert. In der Praxis werden neben diesen binären Operatoren auch noch die erweiterten Versionen mit `operator+=`, `operator-=`, `operator*=`, `operator/=` oder `operator%=` implementiert, um auch die vereinfachte kurze Schreibweise von bspw. `obj01 = obj01 + obj02` mit `obj01+=obj02` anzubieten, wenn es denn sinnvoll erscheint.

Steht auf der linken Seite des binären Operators ein einfacher Datentyp (bspw. `obj01 = 1000 + obj01`) als Operand, müssen Sie eine freie (friend-)Funktion implementieren.

8.3.3 Erweiterte Schreibweise binärer Operatoren (`+=`, `-=`, `*=`, `/=`, `%=`)

Die erweiterte Form der binären Operatoren wie `obj02 += obj01` ist gleichwertig zu `obj02 = obj02 + obj01`, nur wird hier `obj02` nicht zweimal ausgewertet und im Gegensatz zur einfachen binären Version von `operator+` müssen Sie bei `operator+=` kein neues Objekt anlegen und zurückgeben, sondern können mit einer Referenz arbei-

ten, weil hierbei der linke Operand geändert wird. Bei `operator+` mussten Sie ja noch davon ausgehen, dass jemand den Operator mit Klasse `obj03 = obj01+obj02` verwendet – bzw. das ist ja auch die eigentliche Semantik von `operator+`. Da es hierbei nun das eigentliche Ziel ist, den linken Operanden zu ändern, ist eine Implementierung in Form einer freien Hilfsfunktion hier recht sinnfrei. Der rechte Operand wird hierbei in der Regel nicht verändert.

Die typische Deklaration sieht demnach wie folgt aus:

```
C& C::operator+=(C const& op2);
```

Bei der Implementierung des Operators werden gewöhnlich Eigenschaften des Objekts gemäß dem entsprechenden Operator verändert, und am Ende wird eine Referenz auf `*this` zurückgegeben.

Hierzu die Implementierung des `+=`-Operators in der Klasse `Sales` zur Demonstration:

```
00 // kapitel08/overloading07.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
06         paperback(p), ebook(e) {}
07     Sales operator+(const Sales& op2) const;
08     Sales& operator+=(const Sales& op2);
...
08 };

09 Sales Sales::operator + (const Sales& op2) const {
10     Sales tmp(*this);
11     tmp+=op2; // Verwendet Sales::operator+=()
12     return tmp;
13 }

14 Sales& Sales::operator += (const Sales& op2) {
15     paperback += op2.paperback;
16     ebook += op2.ebook;
17     return *this;
18 }
...
19 Sales book01(200, 10);
20 Sales book02(300, 30);
21 book01+=book02;
```

Die Deklaration finden Sie in Zeile 07 und die Definition in den Zeilen 14 bis 18. Zur Demonstration wurde hier auch der `operator+` implementiert, womit wir uns bei der Definition in Zeile 11 die Arbeit, Eigenschaft für Eigenschaft zu kopieren, sparen und diese gleich dem überladenen `operator+=` übergeben, der dies ja in den Zeilen 15 und 16 ohnehin erledigt. Wichtig ist auch der Rückgabewert von `*this` in Zeile 17, womit Sie das aufrufende Objekt (linker Operand) mit seinen neuen Werten (hier addiert mit den Eigenschaften des rechten Operanden) zurückgeben. Auf diese Weise können Sie sich doppelte Tipparbeiten sparen. Daher wurde in Abschnitt 8.3.2 zuvor erwähnt, dass neben der einfachen binären Version häufig auch die erweiterte Version implementiert wird.

Wie immer gilt hier alles Beschriebene auch für die Operatoren `operator-=`, `operator*=`, `operator/=` und `operator%=`.

8.3.4 Unäre Operatoren (+, -)

Die beiden unären Operatoren werden gewöhnlich verwendet, um die Vorzeichen der Eigenschaften eines Objekts umzuwandeln. Die Implementierung erfolgt bei diesem Operator gewöhnlich als Methode einer Klasse und wird genauso realisiert, wie die Präfixschreibweise von `++` bzw. `--`, die in Abschnitt 8.1.3, »Unäre Operatoren«, bereits beschrieben wurde. Die übliche Deklaration sieht hierbei wie folgt aus:

```
C C::operator-() const;
```

Als Rückgabewert erhält man ein neues Objekt mit geändertem Vorzeichen. Das eigentliche Objekt (rechter und alleiniger Operand) bleibt unverändert.

Die Implementierung ist einfach: Es wird eine Kopie vom aufrufenden Objekt erstellt, die Eigenschaften entsprechend dem Operator geändert und dann das neue Objekt zurückgegeben. Hierzu das Beispiel zur Demonstration mit unserer Klasse `Sales`:

```
00 // kapitel08/overloading08.cpp
...
01 class Sales {
02     int paperback;
03     int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
        paperback(p), ebook(e) {}
06     Sales operator-() const;
...
07 };

08 Sales Sales::operator-() const {
```

```

09 Sales tmp(*this);
10 tmp.paperback = -paperback;
11 tmp.ebook = -ebook;
12 return tmp;
13 }
...
14 Sales book01(1000, 500);
15 Sales book02 = -book01; // book02(-1000, -500)

```

Damit hier überhaupt ein Vorzeichenwechsel möglich wird, wurden die Eigenschaften in den Zeilen 02 und 03 von `unsigned` auf ein `signed Integer` geändert. In der Implementierung in den Zeilen 08 bis 13 legen wir zunächst eine Kopie an, ändern das Vorzeichen der beiden Eigenschaften in den Zeilen 10 und 11 gemäß dem Operator (hier `operator-`) und geben die Kopie am Ende an den Aufrufer zurück. In Zeile 15 können Sie den überladenen `operator-` im Einsatz sehen, wo beide Attribute von `book01` in `book02` als negative Werte übergeben werden. Analog funktioniert die Überladung auch mit dem `operator+`.

8.3.5 Bitweise Operatoren (& (binär), |, ^)

Die bitweisen Operatoren lassen sich wie schon die binären Operatoren implementieren (siehe Abschnitt 8.1.1, »Binäre Operatoren«, und 8.3.2, »Binäre Operatoren – `operator+`, `-`, `*`, `/`, `%`«). Allerdings ist es recht schwierig, eine sinnvolle Verwendung dafür zu finden – hängt auch von der Klasse ab. In der Praxis sieht man die bitweisen Operatoren allerdings eher selten als Operatorüberladung implementiert.

8.3.6 Erweiterte Schreibweise bitweiser Operatoren (&=, |=, ^=)

Hier gilt dasselbe wie eben in Abschnitt 8.3.5 zu den bitweisen Operatoren geschrieben wurde. In der Praxis findet man diese Operatoren eher selten als Überladung einer Klasse. Wollen Sie trotzdem einen bitweisen Operator in der erweiterten Schreibweise implementieren, können Sie sich an Abschnitt 8.3.3, »Erweiterte Schreibweise binärer Operatoren (`+=`, `-=`, `*=`, `/=`, `%=`)«, orientieren.

8.3.7 Logische Operatoren (==, !=)

Die Operatoren `operator==` und `operator!=` dürften wohl zu den am häufigsten überladenen Operatoren gehören. Der Sinn und Zweck der beiden Operatoren ist auch fast immer intuitiv und dient dazu, zwei Objekte auf Gleichheit bzw. Ungleichheit hin zu testen. Zwei Objekte können als gleich gelten, wenn Sie dieselben Zustände enthalten – also gleichwertige Eigenschaften. Natürlich hängt dieser Vergleich u.a. wieder von der Klasse und den darin verwendeten Eigenschaften ab.

Beide Operatoren werden in der Praxis meistens als freie Hilfsfunktionen überladen und liefern, wie es für einen Vergleich üblich ist, `bool` als Typ zurück. Da bei Vergleichen beide Operanden nicht verändert werden, können diese mit `const` gekennzeichnet werden. Hier die übliche Deklaration als freie Funktion:

```
bool operator==(const C& op1, const C& op2);
```

Oder die alternative Version als klasseneigene Methode:

```
bool C::operator==(const C& op2);
```

Bei der Implementierung werden beide Objekte gewöhnlich Eigenschaft für Eigenschaft auf Gleichheit überprüft. Wenn Sie den Vergleich als freie Hilfsfunktion implementieren, sind die Eigenschaften `private` (was ja eigentlich immer der Fall ist), und Sie müssen die Funktion in der Klasse als `friend` deklarieren. Hierzu die Implementierung beider Vergleichsoperatoren in der Praxis:

```
00 // kapitel08/overloading09.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
        paperback(p), ebook(e) {}
06     friend bool operator==( const Sales& op1,
                            const Sales& op2);
07     bool operator!=(const Sales& op2);
...
08 };

09 bool operator==(const Sales& op1, const Sales& op2) {
10     return ( (op1.paperback == op2.paperback) &&
        (op1.ebook == op2.ebook ) );
11 }

12 bool Sales::operator!=(const Sales& op2) {
13     return ( ! ((paperback == op2.paperback) &&
        (ebook == op2.ebook ) ) );
14 }
...
15 Sales book01(1000, 500);
16 Sales book02(book01);
17 Sales book03(444, 555);
```

```

18 if( book01 == book02 ) {
19     // Beide Elemente sind gleich
20 }

21 if( book01 != book03 ) {
22     // Beide Elemente sind nicht gleich
23 }

```

In diesem Beispiel wurde einmal die eine freie (friend-)Funktion (Zeilen 09 bis 11) mit dem `operator==` und einmal eine klasseneigene Version mit `operator!=` (Zeilen 12 bis 14) implementiert. Ansonsten dürfte hier jetzt nichts überraschend Neues dabei sein. Ein Tipp vielleicht noch, womit Sie sich beim `operator!=` den Vergleich der einzelnen Eigenschaften sparen können. Sie verwenden einfach den bereits überladenen `operator==` als Vergleich und negieren den Rückgabewert. Hierzu die etwas elegantere Version von `operator!=`:

```

12 bool Sales::operator!=(const Sales& op2) {
13     return !(*this == op2);
14 }

```

Eine Frage bleibt hierbei allerdings noch im Raum stehen. Die Logik in diesem Fall war es ja, nachzuprüfen, ob zwei Objekte denselben Inhalt haben oder nicht. Was ist jetzt aber, wenn Sie wissen wollen, ob es sich nicht um dasselbe Objekt handelt? Ein Beispiel:

```

01 Sales book01(1000, 500);
02 Sales *ptrBook = &book01;

03 if( book01 == *ptrBook ) {
04     std::cout << "Gleiches Objekt!!!\n";
05 }

```

Hier wurde dem Zeiger `ptrBook` in Zeile 02 die Adresse von `book01` zugewiesen. Im Vergleich in Zeile 03 wollen wir jetzt aber nicht wissen, ob die Eigenschaften identisch sind, sondern ob wir hier nicht mit demselben Objekt mit der gleichen Adresse hantieren. Wollen Sie den Sinn des `operator==` so verwenden, müssen Sie die Implementierung etwas anpassen:

```

01 bool operator==(const Sales& op1, const Sales& op2) {
02     return &op1 == &op2;
03 }

```

In diesem Fall gibt jetzt ein Vergleich mit `operator==` nur dann `true` zurück, wenn beide Operanden dieselbe Adresse (!) haben. Zwei Objekte mit nur identischen Eigenschaften werden hier jetzt `false` zurückgegeben.

8.3.8 Die logischen Verknüpfungen (&&, || und !)

Den logischen unären `operator!` haben Sie bereits in Abschnitt 8.1.4, »Unäre Operatoren als (friend-)Funktion überladen«, kennengelernt, wo Sie ihn als freie (friend-) Hilfsfunktion überladen haben. In der Praxis implementiert man diesen Operator allerdings eher als Methode einer Klasse. Sinn und Zweck des `operator!` ist die Negation bzw. Verneinung oder auch Überprüfung, ob die Eigenschaften eines Objekts nicht »leer« sind. Die Deklaration der Klassenversion von `operator!` sieht für gewöhnlich wie folgt aus:

```
bool C::operator!() const;
```

In der Implementation müssen Sie lediglich die einzelnen Eigenschaften vergleichen und zurückgeben. Bezogen auf die Klasse `Sales` sieht die Implementation wie folgt aus:

```
01 bool Sales::operator!() const {
02     return (paperback==0) && (ebook==0);
03 }
```

Die anderen beiden logischen Verknüpfungen von `operator||` und `operator&&` werden wie schon die binären Operatoren (siehe Abschnitt 8.1.1, »Binäre Operatoren«) realisiert. In der Praxis sollten beide Operatoren auch in der semantischen Bedeutung einer booleschen UND- bzw. ODER-Verknüpfung implementiert werden. Die Semantik von ODER besagt, dass wenn bei einer Auswertung des ersten Operanden das Ergebnis bereits `true` ist, der zweite Operand nicht mehr ausgewertet wird. Bei UND müssen beide Operanden `true` sein. Leider wird diese Semantik bei der Überladung von logischen Operatoren mit eigenen Datentypen (wie einer Klasse) nicht korrekt übernommen. In der Praxis sieht man daher diese Operatoren relativ selten überladen. Wenn Sie diesen Operator trotzdem implementieren wollen, sollten Sie dies als freie (friend-)Hilfsfunktion tun, um eine mögliche implizite Konvertierung des ersten Arguments zu vermeiden.

8.3.9 Vergleichsoperatoren (<, <=, >=, >)

Der Sinn und Zweck dieser Operatoren ist der Vergleich von Objekten auf kleiner, größer usw. (Ordnungsrelation) zu testen. Auch hier wird in der Praxis gerne auf freie (friend-)Hilfsfunktionen zurückgegriffen, um eine implizite Konvertierung des linken Operanden zu umgehen.

Diese Operatoren können Sie, wie schon die Vergleichsoperatoren `operator==` und `operator!=` deklarieren und implementieren. Trotzdem gibt es bei diesen Operatoren eine Besonderheit, worauf ich Sie unbedingt hinweisen will, für den Fall, dass Sie diese Operatoren überladen wollen. Von der Logik her ist es nämlich völlig ausreichend, nur

die Vergleichsoperatoren `operator<` und `operator==` zu überladen. Alle anderen Vergleichsoperatoren können dann damit nachgebildet werden. Der folgende Codeausschnitt unserer einfachen Klasse `Sales` demonstriert Ihnen diesen Fall:

```

00 // kapitel08/overloading10.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
        paperback(p), ebook(e) {}
06     friend bool operator<(const Sales& op1,
                           const Sales& op2);
07     bool operator==(const Sales& op2) const;
08     bool operator>(const Sales& op2) const;
09     bool operator>=(const Sales& op2) const;
10     bool operator<=(const Sales& op2) const;
11     bool operator!=(const Sales& op2) const;
...
12 };

13 bool Sales::operator==(const Sales& op2) const {
14     return ( (paperback + ebook) ==
              (op2.paperback + op2.ebook) );
15 }

16 bool operator<(const Sales& op1, const Sales& op2) {
17     return ( (op1.paperback + op1.ebook) <
              (op2.paperback + op2.ebook) );
18 }

19 bool Sales::operator>(const Sales& op2) const {
20     return ( ! (*this < op2) && !(*this == op2) );
21 }

22 bool Sales::operator>=(const Sales& op2) const {
23     return ( ( (*this == op2) || !(*this < op2) ) );
24 }

25 bool Sales::operator<=(const Sales& op2) const {
26     return ( ( (*this == op2) || (*this < op2) ) );
27 }

```



```

28 bool Sales::operator!=(const Sales& op2) const {
29     return ( !(*this == op2) );
30 }

```

Zugegeben, hier ist der Fall recht einfach, weil unsere Klasse gerade mal zwei Eigenschaften hat. Aber bei aufwendigeren Klassen sparen Sie sich hiermit eine Menge Tipparbeit. Im Beispiel wurden zunächst die beiden Vergleichsoperatoren `operator==` (als Methode) und `operator<` (als freie Hilfsfunktion) implementiert. Alle anderen Operatoren wurden dann in den Zeilen 19 bis 30 nur noch auf Grundlage dieser beiden Operatoren überladen. Allerdings müssen Sie hier ziemlich aufmerksam sein, die richtige logische Verknüpfung zu formulieren. Bei `operator>` bspw. prüfen wir in Zeile 20, ob der Wert des rechten Operanden nicht kleiner als der rechte Operand UND der linke Operand nicht gleich dem rechten Operanden ist. Trifft dies zu, ist der linke Operand größer als der rechte Operand, und diese Überladung liefert `true` zurück. Die internen Vergleiche der Eigenschaften übernehmen jetzt nur noch die Operatoren `operator==` und `operator<`.

Hier muss noch hinzugefügt werden, dass wir die Semantik des Vergleichs jetzt in `operator==` und `operator<` so implementiert haben, dass beide Eigenschaften zusammenaddiert werden (`paperback+ebook`) und somit die Gesamtanzahl der verkauften Büchern verglichen wird. Wir vergleichen hier also nicht mehr Eigenschaft für Eigenschaft, wie wir dies zuvor in Abschnitt 8.3.7, »Logische Operatoren (`==`, `!=`)«, mit den Vergleichsoperatoren `==` und `!=` gemacht haben.

operator== und operator< reichen aus

Im Grunde können Sie sich im Beispiel aus Listing *overloading10.cpp* das Ausprogrammieren aller Vergleichsoperatoren auch sparen und sich auf `operator==` und `operator<` beschränken, wie es auch bspw. viele Algorithmen der Standardbibliothek machen. An vielen Stellen verlangt die Standardbibliothek, dass Objekte »vergleichbar« sind – dabei bemüht sich die Bibliothek aber, nicht alle Vergleichsoperatoren zu verlangen. Vielmehr sind die Algorithmen so programmiert, dass man mit `operator==` und `operator<` auskommt.

In unserem Beispiel könnten Sie etwa bis auf `operator==` und `operator<` alle anderen Vergleichsoperatoren streichen und dann den Namensraum `std::rel_ops` aus der Headerdatei `<utility>` einbinden (siehe Abschnitt 12.6.3, »Vergleichsoperatoren für eigene Typen«). In diesem Namensraum sind Funktions-Templates enthalten, welche aus dem `operator==` das Verhalten (`!=`) und aus dem `operator<` das Verhalten von `>`, `<=` und `>=` ableiten:

```

00 // kapitel08/overloading10_2.cpp
01 #include <utility>
02 using namespace std::rel_ops;
...

```

8.3.10 Inkrement- und Dekrement-Operator (++ , --)

Die Überladung beider unärer Operatoren wurde bereits in Abschnitt 8.1.3, »Unäre Operatoren«, ausführlich demonstriert, so dass ich mir eine erneute Ausführung hier sparen kann.

8.3.11 Die Operatoren operator*, operator-> und operator-*

Die folgenden Operatoren stehen im Zusammenhang mit rohen Zeigern und werden hauptsächlich für selbst erstellte Containerklassen als Iterator verwendet (siehe Abschnitt 12.4, »Iteratoren«). Ein Iterator bietet Zugriff auf den Inhalt der Daten im Container. Im Gegensatz zum nackten Zeiger wird hierbei allerdings der komplexe rohe Zugriff versteckt, so dass der Anwender kaum noch den rohen Zeiger mit * verwenden muss. Alle Standardcontainer von C++ bieten solche Iteratoren an und verstecken somit den komplexen internen Zugriff auf die Container. Man sagt auch, die Dereferenzierung erfolgt über die Dereferenzierung des eingebetteten Zeigers. Neben Iterator-Klassen könnten diese Operatoren auch in Verbindung mit Smart Pointer-Klassen verwendet werden.

Die Motivation, diese Operatoren zu überladen, ist es vielleicht, eine eigene Bibliothek zu erstellen. In der Praxis bietet C++ mit der Standardbibliothek längst fertige und lang erprobte Sachen an. In der Praxis werden Sie daher eher selten in den »Genuss« kommen, rohe Zeigersachen auf einer solch tiefen Ebene zu überladen – es sei denn, Sie haben ein Hang dazu, das Rad immer wieder neu zu erfinden.

8.3.12 Der Adressoperator &

Die übliche Semantik für den Operator liegt darin, die Adresse eines Objekts zurückzuliefern. In der Praxis kann man diesen Operator vielleicht nützlich überladen, wenn man einen Smart Pointer speichern will und bei Aufruf des Adressoperators automatisch die Adresse der referenzierten Daten zurückgibt. Persönlich würde ich allerdings vom Überladen des Adressoperators abraten, weil es damit dann nicht mal mehr möglich ist, die Adresse zweier Objekte zu vergleichen, bzw. weil er ein unerwartetes Verhalten zurückliefert. Bedenken Sie, dass Sie mit einer Überladung dieses Operators in das Zeigerkonzept von C++ eingreifen und somit die komplette Verantwortung der Speicherung und der Abfrage von Daten übernehmen.

8.3.13 Der Komplementoperator ~

Dieser Operator hat ja die Semantik einer komponentenweisen booleschen Verneinung und könnte verwendet werden, um ein Objekt mal schnell zwischen zwei Zuständen hin- und herzuschalten. Allerdings wird auch dieser Operator eher selten überladen.

8.3.14 Ein- und Ausgabeoperatoren – `operator>>` und `operator<<`

Viele (Standard-)Klassen bieten die Möglichkeit an, die Stream-Objekte `std::cin`, `std::cout` und `std::cerr` über die Operatoren `>>` und `<<` zu nutzen. So ist es bspw. nicht verwunderlich, dass auch für die meisten Standardcontainer-Klassen (siehe Abschnitt 12.2, »Standardcontainer-Klassen«) die Streaming-Operatoren `operator>>` und `operator<<` für die Ein- und Ausgabe verwendet werden können.

Die bitweisen Shift-Operatoren `<<` und `>>`

Beide Operatoren können auch für die bitweise Verschiebung als Methode einer Klasse implementiert werden. Allerdings werden die Shift-Operatoren in der Praxis kaum überladen, weshalb hier nicht näher darauf eingegangen wird.

Anfangen wollen wir mit der Eingabe über das Stream-Objekt `std::cin`, das ein Objekt der Klasse `std::istream` ist, welche wiederum in der Headerdatei `<iostream>` definiert ist. Die übliche Deklaration, um `operator>>` für `std::cin` zu überladen, sieht wie folgt aus:

```
std::istream& operator>>(std::istream& lop, C& rop);
```

Über den linken Operanden, das Stream-Objekt `std::cin` (hier `lop`), werden die Daten über den Operator `operator>>` in das Objekt des rechten Operanden (hier `rop`) geschoben. Da der rechte Operand die Daten erhält, darf dieser nicht konstant sein. Weil `std::cin` ein Objekt der Klasse `std::istream` ist und Sie in der Regel auf private Daten der rechten Operanden der Klasse `C` zugreifen wollen, müssen Sie die Überladung als freie friend-Funktion implementieren. Die Rückgabe des Stream-Objekts `std::istream` ermöglicht hier außerdem auch die typische Verkettung von `std::cin >> obj01 >> obj02`.

Ähnlich lässt sich auch das Gegenstück für die Ausgabe überladen, nur dass Sie hierbei `operator<<` überladen müssen. Für die Ausgabe benötigen Sie die Klasse `std::ostream`, welche sich ebenfalls im Header `<iostream>` befindet. Die Deklaration, um `operator<<` für `std::cout` zu überladen, kann wie folgt aussehen:

```
std::ostream& operator<<(std::ostream& lop, const C& rop);
```

Hier werden vom rechten Operanden (hier `rop`) der Klasse `C` die Daten über den Operator `operator<<` in das Stream-Objekt `lop` (gewöhnlich `std::cout`) zur Ausgabe auf dem Bildschirm geschoben. Da der rechte Operand bei der Ausgabe nicht geändert wird, kann dieser konstant sein. Da Sie hier außerdem im rechten Operanden auf private Daten treffen könnten, sollten Sie auch diese Operatorüberladung als freie friend-Funktion implementieren. Auch hier ist dank der Rückgabe von `std::ostream` eine Verkettung wie `std::cout << obj01 << obj02` möglich.

Lesen und Schreiben von Dateien

Wohlgemerkt, die beiden Operatoren `operator>>` und `operator<<` lassen sich, so implementiert, nicht nur für die Stream-Objekte `std::cin` bzw. `std::cout` verwenden. Statt `std::cin` können Sie hier genauso gut `std::ifstream` verwenden, um etwas in eine Datei statt in die Standardausgabe zu schreiben. Analog gilt Gleiches auch für `std::cout`, wo Sie mit `std::ofstream` jederzeit etwas aus einer Datei lesen könnten. Vorausgesetzt natürlich, Sie haben einen Datei-Stream geöffnet. Mehr dazu erfahren Sie in Abschnitt 13.6, »Streams für die Datei-Ein-/Ausgabe«.

Im folgenden Beispiel sehen Sie eine mögliche Implementierung von `operator>>` und `operator<<` für die Klasse `Sales`, um die Eigenschaften der Klasse über `std::cin` einzulesen bzw. über `std::cout` wieder auszugeben:

```
00 // kapitel08/overloading11.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
06         paperback(p), ebook(e) {}
07     friend std::istream& operator>>(std::istream& lop, Sales& rop);
08     friend std::ostream& operator<<(std::ostream& lop, const Sales& rop);
...
08 };

09 std::istream& operator>>(std::istream& lop, Sales& rop) {
10     std::cout << "Paperback : "; lop >> rop.paperback;
11     std::cout << "Ebook      : "; lop >> rop.ebook;
12     return lop;
13 }

14 std::ostream& operator<<(std::ostream& lo, const Sales& ro ) {
15     lo << ro.paperback << "/" << ro.ebook;
16     return lo;
17 }
...
18 Sales book01;
19 std::cin >> book01;
20 // operator>>(std::cin, book01)
21 std::cout << book01 << std::endl;
22 // operator<<(std::cout, book01)
```

In den Zeilen **06** und **07** werden beide Operatorüberladungen als freie friend-Funktionen deklariert.

Wenn Sie in Zeile **19** den `operator>>` aufrufen, wird die Überladung in den Zeilen **09** bis **13** ausgeführt. Als Parameter für `lop` erhält die Funktion das Stream-Objekt `std::cin` (den linken Operanden) und das Objekt `book01` für den Parameter `rop` als rechten Operanden. Alternativ kann der Operator auch wie in der auskommentierten Zeile **20** aufgerufen werden. Da `lop` nun mit dem Stream-Objekt `std::cin` verbunden ist werden jetzt in den Zeilen **10** und **11** jeweils die Werte von der Standardeingabe eingelesen und in die privaten Daten des Objekts geschoben. Zurückgegeben wird am Ende immer der Eingabe-Stream (Zeile **12**).

Ähnlich verläuft dies dann auch mit dem überladenen `operator<<`, welchen Sie hier in Zeile **21** (oder in alternativer Schreibweise in Zeile **22**) aufrufen. Die Überladung sehen Sie in den Zeilen **14** bis **17** definiert. Hierbei bekommt der Parameter `lo` das Stream-Objekt `std::cout` (linker Operand) und der Parameter `ro` erhält das Objekt vom Typ `Sales`. `lo` ist jetzt mit dem Stream-Objekt `std::cout` verbunden, und daher werden die Eigenschaften des Objekts in Zeile **15** über `<<` an die Standardausgabe geschoben und auf dem Bildschirm ausgegeben. Hier wird am Ende immer der Ausgabe-Stream (Zeile **16**) zurückgegeben.

Statt friend-Funktion auch Delegation möglich

Wollen Sie keine friend-Funktion verwenden, können Sie die Operatoren auch an public-Methoden delegieren, bspw.:

```
// keine friend-Funktion mehr - nur eine freie Funktion
std::ostream& operator<<(std::ostream& lop, const Sales& rop) {
    return rop.output(lop); // Arbeit delegieren
}
// Diese public-Methode von Sales tut jetzt unsere Arbeit
std::ostream& Sales::output(std::ostream& os) const {
    std::cout << paperback << "/" << ebook;
    return os;
}
```

8.3.15 Der Konvertierungsoperator ()

Der Konvertierungsoperator (oder auch Cast-Operator) `()` (nicht zu verwechseln mit dem Funktionsoperator, siehe Abschnitt 8.3.16, »Funktionsoperator `()` – Funktionsobjekte«) ermöglicht Ihnen eine implizite Konvertierung von Objekten einer Klasse

in einen beliebigen anderen Datentyp. Die Deklaration dieser Operatorüberladung sieht wie folgt aus:

```
C::operator TYP() const;
```

Es fällt gleich auf, dass hier kein Rückgabetyt benötigt wird. Dies ist nicht nötig, weil dieser Typ bereits mit TYP vorgegeben wird. Diese Operatorüberladung muss als Methode implementiert werden. Es wird außerdem auch empfohlen, eine `const`-Methode daraus zu machen, um auch `const`-Objekte konvertieren zu können.

Hierzu die Überladung des Konvertierungsoperators zur Demonstration mit anschließender Erläuterung:

```
00 // kapitel08/overloading12.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
        paperback(p), ebook(e) {}
06     operator unsigned int() const;
07     operator bool() const;
...
08 };

09 Sales::operator unsigned int() const {
10     return paperback + ebook;
11 }

12 Sales::operator bool() const {
13     return( (paperback!=0) && (ebook!=0) );
14 }
...
15 Sales book01(2000, 1234);
16 unsigned int gesamt = book01;
17 std::cout << gesamt << std::endl;

18 if( book01 ) std::cout << "Bücher wurden verkauft\n";

19 Sales book02;
20 if(!book02)std::cout << "Keine Bücher wurden verkauft\n";
```

```

21 if( (!book01) || (!book02) ) {
22     std::cout << "Eines der Objekte ist noch leer\n";
23 }

24 Sales book03(1000, 2000);
25 if( book01 && book03 ) {
26     std::cout << "Beide Objekte sind nicht leer\n";
27 }

```

In den Zeilen **06** und **07** sehen Sie die Deklarationen des Konvertierungsoperators. Hierbei wurde jeweils einmal eine Version mit `unsigned int` und eine Version mit `bool` deklariert.

Die Implementierung für den Konvertierungsoperator von `Sales` nach `unsigned int` finden Sie in den Zeilen **09** bis **11** definiert. Die Methode selbst macht nichts anderes, als in Zeile **10** die beiden Eigenschaften zusammenaddiert zurückzugeben. Durch diesen Konvertierungsoperator ist die Verwendung in Zeile **16** möglich, wo Sie ein `Sales`-Objekt einem `unsigned int` zuweisen. Die Variable `gesamt` enthält anschließend die Summe der Eigenschaften `Sales::paperback` und `Sales::ebook` des Objekts `book01`.

Interessanter dürfte vielleicht die zweite Implementierung von `bool` in den Zeilen **12** bis **14** sein. Dort geben wir den booleschen Wert `true` zurück, wenn beide Eigenschaften des aufrufenden `Sales`-Objekts nicht gleich 0 sind – wenn also bereits Bücher verkauft wurden. Dank dieser Überladung kann ein Objekt jetzt in jedem booleschen Ausdruck verwendet werden. In den Zeilen **18** oder **20** wird bspw. tatsächlich der Konvertierungsoperator von `Sales` nach `bool` aufgerufen.

Besser noch, dank der Ausprogrammierung von `bool` können Sie jetzt ganz komfortabel die logischen Operatoren `&&`, `||` und `!` verwenden, ohne diese Operatoren zu überladen (siehe Zeilen **20**, **21** und **24**). In allen Beispielen wird das Objekt in den `if`-Anweisungen in ein `bool` konvertiert und eben auch als boolescher Ausdruck ausgewertet.

Versehentliche Konvertierung verhindern

Wollen Sie eine versehentliche Konvertierung vermeiden, können Sie das Schlüsselwort `explicit` vor die Deklaration setzen:

```
explicit operator unsigned int() const;
```

Durch `explicit` muss jetzt explizit die Schreibweise

```
unsigned int gesamt(book01);
//bzw. in C++11: unsigned int gesamt{book01};
```

verwendet werden.

Achtung vor unerwarteten Seiteneffekten!!!

Das Überladen des Konvertierungsoperators für mehrere Typen kann recht nützlich sein und eine Menge an Tipparbeit ersparen. Allerdings treten hierbei immer wieder überraschende Fehler auf, weil man häufig nicht genau sieht, wo der Compiler vielleicht eine implizite Konvertierung vornimmt und wo man diese vielleicht gar nicht erwartet. Wenn Sie bspw. zuvor ausprobiert haben, die Konvertierung von `unsigned int` mit `explicit` zu vermeiden, so hat Folgendes

```
explicit operator unsigned int() const;
operator bool() const;
...
Sales book01(2000, 1234);
unsigned int gesamt = book01; // = 1 ????
```

dazu geführt, dass `gesamt` der Wert 1 (!) zugewiesen wurde, anstatt sich zu beschweren, dass `Sales` nicht nach `unsigned int` konvertiert werden kann, wie man das eben vom Schlüsselwort `explicit` erwarten würde. Hier hat sich der Compiler eben einfach nach einer anderen möglichen Konvertierung umgesehen und ist mit der Umwandlung nach `bool` fündig geworden, welche wir hier in diesem Beispiel ebenfalls implementiert haben. Zwar gibt unser Konvertierungsoperator nach `bool` entweder `true` oder `false` zurück, aber der Compiler macht implizit daraus 1 oder 0.

8.3.16 Funktionsoperator () – Funktionsobjekte

Auch den Funktionsoperator `()` können Sie überladen – nicht verwechseln mit dem Konvertierungsoperator (siehe Abschnitt 8.3.15). Objekte, welche mit dem Funktionsoperator überladen werden, werden als *Funktionsobjekte* bzw. *Funktoren* bezeichnet. Damit können Sie Objekte der Klasse praktisch wie Funktionen ausführen lassen. Eine übliche Deklaration, welche nur als Methode implementiert werden kann, sieht wie folgt aus:

```
RTYP C::operator()(TYP val1, ...);
```

Anzahl und Typen der Parameter sind frei wählbar. `RTYP` ist der Rückgabotyp des Funktionsobjekts.

Funktionszeiger

Funktionsobjekte sind in C++ so etwas wie die klassischen Funktionszeiger für C.

Das folgende Beispiel zeigt Ihnen eine einfachste Implementierung von Funktionsobjekten:


```

00 // kapitel08/overloading13.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
        paperback(p), ebook(e) {}
06     int operator() (unsigned int books) const;
07     int operator() (Sales& books) const;
...
08 };

09 int Sales::operator()(unsigned int books) const {
10     return paperback+ebook+books;
11 }

12 int Sales::operator()(Sales& books) const {
13     return paperback+ebook+books.paperback+books.ebook;
14 }
...
15 Sales book01(2000, 1000);
16 Sales book02(1000, 500);
17 unsigned int newbooks = 432;

18 std::cout << book02(newbooks) << std::endl;
19 std::cout << book02(book01) << std::endl;

```

In den Zeilen **06** und **07** sehen Sie zwei Funktionsobjekte deklariert. Beide geben `int` als Rückgabewert zurück. Ein Funktionsobjekt erhält als Parameter `unsigned int` und das andere ein `Sales`-Objekt.

Rufen Sie das Funktionsobjekt mit einem `unsigned-int`-Wert auf, wie in Zeile **18** geschehen, bekommen Sie hier zusätzlich zum übergebenen `unsigned-int`-Wert eine Summe aller verkauften Bücher (`paperback+ebook+unsigned-int-Wert`) aus der Implementation in den Zeilen **09** bis **11** zurück.

Ähnliches wurde auch implementiert, wenn das Funktionsobjekt mit einem `Sales`-Objekt aufgerufen wird, wie in Zeile **19**. Hier wird dann in der Implementation in den Zeilen **12** bis **14** die komplette Summe aller Bücher vom aufrufenden Objekt und dem Objekt als Parameter zurückgegeben.

Dies ist natürlich nur ein Beispiel. Sie können auch Funktionsobjekte mit anderen Typen als Rückgabewert oder Parametern verwenden.

8.3.17 Der Indexoperator []

Die Semantik für diesen Operator ist ganz klar ein indizierter Zugriff (Array-Zugriff) wie er bspw. bei den Standardcontainern `std::vector`, `std::map` oder `std::array` (neu in C++11) verwendet wird. Die Deklaration der Überladung, welche nur als Methode einer Klasse implementiert werden kann, sieht wie folgt aus:

```
TYP& C::operator[](TYP);
```

Der TYP ist jeweils frei wählbar. Sie können auch `const`-Versionen davon implementieren. So bietet die Standardbibliothek bspw. oft eine `const`- und eine Nicht-`const`-Version des überladenen `operator[]` an.

Verwenden Sie die Bordmittel von C++

Da im Buch des Öfteren darauf hingewiesen wurde, auf rohe Sachen zu verzichten, fällt dieser Operator in gewisser Hinsicht in diese Kategorie – das hängt auch davon ab, wie Sie diesen implementieren. Sie sollten hier nur hinterfragen, ob Sie wirklich das Rad neu erfinden wollen oder nicht doch die längst erprobten Sachen der Standardbibliothek von C++ verwenden wollen. Selbst assoziative Arrays bietet Ihnen C++ bspw. mit dem Standardcontainer `std::map` längst an. Natürlich ist man als Programmierer immer ein wenig gewillt, etwas »selbst« machen zu wollen, und vielleicht ist dies zum Herumexperimentieren auch ganz gut geeignet. Aber im rauen Programmieralltag greift man dann doch eher auf fertige und erprobte Hausmittel zurück.

Hierzu vielleicht ein Minibeispiel, in dem `operator[]` überladen wurde, um Postleitzahlen anhand des Ortes bzw. Orte anhand der Postleitzahl auszugeben. In der Praxis verwendet man hierfür allerdings den Standardcontainer `std::map`.

```
00 // kapitel08/overloading14.cpp
...
01 class Postleitzahl {
02     unsigned int plz;
03     std::string ort;
04 public:
05     Postleitzahl(unsigned int p=0, std::string o="") :
        plz(p), ort(o) {}
06     int operator[](const std::string& o) {return plz;}
07     std::string& operator[](const unsigned int p) {
08         return ort;
09     }
10 };
...
11 Postleitzahl plz01(86415, "Mering");
```

```
12 std::cout << plz01["Mering"] << std::endl; // = 86415
13 std::cout << plz01[86415] << std::endl;    // = Mering
```

Hier wurde der Operator `[]` zweifach überladen und kann einmal mit `std::string` als Array-Zugriff (assoziiertes Array) aufgerufen werden (siehe Zeile 12) und gibt so vom aufrufenden Objekt die entsprechende Eigenschaft vom Typ `int` zurück – also die Postleitzahl für einen Ort. Die zweite Überladung ist genau umgekehrt, hier wird der Operator `[]` mit einem `int` als Parameter aufgerufen (siehe Zeile 13) und liefert den passenden String für das Objekt zurück – hier somit den Ort für die Postleitzahl.

8.3.18 Die Operatoren `new`, `new[]`, `delete` und `delete []`

Die Operatoren sind für die Speicherreservierung von einzelnen Objekten (`new` und `delete`) und allokierten Arrays (`new[]` und `delete[]`) gedacht und können auch als eigene Operatoren überladen werden. Die Operatoren müssen als statische Methode für eine Klasse überladen werden. Auch wenn Sie die Operatoren nicht explizit mit `static` kennzeichnen, wird der Compiler dies implizit immer so anlegen. Hierzu zunächst die Deklaration für `new` und `delete` für einzelne Objekte:

```
static void* C::operator new (size_t size);
static void C::operator delete(void *ptrMemory);
```

Die Anzahl der zu reservierenden Bytes für den Compiler wird mit `size` beim Operator `new` angegeben. Als Rückgabewert liefert diese Methode den Zeiger auf die Anfangsadresse des reservierten Speichers als rohen typlosen `void`-Zeiger zurück.

Der überladene `delete`-Operator hingegen erhält als Parameter den `void`-Zeiger auf die Anfangsadresse des freizugebenden Speichers, den Sie zuvor mit `new` erhalten haben.

Freie Versionen

Neben der Möglichkeit, `new` und `delete` für Klassen zu überladen, können Sie auch das freie `new` bzw. `delete` überschreiben. Wenn Sie dies allerdings tun, gibt es keine Möglichkeit mehr, auf die vom Compiler bereitgestellten Versionen zuzugreifen. Darauf wird allerdings in diesem Buch nicht eingegangen.

Hierzu ein einfaches Beispiel, welches lediglich `new` und `delete` ohne eine spezielle Funktionalität überlädt:

```
00 // kapitel08/overloading15.cpp
...
01 class Sales {
02     unsigned int paperback;
03     unsigned int ebook;
```

```

04 public:
05     Sales(unsigned int p=0, unsigned int e=0) :
        paperback(p), ebook(e) {}
06     static void* operator new(size_t size);
07     static void operator delete(void* ptrMemory);
...
08 };

09 void* Sales::operator new(size_t size) {
10     void* ptrMemory = new char[size];
11     std::memset(ptrMemory, 0, size); // Initialisieren
12     std::cout << "Sales::new wird ausgeführt\n";
13     return ptrMemory;
14 }

15 void Sales::operator delete(void* ptrMemory) {
16     std::cout << "Sales::delete wird ausgeführt\n";
17     delete [] ptrMemory;
18 }
...
19 Sales *objPtr = new Sales;
...
20 delete objPtr;

```

Bei dieser Version haben Sie keinerlei Vorteile gegenüber den Standardversionen. Allerdings könnten Sie hier statt der Ausgabe auf dem Bildschirm bspw. die Speicherreservierung und Freigabe mit loggen. Damit lassen sich Fehler bei der Speicherreservierung aufspüren.

Sollen `new` und `delete` hingegen für Arrays von Objekten überladen werden, können Sie folgende Deklarationen dafür verwenden:

```

static void* C::operator new[](size_t size);
static void C::operator delete[](void* ptrMemory);

```

Der Unterschied zwischen den Methoden für einzelne Objekte unterscheidet sich im Grunde nur durch den Indexoperator am Ende von `new` bzw. `delete`.

Fehlerbehandlung

Auch die Fehlerbehandlung spielt eine wichtige Rolle, um die Sie sich ebenfalls kümmern müssen. So wirft der Operator `new` bspw. die Ausnahme `std::bad_alloc`, wenn nicht genügend zusammenhängender Speicher reserviert werden konnte.

Es gibt allerdings noch eine weitere Version des Operators `new`, bei dem als zweites Argument `std::nothrow` erwartet wird und statt einer Ausnahme der Nullzeiger im Fall eines Fehlers zurückgegeben wird.

new und delete überladen oder nicht?

Auch hierbei sollten Sie sich im Klaren sein, dass Sie hiermit manuell in die komplexe Speicherverwaltung eingreifen. Daher sollten Sie schon ganz genau wissen, was Sie hier tun. Wird die Überladung fehlerhaft implementiert, ist die Gefahr sehr groß, dass das ganze Programm unbrauchbar bzw. instabil wird. Allein dass man hier schon mit typlosen und rohen `void`-Zeigern hantiert, zeigt, auf welcher tiefen Ebene gearbeitet wird. Der einzige Grund, diese Operatoren zu überladen, kann nur sein, Geschwindigkeit und/oder Speicherplatz zu gewinnen. Aber auch für die Fehlersuche kann es manchmal ganz hilfreich sein, diese Operatoren zu überladen, um Speicherreservierungen und Freigaben zu protokollieren und damit Memory Leaks aufzuspüren.

Hierzu nochmals ein paar Grundregeln, welche Sie einhalten müssen, wenn Sie `new` bzw. `new[]` und `delete` bzw. `delete[]` überladen wollen:

- ▶ Der erste Parameter von `new` muss immer `std::size_t` sein. Ansonsten kann `new` noch weitere Parameter haben. `delete` darf nur einen Parameter vom Typ `void*` haben – kann aber auch einen weiteren Parameter vom Typ `std::size_t` haben.
- ▶ `new` muss immer als statische Methode implementiert werden.
- ▶ Der Rückgabewert von `new` muss immer `void*` sein. `delete` hingegen hat keinen Rückgabewert und muss nur `void` sein.
- ▶ `new` kann mehrfach überladen werden, von `delete` hingegen darf es nur eine Version geben.

std::size_t

Der Typ `std::size_t` wird vom Compiler zur Speicherreservierung verwendet und ist häufig ein typedef vom Typ `unsigned int` oder `unsigned long`.

8.4 Übersicht der Operatoren in einer Tabelle

In Tabelle 8.3 finden Sie eine Übersicht aller Operatoren in der Prioritätsreihenfolge und mit dem Hinweis, ob diese überladbar sind. Ein **+** bei **M** (= Methoden) bedeutet, der Operator kann als Methode überladen werden. Ein **+** bei **F** (Funktion) bedeutet, dass der Operator als freie Funktion überladbar ist. Ein **-** bei **M** oder **F** bedeutet, dass der Operator eben nicht als Methode und/oder Funktion überladbar ist. Bei der Syn-

tax finden Sie eine mögliche Deklaration, wie Sie den Operator überladen können. Die Syntax mit `C::` stellt die Syntax für die Methode einer Klasse dar. Die Syntax ohne den Bereichszugriff `C::` ist die Syntax für die freie (friend-)Funktion.

Operator	Bezeichnung	M	F	Syntax
1. Priorität				
<code>::</code>	Scope-Operator	-	-	-
2. Priorität				
<code>.</code>	Komponenten- zugriff	-	-	-
<code>-></code>	Pfeiloperator	+	-	<code>ptr2obj C::operator->();</code> <code>ptr2obj C::operator->() const;</code>
<code>[]</code>	Indizierungs- operator	+	-	<code>Typ C::operator[] (Typ);</code> <code>Typ C::operator[] (Typ) const;</code>
<code>()</code>	Funktions- aufruf	+	-	<code>Typ C::operator() (Parameter- liste);</code> <code>Typ C::operator() (Parameterliste) const;</code>
<code>++</code>	Post-Inkrement (obj++)	+	+	<code>Typ C::operator++(int);</code> <code>Typ C::operator++(int) const;</code> <code>Typ operator++(Typ, int);</code>
<code>--</code>	Post-Dekre- ment (obj--)	+	+	<code>Typ C::operator--(int);</code> <code>Typ C::operator--(int) const;</code> <code>Typ operator--(Typ, int);</code>
<code>typeid</code>	Typ (RTTI)	-	-	-
<code>const_cast</code>	const-Cast	-	-	-
<code>static_cast</code>	static-Cast	-	-	-
<code>reinterpret_ cast</code>	reinterpreter- Cast	-	-	-
<code>dynamic_cast</code>	dynamic-Cast	-	-	-
3. Priorität				
<code>sizeof</code>	Objektgrößen- Operator	-	-	-

Tabelle 8.3 Übersicht der Operatoren, überladbar oder nicht?

Operator	Bezeichnung	M	F	Syntax
++	Prä-Inkrement (++obj)	+	+	Typ C::operator++(); Typ C::operator++() const; Typ operator++(Typ);
--	Prä-Dekrement (--obj)	+	+	Typ C::operator--(); Typ C::operator--() const; Typ operator--(Typ);
~	bitweises NOT	+	+	Typ C::operator~(); Typ C::operator~() const; Typ operator~(Typ);
!	logisches NOT	+	+	Typ C::operator!(); Typ C::operator!() const; Typ operator!(Typ);
+	Positiv- Operator	+	+	Typ C::operator+(); Typ C::operator+() const; Typ operator+(Typ);
-	Negativ- Operator	+	+	Typ C::operator-(); Typ C::operator-() const; Typ operator-(Typ);
&	Adressoperator	+	+	Typ C::operator&(); Typ C::operator&() const; Typ operator&(Typ);
*	Dereferenz- Operator	+	+	Typ C::operator*(); Typ C::operator*() const; Typ operator*(Typ);
new, delete, new[], delete[]	Speicher- reservierung Speicher- freigabe	+	+	void* operator new(std::size_t); void operator delete(void*); void* operator new[](std::size_t); void operator delete[](void*);
()	C-Cast	-	-	-
4. Priorität				
.*	Feld-Mitglieds- Operator	-	-	
->*	Zeiger-Mit- glieds-Operator	+	+	

Tabelle 8.3 Übersicht der Operatoren, überladbar oder nicht? (Forts.)

Operator	Bezeichnung	M	F	Syntax
5. Priorität				
*	Multiplikation	+	+	Typ C::operator*(Typ); Typ C::operator*(Typ) const; Typ operator*(Typ, Typ);
/	Division	+	+	Typ C::operator/(Typ); Typ C::operator/(Typ) const; Typ operator/(Typ, Typ);
%	Modulo	+	+	Typ C::operator%(Typ); Typ C::operator%(Typ) const; Typ operator%(Typ, Typ);
6. Priorität				
+	Addition	+	+	Typ C::operator+(Typ); Typ C::operator+(Typ) const; Typ operator+(Typ, Typ);
-	Subtraktion	+	+	Typ C::operator-(Typ); Typ C::operator-(Typ) const; Typ operator-(Typ, Typ);
7. Priorität				
<<	Ausgabe bzw. Bit-Shift	+	+	Typ C::operator<<(Typ); Typ C::operator<<(Typ) const; Typ operator<<(Typ, Typ);
>>	Eingabe bzw. Bit-Shift	+	+	Typ C::operator>>(Typ); Typ C::operator>>(Typ) const; Typ operator>>(Typ, Typ);
8. Priorität				
<	Kleiner- Operator	+	+	Typ C::operator<(Typ); Typ C::operator<(Typ) const; Typ operator<(Typ, Typ);
<=	Kleiner-Gleich- Operator	+	+	Typ C::operator<=(Typ); Typ C::operator<=(Typ) const; Typ operator<=(Typ, Typ);
>	Größer- Operator	+	+	Typ C::operator>(Typ); Typ C::operator>(Typ) const; Typ operator>(Typ, Typ);

Tabelle 8.3 Übersicht der Operatoren, überladbar oder nicht? (Forts.)

Operator	Bezeichnung	M	F	Syntax
>=	Größer-Gleich-Operator	+	+	Typ C::operator>=(Typ); Typ C::operator>=(Typ) const; Typ operator>=(Typ, Typ);
9. Priorität				
==	Gleich-Operator	+	+	Typ C::operator==(Typ); Typ C::operator==(Typ) const; Typ operator==(Typ, Typ);
!=	Ungleich-Operator	+	+	Typ C::operator!=(Typ); Typ C::operator!=(Typ) const; Typ operator!=(Typ, Typ);
10. Priorität				
&	bitweises UND	+	+	Typ C::operator&(Typ); Typ C::operator&(Typ) const; Typ operator&(Typ, Typ);
11. Priorität				
^	bitweises XOR	+	+	Typ C::operator^(Typ); Typ C::operator^(Typ) const; Typ operator^(Typ, Typ);
12. Priorität				
	bitweises ODER	+	+	Typ C::operator (Typ); Typ C::operator (Typ) const; Typ operator (Typ, Typ);
13. Priorität				
&&	logisches UND	+	+	Typ C::operator&&(Typ); Typ C::operator&&(Typ) const; Typ operator&&(Typ, Typ);
14. Priorität				
	logisches ODER	+	+	Typ C::operator (Typ); Typ C::operator (Typ) const; Typ operator (Typ, Typ);
15. Priorität				
?:	Bedingungs-operator	-	-	

Tabelle 8.3 Übersicht der Operatoren, überladbar oder nicht? (Forts.)

Operator	Bezeichnung	M	F	Syntax
16. Priorität				
=	Zuweisungsoperator	+	-	Typ C::operator=(Typ); Typ C::operator=(Typ) const;
=	Multiplikationszuweisung	+	+	Typ C::operator=(Typ); Typ C::operator*=(Typ) const; Typ operator*=(Typ, Typ); 1.)
/=	Divisionszuweisung	+	+	Typ C::operator/=(Typ); Typ C::operator/=(Typ) const; Typ operator/=(Typ, Typ); 1.)
%=	Modulo-Zuweisung	+	+	Typ C::operator%=(Typ); Typ C::operator%=(Typ) const; Typ operator%=(Typ, Typ); 1.)
+=	Additionszuweisung	+	+	Typ C::operator+=(Typ); Typ C::operator+=(Typ) const; Typ operator+=(Typ, Typ); 1.)
-=	Subtraktionszuweisung	+	+	Typ C::operator-=(Typ); Typ C::operator-=(Typ) const; Typ operator-=(Typ, Typ); 1.)
<<=	Links-Bit-Verschiebezweisung	+	+	Typ C::operator<<=(Typ); Typ C::operator<<=(Typ) const; Typ operator<<=(Typ, Typ); 1.)
>>=	Rechts-Bit-Verschiebezweisung	+	+	Typ C::operator>>=(Typ); Typ C::operator>>=(Typ) const; Typ operator>>=(Typ, Typ); 1.)
&=	Bitweises-UND-Zuweisung	+	+	Typ C::operator&=(Typ); Typ C::operator&=(Typ) const; Typ operator&=(Typ, Typ); 1.)
=	Bitweises-ODER-Zuweisung	+	+	Typ C::operator =(Typ); Typ C::operator =(Typ) const; Typ operator =(Typ, Typ); 1.)
^=	Bitweises-XOR-Zuweisung	+	+	Typ C::operator^=(Typ); Typ C::operator^=(Typ) const; Typ operator^=(Typ, Typ); 1.)

Tabelle 8.3 Übersicht der Operatoren, überladbar oder nicht? (Forts.)

Operator	Bezeichnung	M	F	Syntax
17. Priorität				
,	Komma	+	+	Typ C::operator,(Typ); Typ C::operator,(Typ) const; Typ operator,(Typ, Typ);
1.) Es gibt Quellen, die besagen, dass die erweiterten Zuweisungsoperatoren nicht außerhalb von Klassen überladen werden können. Im Gegensatz zum Zuweisungsoperator ist dies zwar möglich, aber es wird nicht empfohlen, dies zu tun.				

Tabelle 8.3 Übersicht der Operatoren, überladbar oder nicht? (Forts.)

Zum Weiterlesen: Neu in C++11, der operator""

Neu in C++11 ist der operator"" dazugekommen, mit dem Sie eigene benutzerdefinierte Literale schreiben können. Wie dies geht und mehr dazu, finden Sie in Abschnitt 16.2, »Benutzerdefinierte Literale (C++11)«.

Kapitel 9

Vererbung

Mit der Vererbung lernen Sie in diesem Kapitel ein weiteres grundlegendes Konzept der objektorientierten Programmierung kennen. Bei diesem Konzept haben Sie die Möglichkeit, neue Klassen auf Basis von bestehenden Klassen zu definieren.

Das Grundprinzip der Vererbung (auch Ableitung bzw. abgeleitete Klassen genannt) ist es, vorhandene Klassen weiterzuverwenden und aufbauend darauf neue und erweiterte Klassen zu schreiben. Dabei kann die neue Klasse entweder eine Erweiterung der vorhandenen Klasse oder aber auch eine Beschränkung sein.

Verwenden wir als Beispiel die Klasse `Personal` aus Kapitel 7, »Grundlagen zu den Klassen«. Benötigen Sie jetzt bspw. eine Klasse `Zeitarbeiter` oder `Angestellter`, dann müssen Sie nicht den gleichen, bereits in `Personal` verwendeten Code erneut schreiben, sondern Sie leiten einfach von dieser Klasse ab, so dass jetzt alle Eigenschaften und Methoden von `Personal` auch den neuen Klassen `Zeitarbeiter` und `Angestellter` zur Verfügung stehen. Die Klassen `Zeitarbeiter` und `Angestellter` können Sie jetzt noch um weitere Eigenschaften und Methoden erweitern, welche eben für solche spezielle Klassen nötig sind und in `Personal` nicht vorhanden sind.

Der Vorteil dieser Erweiterung von abgeleiteten Klassen liegt somit darin, dass Sie nicht immer wieder denselben Code für Eigenschaften und Methoden in neuen Klassen schreiben müssen, sondern dass Sie sich einfach bei Eigenschaften und Methoden bereits vorhandener Klassen durch die Vererbung bedienen. Das erspart eine Menge an Tipp- und Testarbeit. Einmal erstellte Klassen müssen nur einmal geschrieben und einmal getestet werden.

Ebenso können Sie von Klassen ableiten, deren Quellcode Sie nicht einmal haben. Wenn lediglich die Headerdatei und der Objektcode (bzw. die Bibliotheksdatei) vorhanden sind, können Sie auch von dieser Klasse ableiten und sie erweitern bzw. modifizieren.

9.1 Grundlagen zur Vererbung

Bevor Sie selbst eine Vererbung erstellen, müssen Sie noch ein paar Fachbegriffe davon kennen. Die Klasse, von der Sie eine neue Klasse ableiten wollen, wird als **Basisklasse** (oder Oberklasse) bezeichnet. Die Klasse, die von der Basisklasse abgelei-

tet wurde, wird als **abgeleitete Klasse** (oder Unterklasse) bezeichnet. Gewöhnlich erbt die abgeleitete Klasse dann die (öffentlichen) Eigenschaften und Methoden der Basisklasse.

Bei der Verwaltung von Personal einer Firma gibt es bspw. verschiedene Arten von Personal: Angestellte, Zeitarbeiter, Hilfsarbeiter, Azubis, Praktikanten usw. Jedes einzelne »Personal« besitzt im Grunde dieselben Eigenschaften (Name, Gehalt, Personalnummer usw.) und Methoden (Gehalt ändern, Personalnummer ändern, Name ändern usw.). Damit das Personal auch unterschieden werden kann, werden von der Basisklasse *Personal* weitere Klassen, wie etwa eine Klasse *Angestellter* oder *Zeitarbeiter*, abgeleitet. Diese abgeleiteten Klassen *Angestellter* oder *Zeitarbeiter* besitzen zusätzlich zur Basisklasse *Personal* noch weitere Eigenschaften, wie bspw. spezielle Leistungszulagen oder den Vermerk, für wie lange der Zeitvertrag eines Zeitarbeiters läuft. Die Beschreibung wird in Abbildung 9.1 demonstriert.

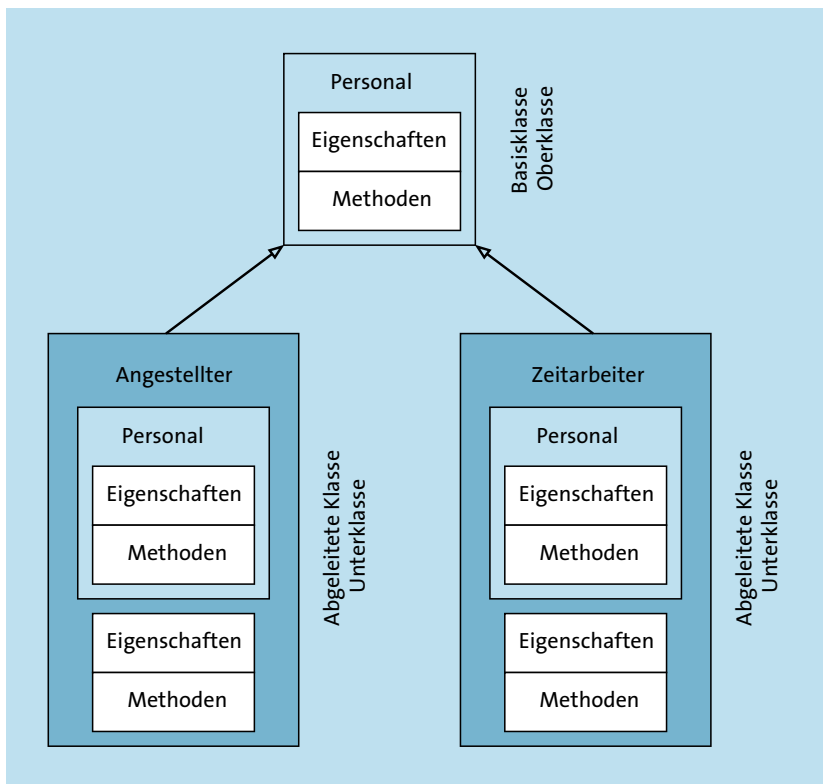


Abbildung 9.1 Aus der Basisklasse »Personal« werden mit »Angestellter« und »Zeitarbeiter« neue abgeleitete Klassen konstruiert.

(Öffentliche) Eigenschaften und Methoden werden von der Basisklasse dabei an die abgeleitete Klasse weitervererbt. Hieraus ergibt sich eine Ist-Beziehung: Ein Objekt vom

Typ »Angestellter« ist somit auch ein spezielles Objekt der Klasse »Personal«. Auch ein Objekt von Typ »Zeitarbeiter« ist ein Objekt der Klasse »Personal« (Abbildung 9.1).

Ist-Beziehung

Ein Objekt einer abgeleiteten Klasse ist somit immer ein Objekt vom Typ der Basis-klassse. Im konkreten Beispiel ist ein Objekt vom Typ `Angestellter` ein spezielles Objekt der Klasse `Personal`. Selbiges gilt auch für ein Objekt vom Typ `Zeitarbeiter`. Die abgeleitete Klasse hat somit immer eine Ist-Beziehung (engl. *is-a-relationship*) zur Basisklasse.

has-a-relationship (HAT-Beziehung)

Die hat-Beziehung (auch Aggregationen) wurde bisher noch nicht erwähnt, obwohl Sie diese bereits mehrmals in der Praxis verwendet haben. Eine hat-Beziehung besteht, wenn ein Element (Eigenschaft) einer Klasse selbst eine (andere) Klasse ist. Vereinfacht würde dies bspw. bedeuten, wenn die Klasse `Personal` ein Element `std::string` für den Namen enthält (was ja auch eine Klasse ist), dann HAT ein `Personal`-Objekt ein Teilobjekt vom Typ `std::string` für den Namen der Person.

Hierbei will ich nochmals gesondert darauf hinweisen, dass eine Vererbung nur in eine Richtung gilt! Auch wenn ein `Angestellter` immer ein `Personal` ist, ist nicht jedes `Personal` automatisch auch ein `Angestellter`. Damit ist gemeint, dass Sie jederzeit ein Objekt der abgeleiteten Klasse `Angestellter` einem Objekt der Basisklasse `Personal` zuweisen können. Sind keine speziellen Eigenschaften vorhanden, würden hier nur die Eigenschaften der abgeleiteten Klasse `Angestellter` der Basisklasse des `Personal`-Objekts zugewiesen werden. Umgekehrt funktioniert dies nicht mehr. Würden Sie ein `Personal`-Objekt einem Objekt der abgeleiteten Klasse `Angestellter` zuweisen, würde dies zu einem Fehler führen, weil die Eigenschaften der abgeleiteten Klasse nicht mehr zugeordnet werden könnten.

Mehrfache Ableitung

Es ist auch möglich, dass eine abgeleitete Klasse ebenfalls abgeleitet werden kann und somit auch eine Basisklasse sein kann. Betrachten Sie Abbildung 9.2, wo wir von der bereits von `Personal` abgeleiteten Klasse `Angestellter` die Klassen `Aussertariflich` und `Leitender_Angestellter` ableiten – was ja durchaus Sinn ergeben würde. Damit ist quasi die Klasse `Angestellter` eine abgeleitete Klasse von `Personal` und die Basisklasse von `Aussertariflich` und `Leitender_Angestellter`.

Hier ist häufig auch die Rede von **direkter und indirekter Basisklasse**. In Abbildung 9.2 wäre somit die Klasse `Aussertariflich` direkt von der Klasse `Angestellter` abgeleitet und indirekt von der Klasse `Personal`.

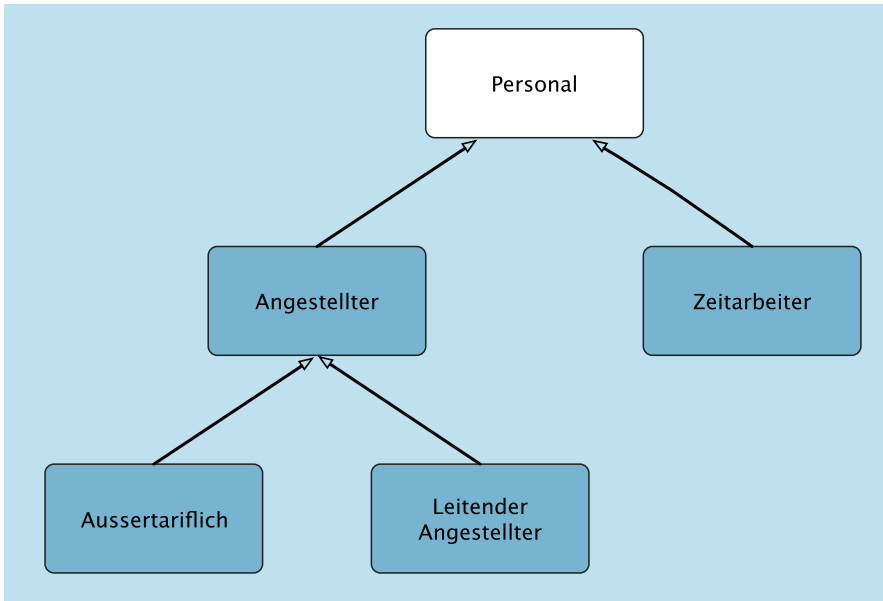


Abbildung 9.2 Es ist auch möglich, dass eine Klasse sowohl eine abgeleitete Klasse als auch eine Basisklasse ist (wie hier die Klasse »Angestellter«).

Um eventuelle Missverständnisse zu vermeiden Hier sollte vielleicht noch erwähnt werden, dass eine mehrfache Ableitung keine Mehrfachvererbung ist. Das ist etwas anderes und wird in Abschnitt 9.10, »Mehrfachvererbung«, behandelt.

Klassenhierarchien von grafischen Oberflächen

So ein Ableiten von einer Klasse zur anderen wird im extremen Maße bei Klassenbibliotheken grafischer Oberflächen verwendet. Bestes Beispiel ist das Qt-Framework, welches im gesamten letzten Kapitel des Buches (Kapitel 17, »GUI-Programmierung mit Qt«) beschrieben wird.

9.2 Abgeleitete Klassen implementieren

Soweit zur Theorie. Jetzt werden Sie erfahren, wie Sie eine Vererbung implementieren können. Damit der Hauptfokus beim Thema der Vererbung bleibt, werden die Beispiele wie immer möglichst einfach gehalten. Mit folgender Syntax können Sie eine Vererbung realisieren:

```

class Klassenname : [Zugriffsrechte] Basisklassenamen {
    // Klassendefinition von Klassenname
};

```

Der Klassenname ist noch wie gewöhnlich eine Klasse, welche Sie anschließend definieren wollen. Hinter dem Doppelpunkt geben Sie den Basisklassenamen ein, von welchem (öffentliche) Eigenschaften und Methoden geerbt werden. Mit Zugriffsrechte legen Sie fest, wie Sie auf die Mitglieder der Basisklasse zugreifen können. Hierfür können Sie `private`, `public` und `protected` verwenden. Die Zugriffsrechte werden noch gesondert in Abschnitt 9.3, »Zugriffsschutz anpassen«, behandelt. In der Praxis werden Sie meistens `public` verwenden. Wenn Sie keine Zugriffsrechte beim Ableiten der Basisklasse hinter dem Doppelpunkt angeben, wird standardmäßig `private` verwendet.

9.2.1 Basisklasse erstellen

Wir wollen hierzu in der Praxis eine leicht vereinfachte Form unserer Klasse `Personal` zum Weitervererben verwenden. Hierzu die Klassendefinition der Headerdatei *personal.h*:

```
00 // kapitel09/vererbung01/personal.h
01 #include <iostream>
02 #include <string>
03 #ifndef PERSONAL_H
04 #define PERSONAL_H
05 static const unsigned int GG = 1900;

06 class Personal {
07     std::string name;      // Name
08     unsigned int pers_nr;  // Personalnummer
09     unsigned int gehalt;   // monatliches Grundgehalt
10 public:
11     Personal();
12     Personal(std::string n, unsigned int pn, unsigned int g);
13     Personal(std::string n, unsigned int pn);
14     Personal(std::string n);
15     void set_name(const std::string &n);
16     std::string get_name() const;
17     void set_pers_nr(unsigned int nr);
18     unsigned int get_pers_nr() const;
19     void set_gehalt(unsigned int g);
20     unsigned int get_gehalt() const;
21     void print_data();
22 };
23 #endif
```

Auf das Abdrucken der Implementierung der Konstruktoren und Methoden in *personal.cpp* wurde hier jetzt verzichtet. Der Code von *personal.cpp* wurde ja schon in

Kapitel 7, »Grundlagen zu den Klassen«, umfassend behandelt. Sie finden diesen Code auch auf der Buch-CD wieder.

Code nicht unbedingt nötig

Wie bereits anfangs erwähnt, ist es nicht zwingend notwendig, den Quellcode der Implementierung zu besitzen. Wenn Sie die Headerdatei haben und die Implementierung lediglich als bereits übersetzte Objekt- oder Bibliotheksdatei vorliegt, so reicht dies ebenfalls aus, um davon abzuleiten.

9.2.2 Von der Basisklasse ableiten

Von dieser Klasse `Personal` wollen wir jetzt eine Klasse `Angestellter` ableiten. Auch hier beschränken wir uns auf ein Minimum und erweitern die Klasse lediglich um eine Eigenschaft, damit die Übersicht auf das Thema bewahrt bleibt.

```

00 // kapitel09/vererbung01/angestellter.h
01 #include <iostream>
02 #include <string>
03 #include "personal.h"
04 #ifndef ANGESTELLTER_H
05 #define ANGESTELLTER_H

06 class Angestellter : public Personal {
07     unsigned int zulagen; // Leistungszulagen
08 public:
09     Angestellter( unsigned int zu=0, std::string n="",
10                  unsigned int pn=0, unsigned int g=GG );
11     void set_zulagen(unsigned int z);
12     unsigned int get_zulagen() const;
13     void print_data();
14 };
15 #endif

```

Hier sehen Sie die Klassendefinition der Klasse `Angestellter`. In Zeile **06** machen Sie hinter dem Doppelpunkt die Klasse `Personal` zur Basisklasse von `Angestellter`. Unsere Klasse `Angestellter` wurde hiermit von der Basisklasse `Personal` abgeleitet. Die drei Eigenschaften `name`, `pers_nr` und `gehalt` der Basisklasse `Personal` werden somit an die abgeleitete Klasse `Angestellter` weitervererbt. Ein Objekt der Klasse `Angestellter` enthält zusätzlich noch die Eigenschaft `zulagen`, (Zeile **07**), wo Extra-Zulagen, Bonus oder außertarifliche Bezahlungen zusätzlich zum Grundgehalt gespeichert werden. Somit enthält ein Objekt `Angestellter` insgesamt vier Eigenschaften. In der Praxis

würde man hier viel mehr Eigenschaften verwenden, aber der Übersicht zuliebe wurde hier darauf verzichtet.

Neben dem Konstruktor und den `set-` bzw. `get-`Methoden fällt auf, dass die abgeleitete Klasse `Angestellter` ebenfalls eine Methode `print_data()` deklariert (Zeile 12). Somit erbt die abgeleitete Klasse `Angestellter` die Methode `print_data()` und redefiniert zusätzlich eine gleichnamige Methode. Auf eine solche Redefinition von Elementen einer Klasse wird noch gesondert in Abschnitt 9.4.4, »Redefinition von Methoden«, eingegangen.

Durch das `public`-Zugriffsrecht, welches wir hier verwenden, können Sie in der Klasse `Angestellter` auf alle `public`-Mitglieder der Klasse `Personal` ohne weitere Einschränkungen zugreifen. Wenn Sie zurück zur Klassendefinition von *kapitel09/vererbung01/personal.h* blättern, so wären dies alle Mitglieder der Zeilen 11 bis 21. Auf `private` Mitglieder der Basisklasse `Personal` kann von der abgeleiteten Klasse `Angestellter` nicht zugegriffen werden. Die Headerdatei (siehe Zeile 03) der Basisklasse `Personal` gehört logischerweise hier auch mit rein.

Jetzt noch die Definition des Konstruktors und der Methoden der abgeleiteten Klasse `Angestellter` in *angestellter.cpp*:

```
00 // kapitel09/vererbung01/angestellter.h
01 #include "angestellter.h"

02 Angestellter::Angestellter(
    unsigned int zu, std::string n,
    unsigned int pn, unsigned int g):
    Personal(n, pn, g),zulagen(zu) {}
03 void Angestellter::set_zulagen(unsigned int z) {
04     zulagen=z;
05 }

06 unsigned int Angestellter::get_zulagen() const {
07     return zulagen;
08 }

09 void Angestellter::print_data() {
10     Personal::print_data();
11     std::cout << "+Zulagen: " << get_zulagen() << std::endl;
12     std::cout << "=Gesamt:" << get_zulagen() + get_gehalt()
        << std::endl;
13 }
```

Der Konstruktor der abgeleiteten Klasse in Zeile 02 ist etwas spezieller als ein gewöhnlicher Konstruktor. Alle Eigenschaften wurden hierbei gleich mit dem Ele-

ment-Initialisierer initialisiert. Hinter dem Doppelpunkt wird der Konstruktor der Basisklasse `Personal` aufgerufen. Dahinter werden dann erst die Eigenschaften der abgeleiteten Klasse initialisiert. Das Thema »Auf- und Abbau von abgeleiteten Klassen« ist recht speziell und wird noch gesondert in Abschnitt 9.5, »Auf- und Abbau von abgeleiteten Klassen«, behandelt.

In den Zeilen **03** bis **08** finden Sie gewöhnliche Methoden der Klasse `Angestellter`. In den Zeilen **09** bis **13** finden Sie dann eine gleichnamige Methode, welche auch schon in der Basisklasse `Personal` enthalten ist. Durch diese Redefinition wird einfach die Methode der Klasse `Personal` überdeckt. Soll der Zugriff hierbei trotzdem über die Methode `print_data()` der Klasse erfolgen, muss diese dann mithilfe des Klassennamens und des Bereichsoperators geschrieben werden (siehe Zeile **10**), weil sonst (ohne `Personal::`) ein endloser Selbstaufruf (Rekursion) von `Angestellter::print_data()` ausgeführt würde.

9.2.3 Erweiterte Klasse verwenden

Das war es auch schon mit der Vererbung. Im Grunde ist es ja nichts anderes als eine Technik, um eine bereits vorhandene Klasse zu erweitern. Natürlich habe ich Ihnen hier noch viele Details vorenthalten oder verschwiegen. Die Feinheiten der Vererbung werden in den nächsten Abschnitten noch umfassend beschrieben.

Der folgende Codeausschnitt soll die abgeleitete Klasse `Angestellter` im Einsatz demonstrieren:

```
00 // kapitel09/vererbung01/main.cpp
01 #include "angestellter.h"
...
02 Angestellter angestellter01;
03 angestellter01.set_name("Max Mustermann");
04 angestellter01.set_pers_nr(400001);
05 angestellter01.set_zulagen(400);
06 angestellter01.print_data();
```

Die Headerdatei in Zeile **01** muss natürlich hier mit rein. In Zeile **02** wird ein leeres Objekt der Klasse `Angestellter` instanziiert. In den Zeilen **03** bis **05** setzen wir einzelne Werte und in Zeile **06** geben wir den Inhalt vom Objekt `angestellter01` auf dem Bildschirm aus.

Alle Dateien beim Übersetzen mit angeben

Bei diesem Beispiel müssen Sie neben *main.cpp* auch noch die Quelldateien *personal.cpp* und *angestellter.cpp* dem Projekt hinzufügen, damit ein ausführbares Programm daraus erstellt wird.

Die Ausgabe des Programms lautet:

```
Name      : Max Mustermann
Pers.Nr.: 400001
Gehalt   : 1900
+Zulagen: 400
= Gesamt: 2300
```

9.2.4 Direkte und indirekte Basisklasse

Es wurde bereits einmal erwähnt und es soll hier nur kurz darauf eingegangen werden. Wenn Sie jetzt die Gruppe `Angestellter` auch noch in die Klassen `Aussertariflich` und `Leitender_Angestellter` aufteilen wollen, so müssen Sie im Grunde genauso vorgehen, wie Sie dies eben mit der Basisklasse `Personal` und der davon abgeleiteten Klasse `Angestellter` gemacht haben:

```
class Personal {
    // Klassendefinition
};

class Angestellter : public Personal {
    // Klassendefinition
};

class Aussertariflich : public Angestellter {
    // Klassendefinition
};

class Leitender_Angestellter : public Angestellter {
    // Klassendefinition
};
```

Hiermit hätten die Klassen `Aussertariflich` und `Leitender_Angestellter` die Klasse `Angestellter` als Basisklasse und die Klasse `Personal` als indirekte Basisklasse (siehe Abbildung 9.3). Auch hierbei werden alle (öffentlichen) Eigenschaften und Methoden weitervererbt. So erben die Klassen `Aussertariflich` und `Leitender_Angestellter` alle (öffentlichen) Eigenschaften und Methoden der Basisklasse `Angestellter` und der indirekte Basisklasse `Personal`.

Die Beziehung zwischen `Angestellter` und `Personal` bleibt gleich. `Personal` bleibt nach wie vor die direkte Basisklasse von `Angestellter` und kann immer noch wie gehabt verwendet werden.

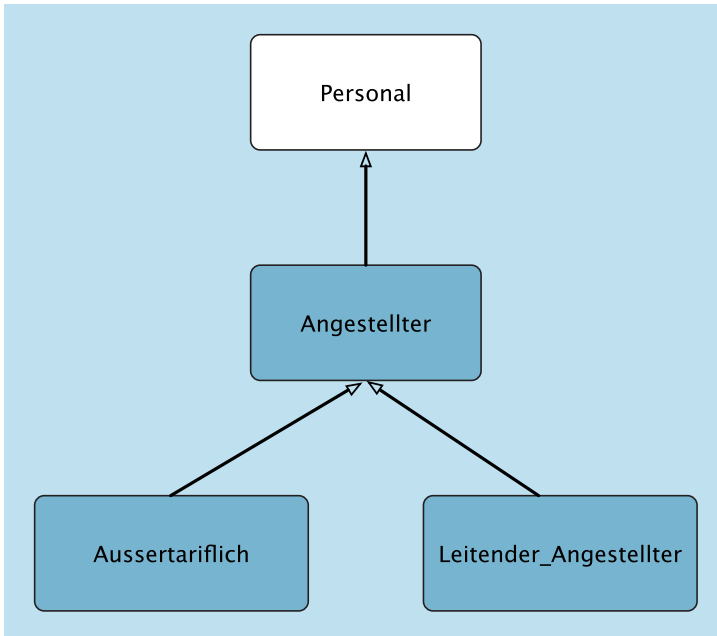


Abbildung 9.3 »Personal« wird zur indirekten und »Angestellter« zur direkten Basisklasse von »Aussertariflich« und »Leitender_Angestellter«.

9.2.5 Abgeleitete Klassen verbieten (C++11)

Sind Sie der Meinung, eine Klasse braucht keine weitere Spezialisierung mehr, oder wollen Sie einfach nicht, dass von einer bestimmten Klasse abgeleitet werden kann (bspw. wenn dort sicherheitsrelevante Daten enthalten sind), dann können Sie diese Klasse mit dem neuen C++11-Schlüsselwort `final` markieren. Ein einfacher Codeausschnitt dazu:

```

01 class Personal final {
02     // Klassendefinition
03 };

04 class Angestellter : public Personal { // Jetzt: Fehler!!!
05     // Klassendefinition
06 };
  
```

Durch das Schlüsselwort `final` in Zeile **01** würde sich der Compiler jetzt über die Ableitung in Zeile **04** beschweren mit der Meldung, dass er eben keine Ableitung von der finalen Basisklasse `Personal` machen kann.

Bei einer mehrfach abgeleiteten Klasse kann dies recht ähnlich implementiert werden. Wollen Sie bspw. die abgeleitete Klasse `Angestellter` als finale Klasse markieren,

wovon anschließend keine weitere Ableitung mehr gemacht werden soll, so können Sie dies wie folgt implementieren:

```

01 class Personal {
02     // Klassendefinition
03 };

04 class Angestellter final : public Personal {
05     // Klassendefinition
06 };

07 class Aussertariflich : public Angestellter { // Fehler!!!
08     // Klassendefinition
09 };

```

Hier markieren wir in Zeile **04** die von `Personal` abgeleitete Klasse `Angestellter` als finale Klasse, wovon jetzt nicht mehr abgeleitet werden kann. Die Ableitung von `Angestellter` in Zeile **07** wird jetzt vom Compiler als Fehler gemeldet.

9.3 Zugriffsschutz anpassen

Zwar wird in der Praxis meistens immer eine `public`-Vererbung gemacht, aber es gibt gelegentlich doch Fälle, wo man eine `private`- oder `protected`-Vererbung machen will. Der Effekt macht sich allerdings erst bemerkbar, wenn Sie mehrere Klassen ableiten. Die erste abgeleitete Klasse hat, unabhängig vom Zugriffsrecht, immer vollen Zugriff auf alle `public`- und `protected`-Mitglieder der Basisklasse. Folgendes bewirken die Zugriffsrechte `public`, `private` und `protected` bei der Vererbung:

- ▶ `public`: Diese Ableitung wird meistens verwendet. Die abgeleitete Klasse erbt dieselben Zugriffsrechte von den direkten und indirekten Basisklassen. Durch eine `public`-Vererbung kann auf alle `public`- und `protected`-Schnittstellen der Basisklasse zugegriffen werden. Das ist die klassische Ist-Beziehung.
- ▶ `private`: Bei dieser Vererbung werden die `public`- und `protected`-Mitglieder der Basisklasse zu `private`-Mitgliedern in der abgeleiteten Klasse. Der Zugriff auf die Mitglieder der Basisklasse ist jetzt nur noch innerhalb der abgeleiteten Klasse möglich. Ein Zugriff von außen ist nicht mehr möglich (auch nicht auf `public`-Mitglieder der Basisklasse). Auch bei einer weiteren Ableitung von der bereits abgeleiteten Klasse (genauer indirekten Basisklasse) wäre kein Zugriff mehr auf die direkte, geschweige denn indirekte, Basisklasse mehr möglich.
- ▶ `protected`: Bei dieser Vererbung werden die `public`- und `protected`-Mitglieder der Basisklasse zu `protected`-Mitgliedern in der abgeleiteten Klasse. Die Zugriffsrechte

innerhalb der Klassen ändern sich nicht. Es kann nach wie vor auf `public`- und `protected`-Mitglieder der Basisklasse zugegriffen werden. Allerdings kann jetzt nicht mehr von außen auf die `public`-Mitglieder der Klasse zugegriffen werden.

Das war jetzt eine ziemliche Menge an Informationen auf einmal. Daher finden Sie in Tabelle 9.1 bis Tabelle 9.3 nochmals einen besseren Überblick, was bei einer `public`-, `protected`- und `private`-Vererbung jeweils passiert.

	public	private	protected
<code>class Basis;</code>	ja	ja	ja
<code>class Abgel01 : public Basis;</code>	ja	nein	ja
<code>class Abgel02 : public Abgel01;</code>	ja	nein	ja
außerhalb einer Klasse	ja	nein	nein

Tabelle 9.1 Diese Tabelle zeigt, ob bei einer `public`-Vererbung die `public`-, `private`- oder `protected`-Mitglieder ebenfalls weitervererbt werden.

	public	private	protected
<code>class Basis;</code>	ja	ja	ja
<code>class Abgel01 : protected Basis;</code>	ja	nein	ja
<code>class Abgel02 : protected Abgel01;</code>	ja	nein	ja
außerhalb einer Klasse	nein	nein	nein

Tabelle 9.2 Diese Tabelle zeigt, ob bei einer `protected`-Vererbung die `public`-, `private`- oder `protected`-Mitglieder ebenfalls weitervererbt werden.

	public	private	protected
<code>class Basis;</code>	ja	ja	ja
<code>class Abgel01 : private Basis;</code>	ja	nein	ja
<code>class Abgel02 : private Abgel01;</code>	nein	nein	nein
außerhalb einer Klasse	nein	nein	nein

Tabelle 9.3 Diese Tabelle zeigt, ob bei einer `private`-Vererbung die `public`-, `private`- oder `protected`-Mitglieder ebenfalls weitervererbt werden.

Überladene Operatoren bei der Vererbung

Für überladene Operatoren gilt dasselbe bei der Vererbung wie für Methoden, und es werden alle nicht privaten Operatorüberladungen aus der Basisklasse weitervererbt – mit einer Ausnahme, wenn in der Basisklasse `operator=` überladen wurde! Der `operator=` der Basisklasse wird niemals an die abgeleitete Klasse weitervererbt.

Es ist schwierig hier ein einzelnes und zudem noch sinnvolles Beispiel zu liefern. Daher finden Sie hier ein Codebeispiel, an dem Sie die Zugriffsrechte selbst ausprobieren können.

```
00 // kapitel09/vererbung02/main.cpp
...
01 class Basis {
02     private : int private_val;
03     public   : int public_val;
04     protected : int protected_val;
05 };

06 class Abgeleitet : /*ACCESS*/ Basis {
07     // Zugriff auf public und protected
08     // von Basis ist hier immer möglich
09 };

10 class Sub_Abgeleitet : public Abgeleitet {
11 public:
12     void set_vals() {
13         private_val = 1; // Immer falsch!!!
14         public_val   = 2; // Ok: public und protected
15         protected_val = 3; // Ok: public und protected
16     }
17 };

18 int main() {
19     Sub_Abgeleitet objekt;
20     objekt.set_vals();
21     objekt.public_val = 4; // Ok: public
22 }
```

Hier können Sie in Zeile **06** den Text `/*ACCESS*/` durch eines der Zugriffsrechte `public`, `protected` oder `private` ersetzen und experimentieren, welche Zuweisungen in der Methode `set_vals()` in den Zeilen **12** bis **16** erlaubt sind und welche nicht. Die Methode wird in Zeile **20** aufgerufen.

Das Beispiel selbst ist einfach. Die Basisklasse `Basis` enthält drei Eigenschaften, welche jeweils im `private`-, `public`- und `protected`-Bereich geschrieben stehen. Die Ableitung in den Zeilen 06 bis 09 ist belanglos, weil hier unabhängig vom verwendeten Zugriffsrecht in Zeile 06 für `/*ACCESS*/` immer Zugriff auf alle `public`- und `protected`-Mitglieder besteht. Interessant wird das Zugriffsrecht von `/*ACCESS*/` erst in der nächsten Ableitung, welche in den Zeilen 10 bis 17 geschrieben wurde. Hier werden Sie herzlich dazu eingeladen, die Fehlermeldungen des Compilers zu lesen. Dieser wird Ihnen verraten, welcher Zugriff bei welchem Zugriffsrecht nicht mehr erlaubt ist.

Definitiv immer verboten ist der Zugriff auf `private_val`, wie dies in Zeile 13 versucht wurde. Dieser Zugriff wäre allerdings auch schon in der indirekten Basisklasse Abgeleitet verboten gewesen. Ich denke, das sollte keine Überraschung mehr sein. Ein Zugriff auf `public_val` (Zeile 14) und `protected_val` (Zeile 15) innerhalb der Methode `set_vals()` ist grundsätzlich immer möglich, wenn Sie für `/*ACCESS*/` in Zeile 06 eine `public`- bzw. `protected`-Vererbung verwendet haben.

Der direkte Zugriff von außen auf die `public`-Eigenschaft in Zeile 21 ist nur möglich, wenn Sie eine `public`-Vererbung in Zeile 06 angelegt haben. Mit einer `protected`-Vererbung hingegen wird der Zugriff von außen verboten. Aber im Gegensatz zu einer `private`-Vererbung funktioniert trotzdem noch der Zugriff innerhalb der Klasse, wie in Zeile 14 und 15 demonstriert werden soll.

protected-Mitglieder einer Klasse

In den Abschnitten zuvor war des Öfteren auch die Rede von `protected` bei der Vererbung, ohne dass bisher auf die `protected`-Mitglieder einer Klasse eingegangen wurde. Das soll hier jetzt nachgeholt werden.

Ein mit `protected` gekennzeichnetes Mitglied der Klasse ist wie ein `private`-Mitglied von außen geschützt. Innerhalb von abgeleiteten Klassen verhält sich ein als `protected` gekennzeichnetes Mitglied dann wieder wie ein `public`-Mitglied, und es kann mit Methoden oder `friend`-Funktionen darauf zugegriffen werden. Mit `protected` sorgen Sie somit dafür, dass nur noch abgeleitete Klassen einen Zugriff auf die Mitglieder der Klassen haben.

In der Praxis sollten Sie `protected` mit Bedacht und endgültig einsetzen. Wenn Sie bspw. die Deklaration eines `protected`-Mitglieds einer Klasse ändern, so werden umfangreiche Prüfungen in allen von dieser Klasse abgeleiteten Klassen nötig.

Zur Demonstration soll jetzt in unserer Basisklasse `Personal` eine Methode mit `protected` gekennzeichnet werden:

```
00 // kapitel09/vererbung03/personal.h
...
01 class Personal {
```

```

02  std::string name;      // Name
03  unsigned int pers_nr;  // Personalnummer
04  unsigned int gehalt;   // monatliches Grundgehalt
05  public:
06  Personal();
07  Personal(std::string n, unsigned int pn, unsigned int g);
08  Personal(std::string n, unsigned int pn);
09  Personal(std::string n);
10  void set_name(const std::string &n);
11  std::string get_name() const;
...
12  protected:
13  void set_gehalt(unsigned int g);
14  };

```

Hier wurde in Zeile 12 und 13 die Methode `set_gehalt()` mit `protected` gekennzeichnet. Somit kann diese Methode zwar noch in unserer abgeleiteten Klasse `Angestellter` verwendet werden, aber ein nachträgliches Ändern von außen ist nicht mehr erlaubt:

```

00  // kapitel09/vererbung03/main.cpp
...
01  #include "angestellter.h"
...
02  Angestellter angestellter01;
03  angestellter01.set_name("Max Mustermann");
04  angestellter01.set_pers_nr(400001);
05  angestellter01.set_zulagen(400);
06  angestellter01.set_gehalt(2300); //Nicht erlaubt:protected
07  angestellter01.print_data();

08  Angestellter angestellter02(200, "Max Musterfrau", 400002, 2700 );
09  angestellter02.print_data();
...

```

Ein Zugriff, wie er hier in Zeile 06 versucht wurde, ist jetzt über das Objekt von außen nicht mehr möglich und wird vom Compiler mit einer Fehlermeldung quittiert. Entweder hier wird gleich in Zeile 08 das Gehalt beim Anlegen des Objekts festgelegt, oder es wird das Standardgehalt verwendet. Damit ist sichergestellt, dass das Grundgehalt nicht mehr einfach so von außen geändert werden kann.

protected-Mitglieder nach außen durchreichen

Unter Umständen ist es jetzt trotzdem möglich, in der Klasse `Angestellter` den `protected`-Schutz der Basisklasse zu unterlaufen und nach außen weiterzureichen. Es ist nicht schön, weil Sie damit das Prinzip von `protected` aufweichen, soll hier aber trotzdem kurz demonstriert werden:

```
00 // kapitel09/vererbung04/angestellter.h
...
01 class Angestellter : public Personal {
02     unsigned int zulagen;    // Leistungszulagen
03 public:
04     Angestellter( unsigned int zu=0, std::string n="",
                   unsigned int pn=0, unsigned int g=GG );
05     void set_zulagen(unsigned int z);
06     unsigned int get_zulagen() const;
07     void print_data();
08     using Personal::set_gehalt;
09 };
```

Hier wird noch davon ausgegangen, dass die Methode `Personal::set_gehalt()` in der Klasse `Personal` als `protected` gekennzeichnet wurde. Durch Zeile **08** mit `using` übernehmen wir die Methode `Personal::set_gehalt` in den `public`-Bereich der Klasse `Angestellter`, wodurch ein Zugriff von außen wieder möglich wäre.

Dieses »Durchreichen« lässt sich auch beim Ableiten von `private`-Vererbungen verwenden. Auch hierzu ein kurzes Beispiel (allerdings ebenfalls nur unter Vorbehalt weiterzuempfehlen):

```
00 // kapitel09/vererbung05/main.cpp
...
01 class Basis {
02     public    : int public_val;
03     protected : int protected_val;
04 };

05 class Abgeleitet : private Basis {
06     // Zugriff auf public und protected
07     // von Basis ist hier immer möglich
08     protected:
09         using Basis::public_val;
10         using Basis::protected_val;
11 };

12 class Sub_Abgeleitet : public Abgeleitet {
```

```

13 public:
14     void set_vals() {
15         public_val    = 1;
16         protected_val = 2;
17     }
18 };
...
19 Sub_Abgeleitet objekt;
20 objekt.set_vals();      // Ok
21 objekt.public_val = 3;  // protected ! Fehler !

```

Hier wurde in Zeile 05 eine private-Ableitung von Basis an die Klasse Abgeleitet durchgeführt. Hierbei wäre es nicht mehr möglich gewesen, in der davon abgeleiteten Klasse Sub_Abgeleitet auf die Variablen Basis::public_val und Basis::protected_val in den Zeilen 15 und 16 zuzugreifen. Da wir hier aber die beiden Variablen in den Zeilen 09 und 10 in den protected-Bereich mit using gestellt haben, werden diese Variablen jetzt in der abgeleiteten Klasse Sub_Abgeleitet übernommen. Einem Zugriff wie in Zeile 20 steht daher nichts mehr im Wege, aber der Zugriff in Zeile 21 ist aufgrund von protected in Zeile 08 von außen nicht erlaubt. Dieser wäre nur möglich, wenn Sie in Zeile 08 den Zugriff statt auf protected auf public ändern würden.

Wie allerdings eingangs erwähnt, ist das nicht unbedingt ein sauberer Stil und sollte nur im äußersten Notfall verwendet werden, wenn sich das Umschreiben des Designfehlers, den Sie da vor sich haben, einfach als zu aufwendig erweist.

9.4 Abgeleitete Klassen verwenden und erweitern

Nachdem Sie jetzt mit den Zugriffsrechten vertraut sind, wollen wir uns wieder den Basics der Vererbung widmen. Der Hauptanwendungszweck der Vererbung ist die Erweiterung bereits vorhandener Klassen um weitere Eigenschaften und Methoden sowie die Verwendung bereits vorhandener (öffentlicher) Eigenschaften und Methoden. Und auf diese Mitglieder der Basisklassen und abgeleiteten Klassen soll hier jetzt etwas genauer eingegangen werden.

9.4.1 Zugriff auf die erweiterten Mitglieder der abgeleiteten Klasse

Auch wenn es bereits erwähnt wurde und mittlerweile klar sein sollte, mit Methoden abgeleiteter Klassen können Sie auf alle Mitglieder zugreifen, die in dieser abgeleiteten Klasse zusätzlich definiert wurden:

```

00 // angestellter.cpp
...
01 void Angestellter::set_zulagen(unsigned int z) {
02     zulagen=z;
03 }

04 unsigned int Angestellter::get_zulagen() const {
05     return zulagen;
06 }
...
...
07 // main.cpp
...
08 Angestellter angestellter01;
09 angestellter01.set_zulagen(500);

```

Hier greifen Sie mit den Methoden `Angestellter::set_zulagen()` und `Angestellter::get_zulagen()` direkt auf die hauseigene `private` Eigenschaft `zulagen` der Klasse `Angestellter` zu.

Auch der Zugriff von außen auf die Mitglieder der abgeleiteten Klasse erfolgt ganz gewöhnlich über das Objekt und die in `public` geschriebenen Mitglieder, wie in Zeile **09** zu sehen ist. Zugriffe von außen auf `private`- und `protected`-Mitglieder ist bei abgeleiteten Klassen (natürlich auch) nicht möglich.

9.4.2 Zugriff auf die Mitglieder der Basisklasse

Beim Zugriff auf die Mitglieder der Basisklasse sieht es ein wenig anders aus, weshalb hier noch ein paar Absätze hinzugefügt werden müssen.

protected- und public-Mitglieder der Basisklasse

In einer abgeleiteten Klasse können Sie auf alle Mitglieder der Basisklasse zugreifen, welche im `public`- und `protected`-Bereich geschrieben wurden. Zugriff auf `private`-Mitglieder der Basisklasse ist aus einer abgeleiteten Klasse nicht möglich. Bezogen auf die Methode `Angestellter::print_data()` können Sie wie folgt auf die Mitglieder der Basisklasse `Personal` zugreifen:

```

00 // angestellter.cpp
...
01 void Angestellter::print_data() {
02     std::cout << "Name      : " << get_name() << std::endl;
03     std::cout << "Pers.Nr.: " << get_pers_nr() << std::endl;
04     std::cout << "Gehalt   : " << get_gehalt() << std::endl;

```

```

05     std::cout << "+Zulagen: " << get_zulagen() << std::endl;
06     std::cout << "=Gesamt:" << get_zulagen() + get_gehalt()
        << "\n\n";
07 }

```

Hier sehen Sie, wie in der Methode `print_data()` der Klasse `Angestellter` auf die öffentlichen (`public`) Methoden `get_name()`, `get_pers_nr()` und `get_gehalt()` der Klasse `Personal` in den Zeilen **02**, **03**, **04** und **06** zugegriffen werden kann, weil diese ja über die Vererbung von `Personal` an `Angestellter` weitervererbt wurden. Mit `protected`-Mitgliedern der Basisklasse wäre dies genauso möglich.

private-Mitglieder der Basisklasse

Nur bei einem Zugriff auf die `private`-Mitglieder der Basisklasse besteht auch bei den abgeleiteten Klassen keine Möglichkeit, direkt darauf zuzugreifen. Folgendes ist somit nicht möglich:

```

00 // angestellter.cpp
...
01 void Angestellter::print_data() {
02     std::cout << "Name      : " << name << std::endl;
03     std::cout << "Pers.Nr.: " << pers_nr << std::endl;
04     std::cout << "Gehalt   : " << gehalt << std::endl;
05     std::cout << "+Zulagen: " << zulagen << std::endl;
06     std::cout << "=Gesamt : " << zulagen + gehalt << "\n\n";
07 }

```

Hier wurde versucht, in den Zeilen **02**, **03**, **04** und **06** auf mit `private` gekennzeichnete Eigenschaften der Klasse `Personal` direkt zuzugreifen, was nicht mehr möglich ist und vom Compiler mit einer entsprechenden Fehlermeldung quittiert wird. Der Zugriff der erweiterten mit `private` gekennzeichneten Eigenschaft `zulagen` (Zeile **05** und **06**) der abgeleiteten Klasse `Angestellter` innerhalb der abgeleiteten Klasse ist natürlich nach wie vor möglich.

Zugriff von außen

Der Zugriff von außen über ein Objekt der abgeleiteten Klasse ist nur auf öffentliche (`public`) Mitglieder der Basisklasse möglich. Mitglieder, die in der Basisklasse im `private`- oder `protected`-Bereich stehen, können von außen nicht mehr erreicht werden:

```

00 // main.cpp
...
01 Angestellter angestellter01;
02 angestellter01.set_name("Max Mustermann");

```

```

03 angestellter01.set_pers_nr(400123);
04 angestellter01.set_zulagen(500);
05 angestellter01.print_data();

```

Hier wurden in den Zeilen **02** und **03** die public-Mitglieder `set_name()` und `set_pers_nr()` der Basisklasse `Personal` verwendet. In den Zeilen **04** und **05** hingegen wurde auf public-Mitglieder der abgeleiteten Klasse `Angestellter` zugegriffen.

9.4.3 Suche nach dem passendem Namen

Auf der Suche nach dem richtigen Bezeichner startet der Compiler immer zuerst mit der Klasse, von der Sie ein Objekt instanziiieren, und arbeitet sich dann bei der Suche in der Klassenhierarchie durch die indirekten und direkten Basisklassen nach oben und sucht dort nach einem passenden Bezeichner. Ein einfaches Beispiel:

```

01 Angestellter angestellter01;
02 angestellter01.set_name("Max Mustermann");

```

In Zeile **02** wurde die Methode `set_name()` verwendet. Der Compiler sucht hierbei zunächst in der abgeleiteten Klasse `Angestellter` nach dem Bezeichner `set_name()`. Im gegebenen Beispiel wird er dort nicht fündig und setzt seine Suche dann eine Stufe höher in der Basisklasse `Personal` fort, wo die passende Methode gefunden wird.

9.4.4 Redefinition von Methoden

Jetzt soll hier noch über die *Redefinition* von Methoden aufgeklärt werden. Da Sie ja zuvor in Abschnitt 9.4.3 erfahren haben, wie nach dem richtigen Namen gesucht wird, werden Sie schon ahnen, wie es sich jetzt bei der Redefinition verhält. In unserem Beispiel mit der Basisklasse `Personal` und der abgeleiteten Klasse `Angestellter` gab es ja zweimal eine Methode mit demselben Namen: `print_data()`:

```

00 // personal.cpp
...
01 void Personal::print_data() {
02     std::cout << "Name      : " << get_name() << std::endl;
03     std::cout << "Pers.Nr.:" << get_pers_nr() << std::endl;
04     std::cout << "Gehalt   : " << get_gehalt() << std::endl;
05 }
...
...
00 // angestellter.cpp
...

```

```

01 void Angestellter::print_data() {
02     std::cout << "Name      : " << get_name() << std::endl;
03     std::cout << "Pers.Nr.:" << get_pers_nr() << std::endl;
04     std::cout << "Gehalt   : " << get_gehalt() << std::endl;
05     std::cout << "+Zulagen:" << get_zulagen() << std::endl;
06     std::cout << "=Gesamt:" << get_zulagen() + get_gehalt()
           << "\n\n";
07 }

```

Was hier passiert, hat allerdings nicht, wie vielleicht vermutet, mit einer Überladung zu tun, sondern eher mit dem Thema Gültigkeitsbereich, wo eine lokalere Variable immer die globalere Version innerhalb des Gültigkeitsbereichs überdeckt. Somit überdeckt die Methode `Angestellter::print_data()` die gleichnamige Methode `Personal::print_data()` in einem neuen Gültigkeitsbereich.

Bezogen auf die Suche nach der richtigen Methode, würde zunächst in der abgeleiteten Klasse nach `print_data()` gesucht werden. Im Beispiel würde der Compiler fündig und hätte somit schon seine Methode und würde in der Basisklasse gar nicht mehr danach suchen.

Wenn Sie innerhalb derselben Klasse hingegen eine gleichnamige Methode verwenden, dann würden Sie die Methoden überladen – was durchaus auch erlaubt ist und dann eben als *mehrfache Redefinition* bezeichnet wird.

Zweck einer Redefinition

Eine Redefinition hat denselben Vorteil wie schon die Vererbung selbst. Mit einer Redefinition können Sie die Methoden der Basisklasse um die neuen Eigenschaften der abgeleiteten Klasse erweitern.

Zugriff auf das gleichnamige Mitglied der Basisklasse

Um in einer abgeleiteten Klasse oder von außen über ein Objekt auf die gleichnamige (öffentliche) Methode der Basisklasse zugreifen zu können, müssen Sie den Zugriff mithilfe des Bereichsoperators `::` machen:

```

00 // angestellter.cpp
01 void Angestellter::print_data() {
02     Personal::print_data();
03     std::cout << "+Zulagen:" << get_zulagen() << std::endl;
04     std::cout << "=Gesamt:" << get_zulagen() + get_gehalt()
           << "\n\n";
05 }
...

```



```

06 // main.cpp
07 Angestellter angestellter01;
...
08 angestellter01.Personal::print_data();

```

Hier greifen Sie innerhalb der Methode `Angestellter::print_data()` auf die gleichnamige Methode `Personal::print_data()` zu (Zeile 02) und rufen diese auf. Damit ersparen Sie sich die Tipparbeit, die bereits gemacht wurde, und erweitern die Ausgabe von `Personal::print_data()` und die Ausgabe der Eigenschaften von der abgeleiteten Klasse `Angestellter`. Der Bereichsoperator in Zeile 02 ist hier sehr wichtig, weil Sie sonst beim Aufruf der Methode `Angestellter::print_data()` innerhalb dieser Methode durch ein `print_data()` ohne den Bereichsoperator erneut `Angestellter::print_data()` aufrufen würden. Dieser Vorgang würde sich jetzt endlos wiederholen. Dieses immer wiederkehrende Aufrufen der eigenen Methode wird als Endlosrekursion bezeichnet.

In Zeile 08 wird demonstriert, wie Sie auch nur die Eigenschaften der Basisklasse mithilfe eines Objekts der abgeleiteten Klasse ausgeben können, wenn diese Verwendung auch eher unüblich ist.

Redefinition auch für Eigenschaften

Auch wenn hier nur Methoden redefiniert wurden, so ist in der Praxis auch möglich, Eigenschaften (Daten) zu redefinieren. Hierbei gelten dieselben Regeln, die eben bei den Methoden beschrieben wurden. In der Praxis werden allerdings Eigenschaften eher selten redefiniert.

9.5 Auf- und Abbau von abgeleiteten Klassen

Nachdem wir uns ausgiebig mit den Methoden und Eigenschaften bei der Vererbung beschäftigt haben, fehlen Ihnen noch die Kenntnisse zum Auf- und Abbau von abgeleiteten Klassen – im Klartext: zu den Konstruktoren und Destruktoren.

9.5.1 Aufbau von Objekten

Die Regeln sind im Grunde einfach: Die abgeleitete Klasse ist immer verantwortlich für das Anlegen eines Objekts, weil die abgeleitete Klasse gewöhnlich alle Eigenschaften der (indirekten oder direkten) Basisklasse(n) enthält. Der Aufbau eines Objekts erfolgt dabei stets von oben nach unten in der Klassenhierarchie. Somit wird immer erst das Objekt der direkten (bzw. indirekten) Basisklasse angelegt, ehe die abgeleitete Klasse an der Reihe ist.

Betrachten Sie folgenden Codeausschnitt:

```
class Personal { ... };
class Angestellter : public Personal { ... };
class Leitender_Angestellter : public Angestellter { ... };
```

Hier würden die Konstruktoren in folgender Reihenfolge aufgerufen:

1. Konstruktor von `Personal`
2. Konstruktor von `Angestellter`
3. Konstruktor von `Leitender_Angestellter`

9.5.2 Basisklasse initialisieren

Um das Thema besser zu verstehen, beschränken wir uns nur auf die abgeleitete Klasse `Angestellter` und deren Basisklasse `Personal`. Hier haben wir ja folgende Klassendefinitionen:

```
00 // kapitel09/vererbung04/personal.h
...
01 class Personal {
02     std::string name;        // Name
03     unsigned int pers_nr;    // Personalnummer
04     unsigned int gehalt;     // monatliches Grundgehalt
05 public:
06     Personal();
07     Personal(std::string n, unsigned int pn, unsigned int g);
...
08 };
...
...
09 // kapitel09/vererbung04/angestellter.h
...
10 class Angestellter : public Personal {
11     unsigned int zulagen;    // Leistungszulagen
12 public:
13     Angestellter( unsigned int zu=0, std::string n="",
14                 unsigned int pn=0, unsigned int g=GG );
...
14 };
```

Der Hauptfokus bei diesen Beispielen liegt auf den Konstruktoren in den Zeilen 06 und 07 von *personal.h* und in Zeile 13 von *angestellter.h*. Den Umgang mit gewöhnlichen Konstruktoren einer Klasse haben Sie ja bereits umfassend in Abschnitt 7.4.2,

»Konstruktoren«, kennengelernt. Da Sie jetzt wissen, dass die abgeleitete Klasse verantwortlich ist, die Eigenschaften der Basisklasse zu initialisieren, sollen hier jetzt die Möglichkeiten beschrieben werden, die sich daraus ergeben.

Initialisieren mit Methoden

Die schlechteste Möglichkeit, mit dem Konstruktor `Angestellter` die Eigenschaften der Basisklasse `Personal` zu initialisieren, sieht wie folgt aus:

```
02 Angestellter::Angestellter(unsigned int zu, std::string n,
                           unsigned int pn, unsigned int g){
03     set_zulagen(zu);
04     set_name(n);
05     set_pers_nr(pn);
06     set_gehalt(g);
07 }
```

Bei dieser Version des Konstruktors `Angestellter` setzen wir die Werte für die Eigenschaften mithilfe der Zugriffsmethoden der Basisklasse. Hier die einzelnen Schritte, wie die Initialisierung ausgeführt wird:

1. Der Standardkonstruktor `Personal()` der Basisklasse wird aufgerufen, um die Eigenschaften mit dem Standardwert zu initialisieren.
2. Jetzt wird das Objekt `Angestellter` angelegt und hier über die Methode `set_zulagen()` initialisiert.
3. Danach wird der restliche Code innerhalb des Konstruktors `Angestellter` ausgeführt und initialisiert die Eigenschaften der Basisklasse `Personal` **erneut**, nur jetzt mit den Methoden und übergebenen Werten – also doppelte Arbeit durch zweifache Initialisierung.

Hier haben Sie denselben Nachteil, den Sie schon bei der Erzeugung von Objekten mit Teilobjekten kennengelernt haben. Vielleicht erinnern Sie sich, was Sie in dem Fall mit dem Element-Initialisierer effektiver machen konnten. Das Thema wurde im Abschnitt »Konstruktoren effektiver initialisieren mit Element-Initialisierer« in Abschnitt 7.4.2 umfassend beschrieben.

Basis-Initialisierer

Da unsere Basisklasse auch einen Konstruktor mit allen Parametern enthält, sollten Sie diesen auch aufrufen. Das funktioniert nach demselben Prinzip wie schon beim Element-Initialisierer, nur wird es hier sinngemäß als Basis-Initialisierer bezeichnet. Die Syntax ist identisch zum Element-Initialisierer:

```
Abgeleitet::Abgeleitet(parameter, ...) : Basis(parameter, ...) {...}
```

Auch hier wird der Basis-Initialisierer hinter den Parameterklammern der abgeleiteten Klasse nach einem Doppelpunkt als Konstruktor-Aufruf der Basisklasse hingeschrieben. Sinngemäß übergibt man hier auch gleich die Parameter an den Konstruktor der Basisklasse.

Bezogen auf unseren Konstruktor `Angestellter` können Sie den Basis-Initialisierer für `Personal` wie folgt implementieren:

```
01 Angestellter::Angestellter(unsigned int zu, std::string n,
                             unsigned int pn, unsigned int g)
                             : Personal(n, pn, g) {
02     set_zulagen(zu);
03 }
```

Jetzt wird wesentlich effizienter vorgegangen, und die doppelte Arbeit in der Basis-Klasse wurde hiermit vermieden. Hier die einzelnen Schritte:

1. Es wird der Konstruktor `Personal(string, unsigned int, unsigned int)` aufgerufen, und die Eigenschaften werden gleich initialisiert.
2. Jetzt wird das Objekt `Angestellter` angelegt und hier über die Methode `set_zulagen()` initialisiert.

Basis-Initialisierer und Element-Initialisierer

Zu guter Letzt können Sie sich auch noch den Methodenaufruf sparen und die Eigenschaft(en) der abgeleiteten Klasse über eine Element-Initialisierer-Liste mitgeben. Da Basis- und Element-Initialisierer dieselbe Syntax haben und durch ein Komma voneinander getrennt werden, können Sie dies, getrennt durch ein Komma, hinter den Basis-Initialisierer schreiben und als Einzeiler wie folgt realisieren:

```
00 // kapitel09/vererbung04/angestellter.cpp
...
01 Angestellter::Angestellter(unsigned int zu, std::string n,
                             unsigned int pn, unsigned int g)
                             : Personal(n, pn, g), zulagen(zu) { }
```

Diese Version aus Basis-Initialisierer und Element-Initialisierer stellt die beste Möglichkeit dar, um die Laufzeit des Programms positiv zu beeinflussen.

9.5.3 Abbauen von Objekten

Der Abbau von Objekten mit dem Destruktor geschieht genau in der umgekehrten Reihenfolge als der Aufbau mit dem Konstruktor. Hierbei wird immer erst der Destruktor der abgeleiteten Klasse aufgerufen bis hoch zur Basisklasse.

Wann wird ein Destruktor für abgeleitete Klassen benötigt?

Für die abgeleitete Klasse wird nur ein spezieller Destruktor nötig, wenn ein Vorgang rückgängig gemacht werden soll, wie bspw. das Freigeben eines dynamisch reservierten Speichers an den Heap. Der Destruktor der Basisklasse hingegen wird immer automatisch ausgeführt und muss nicht explizit aufgerufen werden.

Um nochmal auf folgenden Codeausschnitt zurückzukommen:

```
class Personal { ... };
class Angestellter : public Personal { ... };
class Leitender_Angestellter : public Angestellter { ... };
```

Wenn Sie hier ein Objekt der abgeleiteten Klasse `Leitender_Angestellter` haben und es zerstört wird, dann geschieht dies in folgender Reihenfolge:

1. Destruktor: `Leitender_Angestellter`
2. Destruktor: `Angestellter`
3. Destruktor: `Personal`

Bei den Destrukturen ist die Sache wesentlich einfacher, weil Sie zum einen keine Parameter dafür benötigen und weil Sie Destrukturen in der Praxis so gut wie nie direkt aufrufen müssen. Daher müssen Sie sich hierbei weniger Gedanken machen, welche Klasse da von welcher abgeleitet wurde.

Zum Nachlesen

Die Destrukturen wurden in Abschnitt 7.5, »Objekte zerstören – Destrukturen«, umfassend behandelt. Ebenfalls interessant im Zusammenhang mit dem Destruktor ist Abschnitt 7.8.2, »Roher Zeiger als Element in einer Klasse«.

9.5.4 Konstruktoren erben (C++11)

Eine weitere neue Technik, die mit C++11 eingeführt wurde, ist die Vererbung von Konstruktoren. Auch hier wird wieder der Trick mit `using` verwendet, womit Sie die Konstruktoren der Basisklasse der abgeleiteten Klasse hinzufügen können. Damit kann man sich eine Menge an Tipparbeit sparen. Betrachten Sie als Beispiel den folgenden Codeausschnitt:

```
00 // kapitel09/vererbung06/main.cpp
...
01 class Basis {
02 public:
```

```

03  Basis(int ival=0) { std::cout << ival << std::endl; }
04  Basis(std::string str) { std::cout << str << std::endl; }
05  };

06  class Abgeleitet : public Basis {
07  public:
08      Abgeleitet( double dval ):Basis(0) {
09          std::cout << dval << std::endl;
10      }
11      Abgeleitet( int ival ):Basis(ival) {}
12      Abgeleitet(std::string str=""):Basis(str){}
13  };

```

Im Fokus stehen hier die Konstruktoren der abgeleiteten Klasse in den Zeilen **08** bis **12**, wo Sie mithilfe des neuen C++11-Standards und `using` nun auch die Konstruktoren der Basisklasse erben können. Am Ende sieht die Klasse `Abgeleitet` wie folgt aus:

```

00  // kapitel09/vererbung07/main_cpp11.cpp
...
01  class Abgeleitet : public Basis {
02  public:
03      using Basis::Basis;
04      Abgeleitet( double dval ):Basis(0) {
05          std::cout << dval << std::endl;
06      }
07  };
...
08  // Anwendungsbeispiele:
09  Abgeleitet a1("Teststring"); // Basis(std::string)
10  Abgeleitet a2(3.1415);       // Abgeleitet(double)
11  Abgeleitet a3(100000);       // Abgeleitet(int)

```

Durch die Vererbung der Konstruktoren in Zeile **03** hat der C++11-Code jetzt dieselbe Funktionalität wie unser Beispiel im alten C++03-Code – nur jetzt wesentlich kürzer und daher auch wesentlich weniger fehleranfällig.

Auch muss hier noch hinzugefügt werden, dass mit `using` nur *alle* Konstruktoren vererbt werden können. Es ist nicht möglich, einzelne Konstruktoren weiterzuvererben. Unerwünschte geerbte Konstruktoren müssen Sie daher gegebenenfalls in der abgeleiteten Klasse neu definieren. Und Sie müssen sich auch keine Gedanken machen, dass eine `using`-Vererbung von Konstruktoren mehr Ballast mit sich bringt. Der Compiler generiert nur Code für die Konstruktoren, welche dann im Programm auch benutzt werden. Des Weiteren erbt die abgeleitete Klasse auch alle Attribute der Konstruktoren wie die Zugriffsbeschränkungen (`public`, `private`, `protected`), `default`, `delete` usw.

Ein weiterer Nachteil der Vererbung von Konstruktoren ist es, dass auch die abgeleitete Klasse keinen impliziten Standardkonstruktor erhält. Sie müssen daher aufpassen, wenn die abgeleitete Klasse eigene oder neue Eigenschaften enthält, weil durch die neue Syntax gerne vergessen wird, diese Eigenschaften mit Werten zu initialisieren. Aber auch so etwas sollte mithilfe des neuen C++11-Standards kein Problem sein, weil hier ja die Eigenschaften bei der Deklaration schon mit einem Standardwert belegt werden dürfen (siehe Abschnitt 7.4.1, »Klassenelemente direkt initialisieren (C++11)«).

Kein Kopier- und Standardkonstruktor

Der Kopier- und Standardkonstruktor der Basisklasse wird nicht an die abgeleitete Klasse vererbt.

Befindet sich außerdem ein Konstruktor mit derselben Signatur in der abgeleiteten Klasse wie in der Basisklasse (bspw.: `Abgeleitet(int)` und `Basis(int)`), dann überdeckt auch hier wieder die abgeleitete Version die geerbte Version und es würde `Abgeleitet(int)` verwendet.

9.6 Implizite und explizite Typumwandlung in der Klassenhierarchie

Die Ist-Beziehung (*is-relationship*), womit bei einer öffentlichen Vererbung aus einer Klasse die abgeleitete Klasse um die (öffentlichen) Eigenschaften und Methoden erweitert wird, kennen Sie bereits. So ist bspw. ein `Angestellter` ein bestimmtes `Personal`.

9.6.1 Abgeleitete Objekte an Basisklassenobjekte

Diese Ist-Beziehung zwischen einer abgeleiteten Klasse und einer Basisklasse können Sie auch bei der Verwendung von Objekten benutzen. Damit ist es möglich, dass Sie einem Objekt einer Basisklasse ein Objekt einer abgeleiteten Klasse zuweisen können (Abbildung 9.4). Hierbei müssen Sie sich nicht mehr um die Umwandlung kümmern. Diese Typumwandlung geschieht implizit, wie folgender Codeausschnitt zeigen soll:

```
00 // kapitel09/vererbung08/main.cpp
01 ...
02 Angestellter angestellter01(500,"Max Mustermann",400123);
03 angestellter01.print_data();

03 Personal personal01 = angestellter01;
04 personal01.print_data();
```

In Zeile 03 findet eine implizite Typumwandlung der abgeleiteten Klasse `Angestellter` zur Basisklasse `Personal` statt. Die Basisklasse bekommt nur die Eigenschaften (`name`, `pers_nr`, `gehalt`) zugewiesen, welche diese auch selbst enthält. Die erweiterten Eigenschaften (`zulagen`) der abgeleiteten Klasse `Angestellter` werden hierbei nicht beachtet.

Weitere implizite Typumwandlungen

Die implizite Typumwandlung wird auch bei einem Zeiger (Zeiger auf einer Basisklasse bzw. Basisklassenzeiger), einer Referenz auf die Basisklasse oder bei Bedarf bei einem Funktionsaufruf für die Argumente der Funktion durchgeführt.

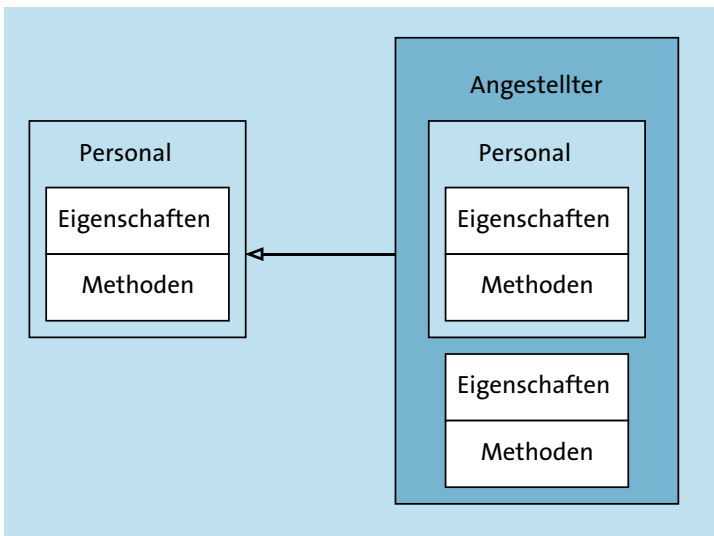


Abbildung 9.4 Durch die Ist-Beziehung können Sie der Basisklasse jederzeit abgeleitete Objekte zuweisen.

9.6.2 Basisklassenobjekt an abgeleitete Objekte

Wie bereits bei der Ist-Beziehung erwähnt, ist ein Objekt einer abgeleiteten Klasse immer auch ein spezielles Objekt der Basisklasse. Aber andersherum trifft dies nicht zu. Daher: Ein Objekt der Klasse `Personal` kann **nicht** einem Objekt der abgeleiteten Klasse `Angestellter` zugewiesen werden. Ein Beispiel:

```

01 Personal personal01("Max Mustermann", 400543, 2000);
02 personal01.print_data();
03 Angestellter angestellter01 = personal01; // Fehler!!!
  
```


Die Zuweisung in Zeile 03 von der Basisklasse `Personal` an ein Objekt der abgeleiteten Klasse `Angestellter` ist nicht möglich. Die Eigenschaften (hier nur `zulagen`) der abgeleiteten Klasse können hier nicht zugeordnet werden.

Down-Cast

Hier vielleicht noch ein paar Worte zum Down-Cast: Manchmal findet man folgenden Versuch, einen Down-Cast zu machen:

```
01 Personal personal01("Max Mustermann", 400543, 2000);
02 Personal *Pptr = &personal01;
03 Angestellter* Aptr = static_cast<Angestellter*>(Pptr);
04 Aptr->set_zulagen(200); // !!!
05 Aptr->print_data();
```

Diese Typumwandlung in Zeile 03 wird gerne als Konvertierung eines Basisklassenobjekts in ein Objekt der abgeleiteten Klasse verstanden. Dem ist leider nicht so! Hier wird ein **Zeiger** der Basisklasse in einen **Zeiger** der abgeleiteten Klasse konvertiert! Dabei erhält die abgeleitete Klasse nach wie vor nicht die fehlenden Eigenschaften. Im Speicher (!) befindet nach wie vor nur ein Objekt der Basisklasse. Es gilt auch weiterhin die Ist-Beziehung: Ein Objekt der abgeleiteten Klasse ist ein Objekt seiner Basisklasse. Aber: Ein Objekt der Basisklasse ist **kein** Objekt der von ihr abgeleiteten Klasse.

9.7 Polymorphie mithilfe von virtuellen Methoden

Manchmal wird ein konkreter Typ eines Objekts erst zur Laufzeit des Programms dynamisch angelegt und erzeugt. Um dies etwas praktischer zu beschreiben, sei hier die Basisklasse `Personal` gegeben. Von dieser Basisklasse wurde jetzt jeweils einmal die Klasse `Angestellter` (wie gehabt) und einmal die Klasse `Zeitarbeiter` abgeleitet (Abbildung 9.5). Also folgende Ausgangssituation besteht hierbei:

```
class Personal { ... };
class Angestellter : public Personal { ... };
class Zeitarbeiter : public Personal { ... };
```

Quellcode zu »zeitarbeiter.h« und »zeitarbeiter.cpp« auf CD

Den Quellcode zu `zeitarbeiter.h` und `zeitarbeiter.cpp` finden Sie auf der Buch-CD unter *kapitel09/vererbung09*. Der Quellcode unterscheidet sich kaum von dem in *angestellter.h* und *angestellter.cpp*, nur dass wir hier statt der extra Eigenschaft `zulagen` die Eigenschaften `monate` verwendet haben, womit angegeben wird, wie lange der Zeitvertrag des Beschäftigten läuft.

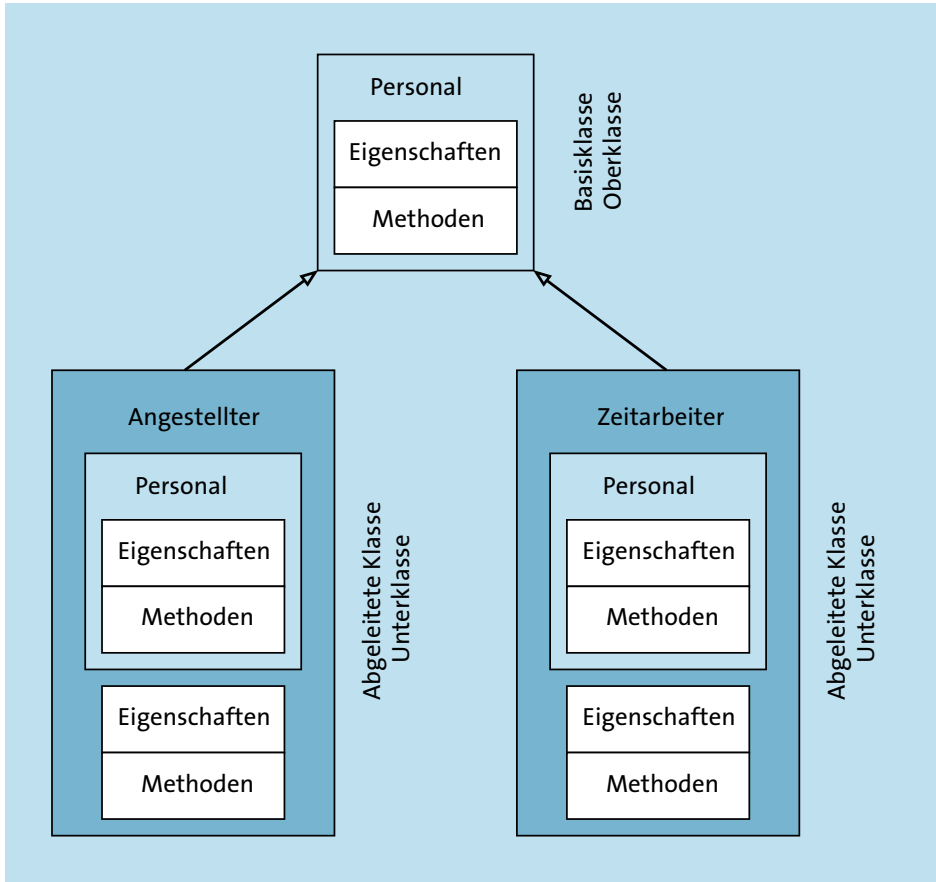


Abbildung 9.5 Die Klassen »Angestellter« und »Zeitarbeiter« haben dieselbe Basisklasse »Personal«.

Wenn Sie jetzt bspw. eine Anwendung erstellen, wo Sie ein `Personal` verwalten müssen, kann hier oft nicht gleich gesagt werden, ob hier ein `Angestellter` oder ein `Zeitarbeiter` eingestellt werden soll:

```

01 // Neuer Angestellter wird eingestellt
02 Personal *PersPtr =
    new Angestellter( 200, "Max Mustermann", 400543, 2800 );
03 PersPtr->print_data();
04 delete PersPtr;
...
05 // Hier soll es ein Zeitarbeiter sein
06 PersPtr=new Zeitarbeiter(
    24,"Max Musterfrau",400321,1950);
07 PersPtr->print_data();
  
```

```

...
08 // oder C++11-like mit std::unique_ptr
09 #include <memory> // std::unique_ptr
10 std::unique_ptr<Personal> PP(new Angestellter(
    200, "Max Mustersohn", 400544, 2900 ) );
11 PP->print_data();

```

Hier wird jeweils ein Basisklassenzeiger mit `PersPtr` verwendet. Auffällig ist, dass hier ein Zeiger vom Typ `Personal`, also der Basisklasse, verwendet wurde. Zunächst wird hier willkürlich in Zeile **02** ein neues Objekt `Angestellter` zur Laufzeit erzeugt, und in Zeile **03** werden die Daten ausgegeben. Denselben Zeiger verwenden wir auch noch, um ein `Zeitarbeiter`-Objekt in Zeile **06** zur Laufzeit anzulegen und ebenfalls auszugeben (hier in Zeile **07**). Die Ausgabe mit `print_data()` führt hier allerdings nicht zum gewünschten Ergebnis, weil die zusätzlichen Eigenschaften der abgeleiteten Klassen nicht berücksichtigt werden. In den Zeilen **10** und **11** wird außerdem auch noch demonstriert, wie Sie einen C++11-like-Basisklassenzeiger mit dem Smart Pointer `std::unique_ptr` verwenden können. Smart Pointer werden in Abschnitt 16.3, »Smart Pointer (C++11)«, behandelt.

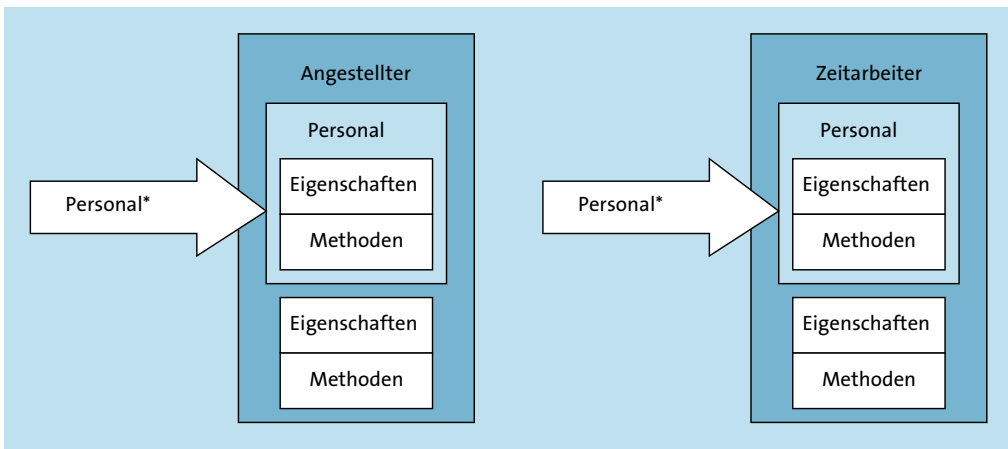


Abbildung 9.6 Ein Basisklassenzeiger, der auf Adressen von abgeleiteten Klassen verweist, hat im Grunde nur Zugriff auf die öffentlichen Eigenschaften und Methoden der Basis-Klasse.

Basisklassenzeiger

Durch die Ist-Beziehung zwischen abgeleiteten Klassen und einer Basisklasse lässt sich auch die Verwendung von Zeigern (und Referenzen) realisieren. Ein solcher Zeiger auf einer Basisklasse wird daher auch Basisklassenzeiger genannt und kann auf jedes Objekt einer von ihm abgeleiteten Klasse adressieren.

Allerdings kann ein solcher Basisklassenzeiger, ohne weitere Vorkehrungen, zunächst nur auf die (öffentlichen) Eigenschaften und Methoden der Basisklasse zugreifen (Abbildung 9.6). Zusätzlich definierte Elemente der abgeleiteten Klasse sind für den Basisklassenzeiger nicht erreichbar. Aus diesem Grund führt die Ausgabe in den Zeilen **03**, **07** und **11** lediglich dazu, dass die Methode `print_data()` der Klasse `Personal` ausgeführt wird. Die redefinierten Versionen `Angestellter::print_data()` und `Zeitarbeiter::print_data()` hingegen werden nicht verwendet.

Virtuelle Methode implementieren

Hier wird somit eine dynamische Bindung (engl. *dynamic linking*) oder spätere Bindung (engl. *late binding*) benötigt. An dieser Stelle bietet sich eine Lösung an, welche als Polymorphie (= Vielgestaltigkeit) bezeichnet wird. Eine solche Lösung kann in diesem Fall mit einer virtuellen Methode erstellt werden. Virtuelle Methoden lassen sich ganz einfach mit dem Schlüsselwort `virtual` implementieren. In unserem Beispiel brauchen Sie hierfür lediglich in der Klassendefinition von `Personal` das Schlüsselwort `virtual` vor der Deklaration von `print_data()` zu schreiben:

```
00 // kapitel09/vererbung10/personal.h
...
01 class Personal {
02     std::string name;      // Name
03     unsigned int pers_nr;  // Personalnummer
04     unsigned int gehalt;   // monatliches Grundgehalt
05 public:
...
06     virtual void print_data();
...
07 };

08 // kapitel09/vererbung10/main.cpp
...
09 Personal *PersPtr =
    new Angestellter( 200, "Max Mustermann", 400543, 2800 );
10 PersPtr->print_data(); // Angestellter::print_data()
...
11 PersPtr=new Zeitarbeiter(
    24,"Max Musterfrau",400321,1950);
12 PersPtr->print_data(); // Zeitarbeiter::print_data()
...
13 // oder C++11-like
14 #include <memory> // std::unique_ptr
```

```

15  std::unique_ptr<Personal> PP(new Angestellter(
        200, "Max Mustermann", 400543, 2800 ) );
16  PP->print_data();

```

Nur durch das Schlüsselwort `virtual` in Zeile **06** wird der Compiler jetzt veranlasst, passend zum entsprechenden Objekt die passende Methode in den Zeilen **10** bzw. **12** (bzw. **16** bei C++11) aufzurufen und auszuführen, wenn das Objekt über einen Basis-klassenzeiger angesprochen wird.

Polymorphe Klassen

Wenn eine Klasse mindestens eine virtuelle Methode enthält, darf diese als *polymorphe Klasse* bezeichnet werden.

Regeln für die Deklaration einer virtuellen Methode

Eine virtuelle Methode deklarieren Sie in der Basisklasse mit dem Schlüsselwort `virtual`. Die Definition hingegen unterscheidet sich **nicht** von anderen Methoden in der Klasse:

```

class Personal {
...
public:
    virtual void print_data(); // Deklaration
...
};
...
// Definition
void Personal::print_data() {
    std::cout << "Name      : " << get_name() << std::endl;
    std::cout << "Pers.Nr.: " << get_pers_nr() << std::endl;
    std::cout << "Gehalt   : " << get_gehalt() << std::endl;
}

```

Eine virtuelle Methode wird wie eine gewöhnliche Methode an die abgeleitete Klasse weitervererbt – auch wenn diese dort nicht erneut definiert wurde (Stichwort *Redefinition*, siehe Abschnitt 9.4.4, »Redefinition von Methoden«).

In der Praxis will man in der abgeleiteten Klasse gewöhnlich aber die virtuelle Methode der Basisklasse redefinieren und für die abgeleitete Klasse spezialisieren und um die eigenen Eigenschaften erweitern. Hierbei **muss** allerdings die redefinierte Methode dieselbe Signatur und denselben Rückgabewert haben.

Einmal virtuell, immer virtuell

Die als `virtual` deklarierte Methode der Basisklasse ist bei der Redefinition in der abgeleiteten Klasse automatisch ebenfalls wieder `virtual`, auch wenn Sie hier das Schlüsselwort nicht mehr explizit angeben. Sie können das Schlüsselwort aber erneut angeben.

```
01 class Angestellter : public Personal {
...
02 public:
03     virtual void print_data();
04 };
...
05 class Zeitarbeiter : public Personal {
...
06 public:
07     virtual void print_data();
08 };
```

Das Schlüsselwort `virtual` in der abgeleiteten Klasse zu verwenden, wie in den Zeilen **03** und **07** zu sehen, ist optional. Geben Sie das Schlüsselwort hier nicht explizit an, macht der Compiler dies implizit. Eine redefinierte virtuelle Methode der Basisklasse bleibt auch in der abgeleiteten Klasse virtuell, wenn die Signatur und der Rückgabotyp übereinstimmen.

Würden Sie allerdings nur die Methoden in den abgeleiteten Klassen mit `virtual` kennzeichnen und die gleichnamige Methode in der Basisklasse nicht, so bedeutet dies **nicht**, dass die Basisklasse damit automatisch virtuell würde.

Würden Sie eine andere Signatur oder einen anderen Rückgabewert verwenden, so würden Sie zwar eine neue Methode mit demselben Namen implementieren, allerdings wäre diese nicht virtuell. Und in Verbindung mit einem Objekt der abgeleiteten Klasse würde hiermit immer die nicht virtuelle Methode verwendet, weil diese die virtuelle Version dort überdeckt.

Virtuelle Konstruktoren gibt es nicht

Konstruktoren einer Klasse können nicht virtuell gemacht werden, Destruktoren hingegen schon. Mehr dazu gleich in Abschnitt 9.8, »Virtueller Destruktor«.

Dynamische Bindung und eine Tabelle von virtuellen Methoden

An dieser Stelle vielleicht noch ein paar Zeilen zur dynamischen Bindung, falls Sie interessiert sind, wie die oben geschilderte Polymorphie mithilfe von virtuellen

Methoden intern vor sich geht, da die Zuordnung der Methode ja erst zur Laufzeit geschieht (daher auch späte oder dynamische Bindung).

Statische Bindung

Das Gegenstück zur späten bzw. dynamischen Bindung ist die statische (oder auch frühe) Bindung. Bei einer statischen Bindung sind alle Adressen zur Kompilierzeit bekannt und werden sofort in den Maschinencode eingebaut.

Da bei einer polymorphen Klasse nicht zur Übersetzungszeit gesagt werden kann, welche Methode diese verwendet (hängt hier vom Objekt ab, worauf der Basisklassenzeiger verweist), benötigt eine Klasse mit einer virtuellen Methode einen zusätzlichen Speicherplatz, an dem die Adressen der virtuellen Methoden gespeichert werden. Dieser Speicherbereich wird als *virtuelle Methodentabelle* bezeichnet.

Des Weiteren besitzt dann jedes erzeugte Objekt einer polymorphen Klasse einen zusätzlichen Zeiger auf diese virtuelle Methodentabelle, woraus dann die Adresse der passenden virtuellen Methode ermittelt wird.

9.8 Virtueller Destruktor

Im Abschnitt zuvor haben Sie mit dem Basisklassenzeiger ein dynamisches Objekt aus einer Klassenhierarchie erzeugt. Am Ende sollten Sie das Objekt gewöhnlich auch wieder mit einer `delete`-Anweisung freigeben:

```
01 // Neuer Angestellter wird eingestellt
02 Personal *PersPtr = new Angestellter(
    200, "Max Mustermann", 400543, 2800 );
03 PersPtr->print_data();
04 delete PersPtr;
...
05 // Hier soll es ein Zeitarbeiter sein
06 PersPtr = new Zeitarbeiter(
    24, "Max Musterfrau", 400321, 1950 );
07 PersPtr->print_data();
08 delete PersPtr;
...
// bzw. in C++11 mit std::unique_ptr
09 #include <memory> // std::unique_ptr
...
10 std::unique_ptr<Personal> PP(new Angestellter(
    200, "Max Mustermann", 400543, 2800 ) );
11 PP->print_data();
```

Die Reihenfolge beim Abbau von vererbten Klassen wurde bereits in Abschnitt 9.5.3, »Abbauen von Objekten«, beschrieben. Die Konstruktoren werden von unten nach oben (von der abgeleiteten Klasse hoch zur Basisklasse) aufgerufen und die Destrukturen in umgekehrter Reihenfolge. Allerdings werden Sie sich jetzt wohl fragen, da wir hier ja einen Basisklassenzeiger vom Typ `Personal` haben, was jetzt hier wohl beim Zerstören mit dem Teil der abgeleiteten Klasse passiert? Mit dem Smart Pointer `std::unique_ptr` von C++11 wird der Destruktor automatisch am Ende des Gültigkeitsbereichs aufgerufen.

Und Ihre Frage ist berechtigt! Ein Objekt, welches über einen Basisklassenzeiger verwaltet wird, kann nur die virtuellen Methoden in der abgeleiteten Klasse verwenden. Die nicht virtuellen Methoden können nur als Basisklassen-Version verwendet werden. Bezogen auf den Destruktor bedeutet dies schlicht und einfach, dass ohne weitere Vorkehrungen nur der Destruktor der Basisklasse ausgeführt wird. Der oder die weiteren Destrukturen der davon abgeleiteten Klasse(n) werden nicht mehr ausgeführt, wenn ein Basisklassenzeiger verwendet wird.

Compilerwarnung

Wenn Sie eine polymorphe Klasse verwenden und ein Objekt zerstören, sollte sich eigentlich auch der Compiler mit einer Warnung bei Ihnen melden, dass hier ein virtueller Destruktor fehlt.

Tipp

Wenn Sie »sehen« wollen, dass der Destruktor ausgeführt wird, bauen Sie einfach einen Destruktor in die entsprechende Klasse ein mit einer Ausgabe auf dem Bildschirm, dass jetzt ein Objekt in der Klasse »X« zerstört wurde.

Nun werden Sie sich denken, wenn es mit virtuellen Methoden klappt, warum sollte man dann nicht einfach einen virtuellen Destruktor erstellen? Und tatsächlich können Sie auch dieses Problem beheben, indem Sie den Destruktor in der Basisklasse mit `virtual` deklarieren. Dadurch wird sichergestellt, dass wie bei den virtuellen Methoden auch der (oder die) nötige(n) Destruktor(en) der direkten und indirekten Basisklasse(n) aufgerufen wird (werden). Bezogen auf unser Beispiel heißt dies somit:

```
00 // kapitel09/vererbung11/personal.h
...
01 class Personal {
...
02 public:
03     virtual ~Personal() { } // virtueller Destruktor
04 };
```



```

05 // kapitel09/vererbung11/angestellter.h
...
06 class Angestellter : public Personal {
...
07 public:
08     virtual ~Angestellter() { }
...
09 };

10 // kapitel09/vererbung11/zeitarbeiter.h
...
11 class Zeitarbeiter : public Personal {
...
12 public:
13     virtual ~Zeitarbeiter() { }
...
14 };

```

Die Angabe der Destruktoren von `Angestellter` (Zeile 08) und `Zeitarbeiter` (Zeile 13) ist auch hier, wie schon bei den virtuellen Methoden, optional. Auch hier gilt, dass alles, was in der Basisklasse virtuell ist, auch bei den abgeleiteten Klassen virtuell wird – also auch die Destruktoren. Wirklich wichtig ist in diesem Beispiel das `virtual` in der Basisklasse (Zeile 03).

Daher gilt: Wenn eine Klasse als Basisklasse für weitere Klassen verwendet werden soll, die mit einem Basisklassenzeiger auf eine abgeleitete Klasse zeigen kann (und mit `delete` gelöscht werden kann), sollte diese immer auch einen virtuellen Destruktor besitzen. Ganz besonders gilt dies auch, wenn eine Basisklasse mindestens eine virtuelle Methode besitzt. Selbst wenn die Basisklasse keinen Destruktor benötigt, so sollten Sie zumindest einen virtuellen Destruktor mit einem leeren Anweisungsblock definieren.

Jenseits vom Basisklassenzeiger ist alles »normal«

An dieser Stelle muss jetzt nochmals erwähnt werden, dass die virtuellen Methoden und virtuellen Destruktoren nur im Zusammenhang mit Basisklassenzeigern verwendet werden. Wenn Sie ein ganz gewöhnliches Objekt von `Personal`, `Angestellter` oder `Zeitarbeiter` erzeugen und verwenden, dann gibt es keinen Unterschied zwischen den virtuellen und nicht virtuellen Methoden. Sprich, einem »normalen« Objekt ist es egal, ob da das Schlüsselwort `virtual` steht oder nicht. Es »funktioniert«, wie es ohne das Schlüsselwort `virtual` auch funktioniert.

9.9 Pure virtuelle Methoden und abstrakte Klassen

In diesem Abschnitt werden Sie erfahren, wie Sie reine (pure) virtuelle Methoden erstellen und daraus abstrakte Klassen bilden können.

9.9.1 Pure virtuelle Methoden

Der Unterschied zwischen einer virtuellen Methode, wie Sie diese in Abschnitt 9.7, »Polymorphie mithilfe von virtuellen Methoden«, kennengelernt haben und einer puren (oder auch reinen) virtuellen Methode (engl. *pure virtual*) ist, dass eine pure virtuelle Methode in der Basisklasse keine Definition enthält und 0 zugewiesen wird. Die Deklaration einer rein virtuellen Methode sieht demnach wie folgt aus:

```
virtual void methode() = 0; // pure virtual
```

Mit dieser Information weiß der Compiler nun, dass diese Methode in der Basisklasse nicht definiert wird, und vergibt hierfür in der Tabelle der virtuellen Methoden den Nullzeiger (nullptr).

Solche puren virtuellen Methoden werden gewöhnlich dann verwendet, wenn es in einer Basisklasse keine Aufgabe für eine Methode gibt, die dort hinpassen würde. Anstatt also einfach eine leere Methode in einer Klasse anzulegen, die nie aufgerufen wird, macht man diese pur (bzw. rein) virtuell.

Hierzu sei folgender Code einer Basisklasse als Beispiel gegeben, welcher die puren virtuellen Methoden zeigen soll:

```
00 // kapitel09/vererbung12/picture.h
...
01 class PictureFormat {
02     std::string filename;
03 public:
04     PictureFormat( const std::string& f="" ) {filename=f;}
05     virtual ~PictureFormat() {}
06     void set_name(const std::string& f){filename=f;}
07     const std::string& get_name() const {return filename;}

08     virtual void show_picture() const = 0;
09     virtual void show_picture_info() const = 0;
10     // .. usw.
11 };
```

Hier sehen Sie eine Klasse `PictureFormat`, welche Informationen zu einem Bild enthält. Hier wird, vereinfacht, nur der Name des Bildes als Eigenschaft gespeichert. Diese Klasse dient als Basisklasse für verschiedene Bildformate, welche angezeigt

werden können. Die Basisklasse selbst besitzt keine Eigenschaften, um Bilder anzeigen zu können. Da es hier unzählige (Raster-)Bildformate (wie JPEG, TIFF, GIF, PNG usw.) gibt, macht es wenig Sinn, hier alle Grafikformate zu implementieren. Dazu kommt natürlich noch die Frage, ob man sich überhaupt mit allen Grafikformaten auskennt. Mit einem einfachen `std::cout` zur Ausgabe und `std::cin` zum Einlesen ist es hier ja nicht mehr getan.

Hintergrundinfo

In der Praxis dürfte wohl kaum noch einer hergehen und das Lesen und Schreiben von Grafikdateien wie JPEG, TIFF & Co. neu schreiben. In der Regel werden hierfür fertige Bibliotheken wie `libjpeg` (für JPEG; <http://libjpeg.sourceforge.net>) oder `libtiff` (für TIFF; <http://www.libtiff.org>) usw. verwendet und in das Programm eingebunden.

Damit Sie sich jetzt nicht selbst an Ort und Stelle der Basisklasse um die Implementierungen der einzelnen Grafikformate kümmern müssen, haben wir eine Klasse mit reinen virtuellen Methoden erstellt, womit wir die Schnittstelle für weitere Grafikformate vorgeben. Und eine solche Klasse wird auch als *abstrakte Klasse* bezeichnet.

9.9.2 Abstrakte Klassen

Damit eine abstrakte Klasse auch wirklich abstrakt ist, muss mindestens eine pure virtuelle Methode enthalten sein, wie Sie eben in Abschnitt 9.9.1 erfahren haben. Enthält eine solche Klasse jetzt allerdings eine solche pure virtuelle Methode, können davon *keine Objekte* mehr instanziiert werden. Ein Objekt von `PictureFormat` wie folgt, würde vom Compiler mit einer Fehlermeldung abgelehnt werden:

```
PictureFormat file("file.jpg");
```

Ein Objekt einer abstrakten Klasse kann daher nicht erzeugt werden, weil hierbei sonst (pure virtuelle) Methoden aufgerufen werden könnten, welche gar nicht definiert wurden. Daher macht auch eine abstrakte Klasse erst Sinn, wenn davon auch abgeleitet wird.

Implementierungszwang

Wenn Sie von einer abstrakten Klasse ableiten, werden auch die reinen virtuellen Methoden geerbt. Daher müssen Sie in der abgeleiteten Klasse diese reinen virtuellen Methoden auch definieren. Ansonsten würde die abgeleitete Klasse wieder nur eine weitere abstrakte Klasse. Der Vorteil von abstrakten Basisklassen ist daher auch, dass Sie praktisch jeder Klasse, welche von dieser abgeleitet wird, eine Schnittstelle aufzwingen, die implementiert werden muss.

Im Beispiel der Klasse `PictureFormat` müssen in den davon abgeleiteten Klassen somit die Methoden `show_picture()` und `show_picture_info()` implementiert werden. Somit können auch jederzeit andere Programmierer weitere Rastergrafikformate nachimplementieren.

Damit Sie den Sinn von abstrakten Klassen auch verstehen, sollen jetzt davon zwei Klassen abgeleitet werden. In diesem Beispiel werden die Rastergrafik-Formate JPEG und TIFF implementiert – natürlich nur rein theoretisch in Form einer Dummy-Textausgabe statt der echten Rasterung einer Grafik:

```

00 // kapitel09/vererbung12/jpegformat.h
01 #include "picture.h"
...
02 class JPEGFormat : public PictureFormat {
03     std::string info;
04 public:
05     JPEGFormat(const std::string& f="",
                const std::string &i="")
                : PictureFormat(f), info(i) {}
06     void show_picture() const {
07         std::cout << "JPEG-Bild " << get_name()
                << " wird angezeigt\n";
08     }
09     void show_picture_info() const {
10         std::cout << "Informationen zum Bild "
                << get_name() << std::endl;
11         std::cout << info << std::endl;
12     }
13     void set_info(const std::string &i) { info = i; }
14 };

```

Die Definitionen wurden hier auch gleich mit in der Klassendefinition geschrieben, damit das Beispiel übersichtlich bleibt. In der Praxis teilen Sie diese natürlich wie gewöhnlich in eine Header- und Quelldatei auf. Die Headerdatei in Zeile 01 zur abstrakten Basisklasse muss hier mit rein. In Zeile 02 wird die Ableitung der abstrakten Basisklasse durchgeführt. In Zeile 05 finden Sie noch den Konstruktor, womit auch gleich die Eigenschaft der abstrakten Basisklasse initialisiert wird.

In den Zeilen 06 bis 12 werden jetzt die reinen virtuellen Methoden `show_picture()` und `show_picture_info()` aus der abstrakten Basisklasse definiert. Erst wenn Sie alle reinen virtuellen Methoden einer abstrakten Klasse in der abgeleiteten Klasse definiert haben, können Sie auch ein Objekt der abgeleiteten Klasse erzeugen. Imple-

mentieren Sie nur eine der puren virtuellen Methode nicht, dann können Sie kein Objekt der abgeleiteten Klasse erzeugen und haben nur eine weitere abstrakte Klasse.

Code bearbeiten

Die eigentlichen Methoden `show_picture()` und `show_picture_info()` sind nur als Dummy-Methoden implementiert. Statt einer Dummy-Ausgabe müssten Sie hier natürlich echte Funktionen für die Anzeige von Rastergrafik einbauen (bspw. mithilfe von `libjpeg` oder `libtiff`).

Zur Demonstration wollen wir unsere Anwendung noch um ein weiteres Grafikformat mit GIF erweitern, natürlich in diesem Beispiel auch wieder nur mit einer Dummy-Ausgabe:

```
00 // kapitel09/vererbung12/gifformat.h
01 #include "picture.h"
...
02 class GIFFormat : public PictureFormat {
03     std::string info;
04     bool transparency;
05 public:
06     GIFFormat(const std::string& f="",
07               const std::string &i="", bool t=false) :
08         PictureFormat(f), info(i), transparency(t) {}
09     void show_picture() const {
10         std::cout << "GIF-Bild " << get_name()
11                 << " wird angezeigt\n";
12     }
13     void show_picture_info() const {
14         std::cout << "Informationen zum Bild "
15                 << get_name() << std::endl;
16         std::cout << info << std::endl;
17         std::cout << "Transparenz im Bild vorhanden: "
18                 << (transparency?"Ja":"Nein") << std::endl;
19     }
20     void set_info(const std::string &i) { info = i; }
21 };

```

Basisklassenzeiger verwenden

Wie auch schon bei den Basisklassen mit virtuellen Methoden können Sie von abstrakten Klassen auch keine Objekte erzeugen (warum, wurde bereits erwähnt) und müssen auf einen Basisklassenzeiger (oder eine Referenz) zurückgreifen. Und da hier

die Basisklasse auch noch abstrakt ist, kann nur dynamisch zur Laufzeit ein Speicher für die abgeleitete Klasse reserviert werden. Ein einfaches Beispiel:

```
00 // kapitel09/vererbung12/main_basis.cpp
...
01 PictureFormat *PF_ptr =
    new JPEGFormat("file.jpg", "Mein Urlaubsfoto");
02 PF_ptr->show_picture();
03 delete PF_ptr;
04 PF_ptr = new GIFFormat("file.gif", "Banner Webseite");
05 PF_ptr->show_picture();
06 delete PF_ptr;
```

Da in Zeile **02** und **05** die Methode `show_picture()` in `PictureFormat` eine virtuelle ist, wird die richtige Version der abgeleiteten Klasse `JPEGFormat` oder `GIFFormat` auch ausgeführt.

Der Zugriff als C++11-Basisklassenzeiger mit `unique_ptr` lässt sich hingegen wie folgt realisieren:

```
#include <memory> // für std::unique_ptr
...
std::unique_ptr<PictureFormat> PF(new JPEGFormat(
    "IMG1234.jpg", "Weihnachten 2013"));
PF->show_picture();
...
```

Referenz verwenden

Unsere abstrakte Klasse lässt sich aber mithilfe einer Referenz ebenfalls als polymorphe Schnittstelle verwenden, ohne dass Objekte dynamisch zur Laufzeit angefordert werden müssen. Logischerweise können hierbei dann nur Objekte der abgeleiteten Klasse angelegt werden. Hier das Beispiel dazu mit anschließender Erläuterung:

```
00 // kapitel09/vererbung12/main_ref.cpp
...
01 #include "picture.h"
02 #include "jpegformat.h"
03 #include "gifformat.h"
...
04 void display_picture_on_screen(PictureFormat& fmt) {
05     fmt.show_picture();
06 }

07 void display_info_on_screen(PictureFormat& fmt) {
```

```

08     fmt.show_picture_info();
09 }
...
10 JPEGFormat file01("IMG0001.jpg", "Urlaub 2013");
11 GIFFormat file02("banner.gif", "Werbebanner für Homepage");
12 display_picture_on_screen(file01);
13 display_info_on_screen(file01);
14 display_picture_on_screen(file02);
15 display_info_on_screen(file02);

```

In den Zeilen **04** bis **06** bzw. **07** bis **09** wurden zwei globale Funktionen implementiert, welche die abstrakte Klasse `PictureFormat` als Referenz definieren und die dort rein virtuellen Methoden `show_picture()` und `show_picture_info()` aufrufen. Bei den Funktionsaufrufen in den Zeilen **12** bis **15** brauchen Sie sich um nichts mehr zu kümmern. Dank der Polymorphie wird schon innerhalb der Funktionen in den Zeilen **05** und **08** die passende Methode für das `JPEGFormat`- oder `GIFFormat`-Objekt verwendet.

Basisklassenzeiger und Referenzen

Nochmals sollte erwähnt werden, dass Sie aus abstrakten Klassen kein Objekt erzeugen können, weil diese Methoden enthalten, welche gar nicht definiert wurden und somit ein Zugriff auf undefinierte Methoden erfolgen könnte. Daher ist es von enormer Bedeutung, zu verstehen, dass der Zugriff auf eine abstrakte Basisklasse ausschließlich über einen Basisklassenzeiger oder eine Referenz erfolgen kann.

9.9.3 Reine Interface-Klassen

Reine Interface-Klassen (Schnittstellen-Klassen) werden als abstrakte Klassen bezeichnet, welche keinerlei Eigenschaften und lediglich pure virtuelle Methoden enthalten. Eine solche Klasse ist absolut pur und hat keinerlei Implementationen. Ein einfacher Codeausschnitt dazu:

```

class Cloud {
public:
    virtual void read()=0;
    virtual void write()=0;
    virtual void error()=0;
...
};

```

Hier definieren wir eine pure virtuelle Basisklasse `Cloud`, wo, in der davon abgeleiteten Klasse, die Methoden `read()`, `write()` und `error()` immer implementiert werden müssen. Wie diese implementiert werden, bleibt dem Programmierer der davon

abgeleiteten Klasse überlassen, aber diese müssen geschrieben werden, wenn von der Klasse abgeleitet wird.

9.9.4 Typinformationen zur Laufzeit

In C++ wurde eine sehr strenge Typüberprüfung zur Übersetzungszeit einprogrammiert. Trotzdem gibt es manchmal Fälle, in denen eine Typidentifikation erst zur Laufzeit bestimmt werden kann. Mit den virtuellen Klassen haben Sie gesehen, wie Typen erst zur Laufzeit erzeugt werden. In C++ finden Sie eine solche Laufzeit-Typidentifikation (engl. *runtime type identification*, kurz RTTI) mit dem Operator `typeid()` und der Klasse `type_info`.

Mit diesem Operator kann zur Laufzeit die Typidentifikation von einem Objekt ermittelt werden. Damit können Typen miteinander verglichen oder sogar eine implementierungsabhängige Zeichenkette des Typs ausgegeben werden. Im Falle eines Fehlers zeigt `typeid` auf `nullptr`, und es wird eine Ausnahme (= `bad_typeid`) geworfen. Ausnahmen werden in Kapitel 10, »Ausnahmebehandlung (Exceptions)«, besprochen.

Welche Methoden Ihnen mit dem Operator `typeid()` zur Verfügung stehen, sehen Sie in Tabelle 9.4.

Methode	Beschreibung
<code>operator==</code>	Gibt <code>true</code> zurück, wenn zwei Typen gleich sind, ansonsten <code>false</code> .
<code>operator!=</code>	Gibt <code>true</code> zurück, wenn zwei Typen ungleich sind, ansonsten <code>false</code> .
<code>name()</code>	Gibt einen implementierungsabhängigen Namen des Typs zurück.

Tabelle 9.4 Einige Methoden der Klasse »`type_info`«

Der folgende Codeausschnitt soll Ihnen zeigen, wie Sie einen Typ eines Objekts zur Laufzeit ermitteln können. Diese Technik kann manchmal nützlich zum Ermitteln oder Vergleichen von polymorphen Objekten sein, da diese erst zur Laufzeit erzeugt werden.

```

00 // kapitel09/vererbung13/main.cpp
01 #include <typeinfo>
...
02 void Type(PictureFormat& typ) {
03     std::cout << "Typ:" << typeid(typ).name() << std::endl;
04 }
```



```

05 bool BType(PictureFormat& typ1, PictureFormat& typ2) {
06     return ( typeid(typ1) == typeid(typ2) );
07 }
...
08 JPEGFormat file01("IMG0001.jpg", "Urlaub 2013");
09 GIFFFormat file02(
    "banner.gif", "Werbebanner für Homepage");
10 PictureFormat *PF = new JPEGFormat(
    "IMG1234.jpg", "Weihnachten 2013");
11 Type(file01);
12 Type(file02);
13 Type(*PF);

14 if( BType(file01, *PF) ) {
15     std::cout << "Beide Typen sind gleich\n";
16 }
...

```

Die Headerdatei `<typinfo>` in Zeile **01** wird unbedingt benötigt. In den Zeilen **02** bis **04** wurde eine globale Funktion `Type()` definiert, welche mithilfe des Operators `typeid()` und der Methode `name()` den Typ der Klasse auf dem Bildschirm ausgibt. Die Ausgabe des Strings ist allerdings implementierungsabhängig. Die Funktion wird in den Zeilen **11**, **12** und **13** aufgerufen.

In Zeile **05** bis **07** wird die Funktion `BType()` definiert, wo wir zwei Typen miteinander vergleichen. Sind beide Typen gleich, wird `true`, ansonsten `false` zurückgegeben. Die Funktion wird in Zeile **14** bis **16** getestet, wo beide Typen gleich sind.

Die Ausgabe des Programms (implementierungsabhängig) war hier:

```

Typ: 10JPEGFormat
Typ: 9GIFFFormat
Typ: 10JPEGFormat
Beide Typen sind gleich

```

Wem die Ausgabe mit der Methode `name()` nicht gefällt, der kann sich einen benutzerdefinierten String dazu selbst zusammenbasteln und ausgeben. In C++11 bietet sich hierzu die Methode `hash_code()` an, welche allerdings auch eine entsprechende Standardcontainerklasse mit assoziativem Inhalt benötigt. Im Beispiel wird hierzu die neu in C++11 eingeführte Containerklasse `unordered_map` dafür verwendet:

```

00 // kapitel09/vererbung13/main_cpp11.cpp
...
01 #include <unordered_map>

```

```

02 #include <typeinfo>
...
03 JPEGFormat file01("IMG0001.jpg", "Urlaub 2013");
04 GIFFormat file02(
    "banner.gif", "Werbebanner für Homepage");
05 PictureFormat *PF = new JPEGFormat(
    "IMG1234.jpg", "Weihnachten 2013");

06 std::unordered_map<std::size_t, std::string> tnames;
07 tnames[typeid(JPEGFormat).hash_code()] = "JPEGFormat";
08 tnames[typeid(GIFFormat).hash_code()] = "GIFFormat";

09 std::cout << "file01: "
    << tnames[typeid(file01).hash_code()] << std::endl;
10 std::cout << "file01: "
    << tnames[typeid(file02).hash_code()] << std::endl;
11 std::cout << "*PF : "
    << tnames[typeid(*PF).hash_code()] << std::endl;

```

Das Programm bei der Ausführung:

```

file01: JPEGFormat
file01: GIFFormat
*PF : JPEGFormat

```

In der gängigen Praxis von einfachen Anwendungen wird die Laufzeit-Typidentifikation eher selten benötigt. Typische Anwendungsfälle dafür liegen eher im Bereich der Erstellung von Bibliotheken, Frameworks oder Entwurfsmustern.

Nicht nur auf Klassen beschränkt

Es soll hier noch erwähnt werden, dass `typeid()` nicht nur auf Klassen beschränkt ist, sondern auch mit den eingebauten Basisdatentypen verwendet werden kann. Damit lässt sich praktisch auch auf Gleichheit von beliebigen Typen hin testen.

9.9.5 Überschreiben erzwingen mit `override (C++11)`

Ein schwer auffindbarer Fehler ist es, wenn eine virtuelle Methode in der abgeleiteten Klasse nicht korrekt redefiniert wird. Solche Dinge geschehen gerne mal, wenn man einen Tippfehler macht, falsche Argumenttypen verwendet oder sich die virtuelle Methode in der Basisklasse verändert. Im Folgenden ein einfaches Beispiel zur Erläuterung, was hiermit gemeint ist:

```

00 // kapitel09/vererbung14/main.cpp
01 #include <memory> // std::unique_ptr
...
02 class Basis {
03 public:
04     virtual void methode(int) { std::cout << "Basis\n"; }
05 };

06 class Abgeleitet : public Basis {
07 public:
08     virtual void methdoe(int){std::cout << "Abgeleitet\n";}
09 };
...
10 std::unique_ptr<Abgeleitet> aptr(new Abgeleitet{});
11 aptr->methode(12345);

```

Durch Zeile 11 wird hier die Methode `Basis::methode()` ausgeführt, wie die Textausgabe bestätigt, anstatt, wie hier eigentlich beabsichtigt, die Methode `Abgeleitet::methode()`. Bei genauerem Studieren des Codes fällt auf, dass es `Abgeleitet::methode()` gar nicht gibt, weil hier ein dummer Tippfehler passiert ist und `methdoe()` statt `methode()` geschrieben wurde. Solche Kleinigkeiten können bei umfangreichen Projekten wirklich zu schwer auffindbaren Fehlern führen, weil es ja kein echter Fehler ist. In C++11 können Sie sich davor jetzt schützen, indem Sie hinter redefinierten virtuellen Methoden einfach das neue Schlüsselwort `override` schreiben:

```

08 virtual void methdoe(int) override {
    std::cout << "Abgeleitet\n";
}

```

Jetzt würde sich der Compiler bei Ihnen beschweren, dass Sie eine Methode zum Überschreiben markiert haben, die aber in Wirklichkeit gar nichts überschreibt. Dies kann jetzt folgende Gründe haben:

- ▶ Ihnen ist ein einfacher Tippfehler bei den Bezeichnern unterlaufen.
- ▶ Es liegt ein Fehler bei den Typen der Argumente oder dem Rückgabewert vor (bspw. `methode(double)` statt `methode(int)`). Auch ein eventuelles `const` hinter den Methoden gehört dazu.
- ▶ Die Methode in der Basisklasse wurde geändert.
- ▶ Sie haben das Schlüsselwort `virtual` bei der Methode in der Basisklasse vergessen.

Wie override funktioniert

Wenn Sie eine Methode mit `override` markieren, dann machen Sie dies, weil Sie sich bewusst sind, dass diese virtuell ist. Der Compiler überprüft daher lediglich für Sie, ob Sie diese Methode aus der Basisklasse in der abgeleiteten Klasse redefiniert haben. Ist dies nicht der Fall, so bekommen Sie lediglich eine Fehlermeldung, dass diese Methode noch nicht überschrieben (redefiniert) wurde. Ob es sich bei der Ursache hierfür um einen Tippfehler oder Fehler bei den Argumenten handelt, kann der Compiler hier letztendlich auch nicht wissen. Kurz: Wenn es keine virtuelle überschriebene Methode gibt, dann meldet der Compiler einen Fehler.

Virtuellen Destruktor überschreiben

Aus Abschnitt 9.8, »Virtueller Destruktor«, wissen Sie, dass man bei einer Basisklasse, welche mindestens eine virtuelle Methode besitzt, den virtuellen Destruktor definieren sollte. Vergisst man diesen, kann das dazu führen, dass der Speicher nicht mehr ordentlich aufgeräumt wird und das Programm den Speicher immer mehr füllt (= Memory Leak). Oft erstellt man aber eine Klasse, welche zunächst gar nicht als Basisklasse programmiert wurde, und hat sich gar keine Gedanken um den virtuellen Destruktor gemacht. Hier gibt es jetzt einen einfachen Trick, indem Sie einfach in der abgeleiteten Klasse den Destruktor mit `override` versehen. Wurde der virtuelle Destruktor der Basisklasse gar nicht virtuell implementiert, wird Ihnen der Compiler dies mitteilen:

```
01 class Abgeleitet : public Basis {
02 public:
03     void methode(int) override{ std::cout << "Abgeleitet\n"; }
04     ~Abgeleitet() override {};
05 };
```

Wurde in der virtuellen Basisklasse der Destruktor nicht mit `virtual` implementiert, wie dies gewöhnlich empfohlen wird, dann wissen Sie es spätestens, wenn Sie in der abgeleiteten Klasse den Destruktor mit `override` kennzeichnen, wie Sie dies in Zeile **04** sehen können. Ist der Destruktor von `Basis` nicht virtuell, meldet sich der Compiler auch hier wieder mit dem Hinweis, dass `Abgeleitet::~~Abgeleitet()` zum Überschreiben markiert, aber nicht überschrieben wurde – und somit wäre der virtuelle Destruktor nicht implementiert.

9.9.6 Nicht mehr überschreiben mit final (C++11)

Sehr nützlich bei der Vererbung ist auch das neue Schlüsselwort `final`, womit sich Methoden und Klassen kennzeichnen lassen, damit diese nicht mehr überschrieben

bzw. abgeleitet werden können. Auch hierzu ein einfacher Codeausschnitt, welcher das neue Schlüsselwort `final` etwas verständlicher machen soll:

```

01 class EMail {
02 public:
03     virtual void print_header() const;
04     virtual ~EMail() {}
05 };

06 class EMailHeader : public EMail {
07 public:
08     virtual void print_header() const final override {}
09     virtual ~EMailHeader() override {}
10 };

11 class MyEMail : public EMailHeader {
12 public:
13     virtual void print_header() const {} // Fehler !!!
14 };

15 class EMailText final : public EMail {
16     std::string data;
17 public:
18     virtual void print_header() const final override {}
19     virtual ~EMailText() override {}
20     void print_text() {}
21 };

22 class SniffingEMail : public EMailText { // Fehler !!!
23     std::string data;
24     ...
25 };

```

In Zeile **08** wird die Methode `print_header()` mit `final` markiert und kann daher nicht mehr überschrieben werden. In den Zeilen **11** bis **14** leiten wir von der Klasse `EMailHeader` ab und versuchen in Zeile **13** trotzdem nochmals die Methode `print_header()` zu überschreiben. Der Compiler wird dies mit einer Fehlermeldung quittieren. In Verbindung mit Methoden, wie in Zeile **08** geschehen, sollten Sie `final` immer mit `override` kombinieren.

Dasselbe lässt sich auch mit Klassen realisieren. In Zeile **15** erklären wir die Klasse `EMailText` zur finalen Klassen, von der nicht mehr abgeleitet werden kann und soll.

Da unsere Klasse hier reine Textdaten einer E-Mail enthält, wollen wir auch gar nicht mehr, dass jemand davon ableiten kann und so die Daten weiterverarbeiten (bspw. versenden, mitlesen etc.) kann. Die Ableitung in Zeile 22 an die Klasse `SniffingEMail` wird daher vom Compiler wieder mit einer Fehlermeldung quittiert, mit dem Hinweis, dass von der Klasse `EMailText` nicht mehr abgeleitet werden kann, weil diese bereits `final` ist.

override mit final für Methoden

Sie sollten daher `override` mit `final` und `virtual` zusammen verwenden, denn würden Sie eine Methode verwenden, welche gar nicht überschrieben wird (bspw. durch Tippfehler), macht diese im Sinne von virtuellen Methoden keinen Sinn. Dann bräuchte man ja eine Methode auch gar nicht erst virtuell zu machen.

9.10 Mehrfachvererbung

Neben der gewöhnlichen Vererbung gibt es in C++, im Gegensatz zu einigen anderen OOP-Sprachen, auch die Mehrfachvererbung. Damit können Sie eine neue Klasse quasi von mehreren Basisklassen ableiten.

Als Beispiel sei hier eine Klasse `ZweiRad` und eine Klasse `MotorFahrzeug` gegeben. Beide Klassen enthalten die nötigen Eigenschaften, um daraus ein `Motorrad` zu machen. Eine solche mehrfache Ableitung können Sie in der Praxis wie folgt realisieren:

```
class Motorrad : public ZweiRad, public Motorfahrzeug {
    // ...
};
```

Hinter dem Doppelpunkt der Klasse müssen Sie jetzt lediglich, jeweils getrennt durch ein Komma, die einzelnen Basisklassen angeben, von denen geerbt werden soll. Für jede dieser Basisklassen müssen Sie außerdem die entsprechenden Zugriffsrechte `public`, `protected` oder `private` verwenden. Bei einer solchen Mehrfachvererbung werden die Eigenschaften und Methoden wie schon bei der gewöhnlichen einfachen (1:1-)Vererbung mit denselben Regeln weitervererbt. Sollten Sie bei einer der Basisklassen, welche durch ein Komma getrennt werden, kein Zugriffsrecht verwenden, wird auch hier automatisch `private` verwendet.

Eine `public`-Vererbung, die hier meistens verwendet wird, entspricht auch wieder der Ist-Beziehung wie eine einfache Vererbung. In diesem Beispiel **ist** somit ein `Motorrad` ein `ZweiRad` **und ist** auch ein `MotorFahrzeug` (Abbildung 9.7).

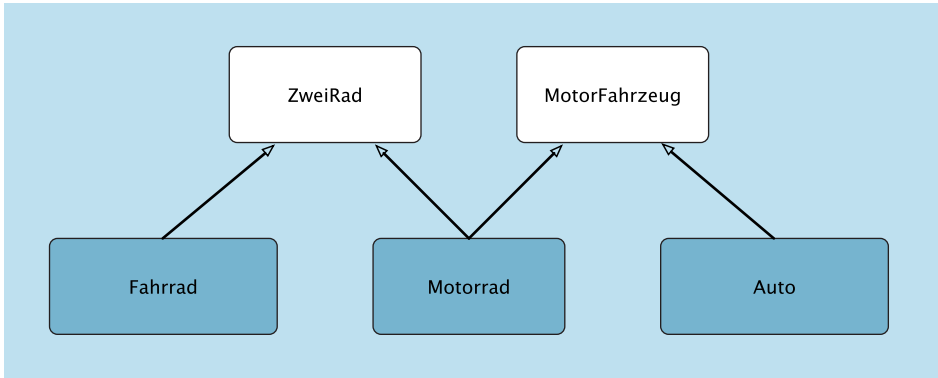


Abbildung 9.7 Beispiel einer Mehrfachvererbung

Aus den beiden Klassen »ZweiRad« und »MotorFahrzeug« kann über eine Mehrfachvererbung die abgeleitete Klasse »Motorrad« erstellt werden. Die abgeleitete Klasse »Fahrrad« wurde einfach von »ZweiRad« und »Auto« wurde von »MotorFahrzeug« abgeleitet – diese Ableitungen sind hier für die Einführung aber nicht von Interesse und dienen nur der Veranschaulichung einer Klassenhierarchie.

Praxisbeispiel: Mehrfachvererbung

Hier nun der Quellcode zur eben beschriebenen Mehrfachvererbung mit anschließender Erläuterung. Zur besseren Übersicht wurde das Beispiel sehr minimal gehalten, und alles wurde auch wieder `inline` in der Klassendefinition geschrieben. Zusätzlich wollen wir hierbei noch eine weitere Basisklasse von `ZweiRad` und `MotorFahrzeug` erstellen, der Sie allerdings vorerst keine Beachtung zu schenken brauchen. Wir benötigen diese, um anschließend ein paar Worst-Case-Szenarios zu beschreiben. Hier zunächst die Basisklasse `Fahrzeug`:

```

00 // kapitel09/vererbung15/fahrzeug.h
...
01 class Fahrzeug {
02     unsigned int geschwindigkeit;
03     public:
04     Fahrzeug(unsigned int kmH=0) : geschwindigkeit(kmH){}
05     void fahren() { std::cout << "Fahrzeug fährt\n"; }
06     void print() {
07         std::cout << "Geschwindigkeit: "
08             << geschwindigkeit << std::endl;
09     }
10     void set_geschwindigkeit(unsigned int kmH) {
11         geschwindigkeit=kmH;
12     }
13     unsigned int get_geschwindigkeit() const {

```

```

12     return geschwindigkeit;
13 }
14 };

```

Jetzt zu unseren beiden Klassen `ZweiRad` und `MotorFahrzeug`, welche wir jeweils auch von der Klasse `Fahrzeug` abgeleitet haben:

```

00 // kapitel09/vererbung15/zweirad.h
01 #include "fahrzeug.h"
...
02 class ZweiRad : public Fahrzeug {
03     unsigned int gewicht;
04 public:
05     ZweiRad(unsigned int kmH=0, unsigned int kg=0) :
        Fahrzeug(kmH), gewicht(kg) {}
06     void print() {
07         Fahrzeug::print();
08         std::cout << "Gewicht: " << gewicht << std::endl;
09     }
10     unsigned int get_gewicht() const {return gewicht;}
11     void set_gewicht(unsigned int kg) {gewicht=kg;}
12 };

00 // kapitel09/vererbung15/motorfahrzeug.h
01 #include "fahrzeug.h"
...
02 class MotorFahrzeug : public Fahrzeug {
03     unsigned int leistung;
04     unsigned int gewicht;
05 public:
06     MotorFahrzeug(unsigned int kmH=0, unsigned int ps=0,
        unsigned kg=0):Fahrzeug(kmH), leistung(ps), gewicht(kg) {}
07     void print() {
08         Fahrzeug::print();
09         std::cout << "Gewicht: " << gewicht << std::endl;
10         std::cout << "Leistung: " << leistung << std::endl;
11     }
12     unsigned int get_leistung() const{return leistung;}
13     void set_leistung(unsigned int ps) {leistung=ps;}
14     unsigned int get_gewicht() const {return gewicht;}
15     void set_gewicht(unsigned int kg) {gewicht=kg;}
16 };

```

An sich finden Sie hier in diesen Klassen nichts Neues, was Sie jetzt nicht auch schon von der normalen Vererbungen her kennen.

Jetzt haben wir die gewünschte Ausgangssituation, um eine Mehrfachvererbung von den Klassen `ZweiRad` und `MotorFahrzeug` an die Klasse `Motorrad` durchzuführen, wodurch wir dann den Zustand von Abbildung 9.7 erreicht hätten:

```
00 // kapitel09/vererbung15/motorrad.h
01 #include "motorfahrzeug.h"
02 #include "zweirad.h"
...
01 class Motorrad : public ZweiRad, public MotorFahrzeug {
02     std::string marke;
03 public:
04     Motorrad(unsigned int kmH=0, unsigned int kg=0,
               unsigned int ps=0, std::string m="") :
               ZweiRad(kmH, kg), MotorFahrzeug(kmH, ps, kg),
               marke(m) {}
05     void print() {
06         MotorFahrzeug::print();
07         std::cout << "Marke: " << marke << std::endl;
08     }
09     std::string get_marke() const{return marke;}
10     void set_marke(std::string m) {marke=m;}
11 };
```

In Zeile **01** erbt die Klasse `Motorrad` von den Klassen `ZweiRad` und `MotorFahrzeug` alle (öffentlichen) Eigenschaften und Methoden. Die Klassen `ZweiRad` und `MotorFahrzeug` sind jetzt die Basisklassen von `Motorrad`. In Zeile **02** haben wir einen einfachen String als Eigenschaft mit der Marke des Motorrads hinzugefügt. Beim Konstruktor von `Motorrad` rufen wir auch gleich die Konstruktoren für `ZweiRad` und `MotorFahrzeug` auf und initialisieren auch gleich die Eigenschaften von der Klasse `Motorrad`. In den Zeilen **05** bis **11** sehen Sie außerdem noch einige Methoden der Klasse `Motorrad`.

In der Methode `Motorrad::print()` in den Zeilen **05** bis **09** können Sie schon mal erkennen, dass es jetzt etwas umständlicher wird, auf die Datenelemente zuzugreifen. Da beide Basisklassen `ZweiRad` und `MotorFahrzeug` jeweils `print()` definiert haben, müssen Sie in Zeile **06** mit dem Bereichsoperator und dem Klassennamen angeben, welche `print()`-Methode Sie hier verwenden wollen.

Warum wir nicht auch noch `ZweiRad::print()` verwenden, dürfte ersichtlich werden, wenn Sie die Methoden `ZweiRad::print()` und `MotorFahrzeug::print()` vergleichen. Beide Versionen greifen jeweils auf deren Basisklasse auf `Fahrzeug::print()` zurück und bei der Verwendung von `Motorrad::print()` würde praktisch zweimal die Geschwindigkeit der indirekten Basisklasse `Fahrzeug` ausgegeben und zweimal das Gewicht (einmal für `MotorFahrzeug` und einmal für `ZweiRad`). Würde eine der beiden

Eigenschaften für das Gewicht verändert, würden Sie dann noch zwei verschiedene Werte zurückerhalten.

9.10.1 Mehrdeutigkeiten

Bisher wurden die indirekte Basisklasse von `Motorrad` und die direkte Basisklasse von `ZweiRad` und `MotorFahrzeug` noch bewusst ignoriert. Wollen Sie jetzt allerdings das Objekt `Motorrad` verwenden, wird die Sache komplizierter. Zum einen haben Sie in den direkten Basisklassen Methoden mit denselben Namen, und jede unserer beiden Basisklassen `ZweiRad` und `MotorFahrzeug` hat jetzt eine eigene Basisklasse `Fahrzeug` (siehe Abbildung 9.8). Denn, nur weil die Klasse `Motorrad` von `ZweiRad` und von `MotorFahrzeug` geerbt hat, ändert sich nicht die Ist-Beziehung zwischen `ZweiRad` und `Fahrzeug` und `MotorFahrzeug` und `Fahrzeug`.

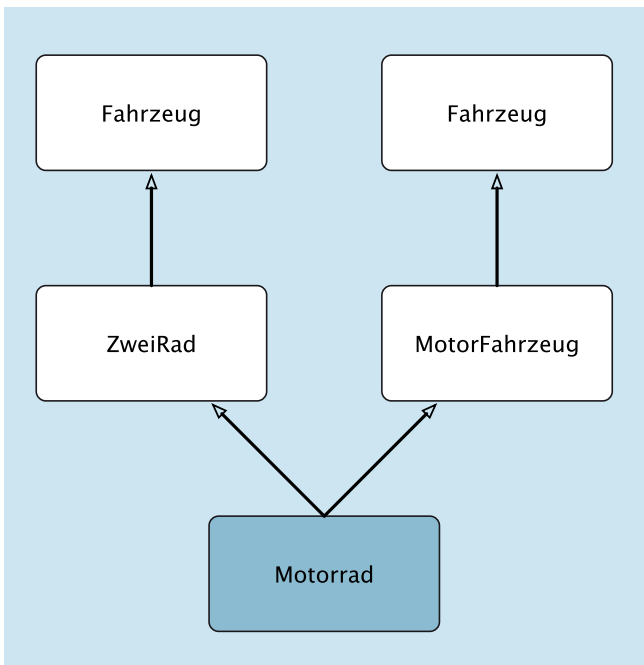


Abbildung 9.8 Bei der Mehrfachvererbung haben jetzt die Klassen »ZweiRad« und »MotorFahrzeug« jeweils eine eigene Klasse »Fahrzeug«, was nicht unbedingt erwünscht sein dürfte.

Zunächst wollen wir uns noch dem Thema der Mehrdeutigkeit widmen. Eine solche Mehrdeutigkeit entsteht bei einer Mehrfachvererbung, wenn verschiedene Basisklassen dieselben Bezeichner für Klassenelemente haben. In unserem Fall ist die Mehrdeutigkeit bei den Methoden `set_gewicht()` und `get_gewicht()` gegeben, welche sowohl in `ZweiRad` als auch in `MotorFahrzeug` definiert wurden.

Mehrdeutigkeiten auflösen

Solche Mehrdeutigkeiten müssen Sie mit dem entsprechenden Klassennamen und Bereichsoperator (Scope-Operator) auflösen (`Klasse::Bezeichner`), damit auch der Compiler bei der Übersetzung diese Klassenmitglieder korrekt auflösen kann. Als Beispiel soll jetzt die Verwendung eines Objekts vom Typ `Motorrad` instanziiert und verwendet werden:

```
00 // kapitel09/vererbung15/main.cpp
...
01 Motorrad bike(220);
02 bike.set_leistung(55);
03 bike.MotorFahrzeug::set_gewicht(300);
04 bike.set_marke("Honda Hero");
05 bike.print();
```

In Zeile **01** legen wir ein neues Objekt `Motorrad` mit 220 km/h Geschwindigkeit an. In den Zeilen **02** bis **04** setzen wir die Eigenschaften für das Objekt. In Zeile **03** helfen wir dem Compiler und lösen die Mehrdeutigkeit der Methode `MotorFahrzeug::set_gewicht()` und `ZweiRad::set_gewicht()` auf, indem wir uns für die Version von `MotorFahrzeug` entschieden haben.

Worst-Case-Szenario

In diesem Beispiel wurden die Mehrdeutigkeiten in den Basisklassen `MotorFahrzeug` und `ZweiRad` mit Absicht implementiert. In der Praxis werden Sie aber wohl eher selten auf zwei Basisklassen mit identischen Bezeichnern stoßen. Das Beispiel sollte nur das Problem der Mehrdeutigkeit in der Mehrfachvererbung demonstrieren und außerdem auch zeigen, dass eine Mehrfachvererbung die Sache durchaus komplizierter machen kann.

Mehrfachvererbung in der Kritik

Sie werden vermutlich neben diesem Buch auch im World Wide Web unterwegs sein und hierbei auch viele Contras gegen Mehrfachvererbung gelesen haben. Die Mehrfachvererbung steht häufig in der Kritik, weil Sie nun mal den Code eher verkompliziert als vereinfacht. Programmiersprachen, wie bspw. Java, haben die Mehrfachvererbung auch gar nicht erst implementiert. Wenn Sie Mehrfachvererbung in Ihren Projekten einsetzen, sind Sie dafür verantwortlich, eventuell auftretende Namenskonflikte und Mehrdeutigkeiten aufzulösen. Die meisten Dinge können Sie auch ohne Mehrfachvererbung lösen.

Oft hat man aber keinen Einfluss auf die Basisklassen und kommt vielleicht nicht ganz um die Mehrfachvererbung herum. Aber Sie wissen jetzt, worauf es dabei ankommt.

Aufruf von Konstruktoren und Destruktoren

Die Reihenfolge der Konstruktor-Aufrufe geschieht nach wie vor von oben nach unten in der Klassenhierarchie. Da bei einer Mehrfachvererbung mehrere Basisklassen für eine Klasse vorhanden sind, werden hier dann die Konstruktoren der Basis-Klasse von links nach rechts aufgerufen. Also der Konstruktor der Klasse, die hinter dem Doppelpunkt der Klassendefinition als Erstes angegeben wurde, wird als Erstes aufgerufen, gefolgt von der nächsten Klasse usw. Im Beispiel der Mehrfachvererbung von `Motorrad`

```
class Motorrad : public ZweiRad, public MotorFahrzeug { ... };
```

würde praktisch zuerst (von links nach rechts) das Teilobjekt `ZweiRad` und dann das Teilobjekt `MotorFahrzeug` angelegt. Erst dann wird der Konstruktor von `Motorrad` ausgeführt.

Gemäß der Klassendefinition muss auch der Konstruktor der mehrfach abgeleiteten Klasse entsprechend angepasst sein. Hier müssen Sie alle Konstruktoren der Basis-Klasse aufrufen. Hier listen Sie hinter den Parameterklammern, gefolgt von einem Doppelpunkt, die Konstruktoren der Basisklassen auf:

```
// Konstruktor der mehrfach abgeleiteten Klasse
Motorrad(unsigned int kmH=0, unsigned int kg=0,
         unsigned int ps=0, std::string m="") :
    ZweiRad(kmH, kg), // 1.Konstruktor
    MotorFahrzeug(kmH, ps, kg), // 2.Konstruktor
    marke(m) {}
```

Die Reihenfolge der Basis-Initialisierer spielt übrigens bei der Definition des Konstruktors keine Rolle. Sie könnten hier bei diesem Konstruktor durchaus die Position von `ZweiRad(kmH, kg)` und `MotorFahrzeug(kmH, ps, kg)` tauschen. Die Reihenfolge des Aufrufes bleibt nach wie vor so, wie sie bei der Definition der mehrfach abgeleiteten Klasse angegeben wurde.

Wenn bei der Definition des Konstruktors der Basis-Initialisierer fehlt, wird der Standardkonstruktor der Basisklasse aufgerufen.

Für die Destruktoren gilt auch bei der Mehrfachvererbung dasselbe wie schon bei der einfachen Vererbung, diese werden in umgekehrter Reihenfolge als die Konstruktor aufgerufen.

9.10.2 Virtuelle Basisklassen (virtuelle Vererbung)

Bezogen auf Abbildung 9.8 ist in unserem Fall jetzt durch die Mehrfachvererbung die indirekte Basisklasse `Fahrzeug` zweifach vorhanden. Das ist allerdings in diesem Bei-

spiel eher unerwünscht und führt spätestens auch dann wieder zu Problemen, wenn Sie versuchen, das Motorrad zu fahren(). In der indirekten Basisklasse Fahrzeug ist die Methode fahren() enthalten. Würden Sie jetzt einfach Folgendes verwenden

```
01 Motorrad bike;
...
02 bike.set_geschwindigkeit(80); // 80 km/h
03 bike.fahren(); // losfahren? Fährt nicht!
```

würde das Motorrad weder auf 80 km/h gesetzt, noch führe es los. Dafür würde der Compiler auch hier wieder eine Mehrdeutigkeit der Methoden set_geschwindigkeit() und fahren() melden. Ist eigentlich auch klar, wenn Sie Abbildung 9.8 betrachten. Denn der Compiler weiß jetzt nicht, ob er wie ein ZweiRad oder wie ein MotorFahrzeug losfahren soll. Natürlich könnten Sie auch hier wieder hergehen und die Mehrdeutigkeit für den Compiler selbst wie folgt auflösen:

```
01 Motorrad bike;
...
02 bike.MotorFahrzeug::set_geschwindigkeit(80);
03 bike.MotorFahrzeug::fahren(); // Jetzt geht es los
```

In C++ können Sie das Problem der mehrfach vorhandenen indirekten Basisklasse jetzt beheben, indem Sie eine indirekte virtuelle Basisklasse aus Fahrzeug machen. Wenn die Rede von virtuell ist, dann ist auch das Schlüsselwort virtual nicht weit. Folgendermaßen machen Sie aus der indirekten Basisklasse Fahrzeug jetzt eine virtuelle Basisklasse:

```
00 // kapitel09/vererbung16/zweirad.h
...
01 class ZweiRad : virtual public Fahrzeug {
...
02 };

00 // kapitel09/vererbung16/motorfahrzeug.h
...
01 class MotorFahrzeug : virtual public Fahrzeug {
...
02 };

00 // kapitel09/vererbung16/main.cpp
...
01 Motorrad bike;
```

```

...
02 bike.set_geschwindigkeit(80); // 80 km/h
03 bike.fahren(); // Jetzt geht es !!!

```

Mehr ist hier nicht nötig, um die Mehrdeutigkeit aufzulösen. Der Aufruf in den Zeilen 02 und 03 ist jetzt dank des Schlüsselwortes `virtual` bei den Ableitungen in den Zeilen 01 bei `zweirad.h` und `motorfahrzeug.h` ohne Mehrdeutigkeit für den Compiler auflösbar und führt zu keiner Fehlermeldung mehr. Für die Klassen `ZweiRad` und `MotorFahrzeug` hat diese Virtualität der direkten Basisklasse `Fahrzeug` noch keinen Effekt. Diese virtuelle Vererbung bekommt erst ihren Sinn bei der Mehrfachvererbung:

```

00 // kapitel09/vererbung16/motorrad.h
...
01 class Motorrad : public ZweiRad, public MotorFahrzeug {
...
02 };

```

Erst nach dieser Mehrfachvererbung in Zeile 01 steht die virtuelle Basisklasse `Fahrzeug` nur noch einmal im Speicher (siehe Abbildung 9.9).

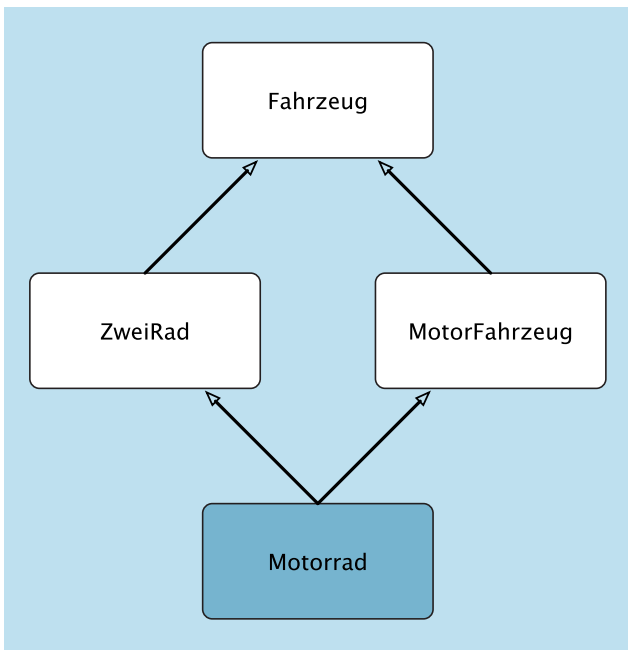


Abbildung 9.9 Dank der virtuellen Vererbung ist die indirekte Basisklasse nur noch einmal im Speicher vorhanden und wird bei einer Mehrfachvererbung nicht mehr mehrfach vererbt.

Virtuelles Erben erst bei Mehrfachvererbung

Die virtuelle Vererbung wirkt sich erst bei einer Mehrfachvererbung aus. Würden Sie bspw. ein Objekt einer Klasse `ZweiRad` oder `MotorFahrzeug` instanziiieren, so hätte jede Instanz nach wie vor eine eigene Basisklasse `Fahrzeug`.

Einmal virtuell, immer virtuell

Die virtuelle Basisklasse bleibt selbstverständlich bei weiteren Ableitungen virtuell. Im Beispiel hätte somit jede von `ZweiRad` und `MotorFahrzeug` abgeleitete Klasse die Klasse `Fahrzeug` als virtuelle Basisklasse.

Aufruf von Konstruktoren und Destruktoren virtueller Basisklassen

Hier gilt im Grunde dasselbe, wie schon zum Aufruf der Konstruktoren bei der Mehrfachvererbung geschrieben wurde: **Nur**, wenn es hier in der Klassenhierarchie noch eine virtuelle Basisklasse gibt, dann wird dieser Konstruktor noch vor den Konstruktoren einer nicht virtuellen Basisklasse ausgeführt. Die nicht virtuellen Basisklassen werden erst danach wie gehabt in der Reihenfolge der Mehrfachvererbung ausgeführt.

Kapitel 10

Ausnahmebehandlung (Exceptions)

Mit der Ausnahmebehandlung haben Sie die Möglichkeit, auf »unerwartete« Aktionen eines Codeabschnitts zu reagieren. Gewöhnlich sind solche Aktionen Fehlersituationen, die beim Programm auftreten können.

In einem Programm können laufend irgendwelche »unerwarteten« Fehler auftreten, welche den Programmablauf stören bzw. das Programm instabil werden lassen. Um einige Beispiele zu nennen:

- ▶ mangelnder Speicher
- ▶ falscher Zugriff auf Dateien
- ▶ Fehler bei arithmetischen Berechnungen
- ▶ Zugriff auf unerlaubte Adressen (vor allem bei Zeigern und Referenzen)
- ▶ falsche Eingabe bei der Standardeingabe des Anwenders
- ▶ Fehler auf dem System

Um auf einen Fehler zu reagieren, könnten Sie jetzt wie folgt vorgehen:

```
00 // kapitel10/ausnahmen01/main.cpp
...
01 double division(int x, int y, bool& check) {
02     if(y==0) {
03         check=false;
04         return 0.0;
05     }
06     check=true;
07     return static_cast<double>(x)/y;
08 }
...
09 bool check;
10 double res = division(10, 0, check);
11 if( !check ) {
12     std::cerr << "Fehler: Division durch 0\n";
13 }
14 else {
```



```

15     std::cout << "Ergebnis : " << res << std::endl;
16 }

```

Zugegeben, der Code erfüllt seinen Zweck und verhindert erfolgreich eine Division durch 0 in der Funktion `division()`. Trotzdem ist diese Art, einen Fehler, wie hier mit der `bool`-Variablen, zu behandeln, relativ aufwendig und auch eher C-typisch.

Bisher dürften Sie auf Fehler mit einer der folgenden drei Möglichkeiten reagiert haben:

- ▶ Programm abbrechen: Mit dieser Möglichkeit behandeln Sie zwar den Fehler, aber häufig ist eine Beendigung des Programms nicht immer wünschenswert.
- ▶ Fehler ignorieren (und hoffen, dass es schon gut gehen wird): Die wohl schlechteste, aber leider auch eine gerne verwendete Möglichkeit.
- ▶ Einen eigenen Fehlercode verwenden: Hierbei werden Funktionen auf deren Rückgabewert hin überprüft, und es wird in einen entsprechenden Fehlercode verzweigt. Allerdings lässt sich dies wiederum nicht mit **Konstruktoren** verwenden, weil diese keinen Rückgabewert zurückgeben.

Von den hier erwähnten Möglichkeiten erscheint die dritte am vernünftigsten, aber es ist nur schwer möglich, so auf alle Ausnahmen zu reagieren, die bei einer Software auftreten. Und zudem führt dies zu einem recht unübersichtlichen Code, weil Fehlerbehandlungen und der normale Code miteinander vermischt werden.

In C++ haben Sie jetzt mit der Ausnahmebehandlung (engl. *exception handling*) die Möglichkeit, auf unerwartete Aktionen des Programms zu reagieren. Korrekt eingesetzt, kann die Verwendung von Ausnahmen dem Programm helfen, wesentlich flexibler auf unerwartete Fehler zu reagieren und sich wesentlich toleranter gegenüber Fehlern zu verhalten. Das Hauptziel der Ausnahmebehandlung ist es, dass ein Fehler erkannt und entsprechend behandelt wird.

10.1 Prinzip der Ausnahmebehandlung

Das Prinzip der Behandlung von Ausnahmen (engl. *exceptions*) ist schnell erklärt. Sie aktivieren eine Ausnahmebehandlung und richten ein entsprechendes Codekonstrukt ein, welches auf diese Ausnahmen an einer zentralen Stelle reagiert. Tritt jetzt ein Fehler auf, werfen Sie eine Ausnahme und behandeln diese an der eben eingerichteten zentralen Stelle. Die Ausführung des Programms wird während der Behandlung der Ausnahme unterbrochen. Abhängig von der Ausnahme behandeln Sie diese und reagieren jetzt entsprechend darauf. Nach der Abarbeitung der Ausnahme **kann** wieder mit dem eigentlichen Programmablauf fortgefahren werden. Der Vorteil, die Fehler auf diese Art und Weise zu behandeln, ist es, dass man den Fehler an einer **zentralen Stelle** behandeln kann und nicht einen Rückgabewert nach

dem anderen überprüfen muss, womit das Programm immer umfangreicher und komplizierter gemacht würde.

10.2 Ausnahmebehandlung implementieren

Um das Prinzip der Ausnahmebehandlung realisieren zu können, bietet C++ einige Schlüsselwörter an. Hier ist zunächst das Schlüsselwort `try` zu nennen, womit Sie einen Codeblock für die Ausnahmebehandlung einrichten und den Code ausführen, welcher eine Ausnahme werfen könnte. Im Fall eines Fehlers gibt es dann das Schlüsselwort `throw`, womit Sie die Ausnahme werfen, falls ein Fehler aufgetreten ist. Um auf diese Ausnahme zu reagieren, müssen Sie diese abfangen. Hierfür richten Sie hinter dem `try`-Block einen oder mehrere Blöcke mit dem Schlüsselwort `catch` ein und der entsprechenden Ausnahme, die Sie dort behandeln wollen.

Hierzu ein theoretischer Pseudocode, wie eine solche Ausnahmebehandlung eingerichtet werden kann:

```

01 try {
02     // Hier kommt der Code hin,
03     // der eine Ausnahme mit throw werfen könnte
04 }
05 catch( AusnahmeTyp01 bezeichner ) {
06     // Ausnahmebehandlung für AusnahmeTyp01
07 }
08 catch( AusnahmeTyp02 bezeichner ) {
09     // Ausnahmebehandlung für AusnahmeTyp02
10 }
11 catch( ... ) {
12     // Ausnahmebehandlung für unbekannte Ausnahmen
13 }
```

Im `try`-Block in den Zeilen **01** bis **04** führen Sie den kritischen Code aus, der eine Ausnahme mittels `throw` werfen könnte. Wurde ein Fehler vom Typ `AusnahmeTyp01` geworfen, wird der Fehler in den Zeilen **05** bis **07** abgefangen und kann behandelt werden. Gleiches gilt, wenn eine Ausnahme vom Typ `AusnahmeTyp02` geworfen wird, für die Zeilen **08** bis **10**. Wurde hingegen eine schwerwiegende Ausnahme geworfen oder eine Ausnahme, für die kein `catch()` eingerichtet wurde, kann (optional) ein `catch(...)` mit den drei Punkten (= Auslassungszeichen oder auch Ellipsenzeichen) als Parameter verwendet werden, wie Sie dies in den Zeilen **11** bis **13** sehen können. Mit einem solchen `catch(...)` können Sie zumindest verhindern, dass sich das Programm in einem undefinierten Zustand verliert, und können so zumindest darauf reagieren.

10.2.1 Ausnahme einleiten – try

Genug der Theorie. In den nächsten Abschnitten soll Schritt für Schritt eine einfache Ausnahmebehandlung implementiert werden. Eine Ausnahmebehandlung wird mit dem Schlüsselwort `try` (engl. für prüfen, versuchen) eingeleitet. Gleich nach `try` verwendet man einen Anweisungsblock, in den der kritische Code reingeschrieben wird, welcher einen Fehler in Form einer Ausnahme auslösen kann. Wo die Ausnahme jetzt ausgelöst wird, spielt keine Rolle. Dies kann durchaus innerhalb des Anweisungsblocks (wenn auch eher unüblich) oder einer darin aufgerufenen Funktion(en)/Methode(en) sein.

try ist ein eigener Gültigkeitsbereich!

Speicherobjekte, die in einem `try`-Block definiert werden, sind auch nur darin gültig! Ein Zugriff von außerhalb auf diese Speicherobjekte ist nicht möglich. Wollen Sie ein Speicherobjekt außerhalb des `try`-Blocks verwenden, müssen Sie es auch außerhalb des `try`-Blocks definieren. Beachten Sie allerdings, dass im Falle eines Fehlers das Objekt, welches den Fehler ausgelöst hat, nicht mehr gültig ist.

Hierzu ein Beispiel mit dem einleitenden Ausnahmeblock `try`, das anschließend genauer erläutert werden soll:

```
00 // kapitel10/ausnahmen02/main.cpp
...
01 double division(int x, int y) {...}

02 void input_vals( int& x, int& y) {...}
...
03 try {
04     int val1=0, val2=0;
05     input_vals(val1, val2); // kritischer Code
06     double res = division(val1, val2); // krit. Code
07     std::cout << "Ergebnis : " << res << std::endl;
08 }
...
```

Der `try`-Anweisungsblock startet in Zeile 03 und endet in Zeile 08. Innerhalb des Blocks haben wir jetzt quasi eine zentrale Fehlerbehandlungsstelle eingerichtet, welche eine einfache Division durchführen soll. Hierbei werden die Funktionen `input_vals()` und `division()` ausgeführt. `input_vals()` wird für die Eingabe der Werte und `division()` für die arithmetische Berechnung verwendet. Beide Funktionen können eine Ausnahme werfen.

10.2.2 Ausnahme werfen – throw

Um jetzt eine Ausnahme innerhalb eines try-Blocks zu werfen, wird das Schlüsselwort `throw` (dt. werfen) mit folgender Syntax verwendet:

```
throw TYP;
```

Mit `throw` wird die Instanz eines beliebigen Datentyps geworfen. Natürlich, und vor allem, werden hier auch eigene Klassen (genauer Fehlerklassen) definiert, geworfen und abgefangen. Dies wird in Abschnitt 10.3, »Ausnahmeklasse implementieren«, genauer behandelt.

Hierzu erweitern wir jetzt das Beispiel, indem wir an passender Stelle eine Ausnahme mit `throw` werfen:

```
00 // kapitel10/ausnahmen02/main.cpp
...
01 double division(int x, int y) {
02     if(y==0) {
03         throw std::string("Division durch 0");
04     }
05     return static_cast<double>(x)/y;
06 }

07 void input_vals( int& x, int& y) {
08     std::cout << "Dividend: ";
09     if( ! (std::cin >> x)) {
10         throw std::string("Fehler bei Eingabe: Dividend");
11     }
12     std::cout << "Divisor: ";
13     if( ! (std::cin >> y)) {
14         throw std::string("Fehler bei Eingabe: Divisor");
15     }
16 }
...
17 try {
18     int val1=0, val2=0;
19     input_vals(val1, val2); // kritischer Code
20     double res = division(val1, val2); // krit. Code
21     std::cout << "Ergebnis : " << res << std::endl;
22 }
...
```

Hier wird jeweils eine Ausnahme in den Zeilen 10 und 14 geworfen, wenn die Eingabe über das Stream-Objekt `std::cin` in der Funktion `input_vals()` eine ungültige Zahl

war. Ebenso wird eine Ausnahme in Zeile 03 geworfen, wenn der Divisor gleich 0 ist, weil eine Division durch 0 nicht erlaubt ist. In allen drei Fällen werfen wir eine Instanz der Klasse `std::string` mit der Fehlermeldung.

10.2.3 Ausnahme abfangen – catch

Zu guter Letzt benötigen Sie noch ein Codekonstrukt, um Ausnahmen abzufangen, die in einem im try-Block geschriebenen Code eventuell geworfen wurden. Hierfür wird eine catch-Anweisung (dt. fangen) verwendet. Die catch-Anweisung besitzt folgende Syntax:

```
catch( DTYP Bezeichner ) {
    // Ausnahme behandeln
}
```

Für den Parameter müssen Sie einen passenden Datentyp verwenden, der im try-Block geworfen werden könnte. `catch()` nimmt nur geworfene Ausnahmen an, wenn

- ▶ der Typ innerhalb von `catch(TYP)` dem Typ der geworfenen Ausnahme entspricht und
- ▶ der Typ innerhalb von `catch(TYP)` eine direkte oder indirekte Basisklasse vom Typ der geworfenen Ausnahme ist.

Mit diesem Grundwissen können Sie jetzt die Ausnahme in unserem Beispiel abfangen und auswerten:

```
00 // kapitel10/ausnahmen02/main.cpp
...
01 try {
02     int val1=0, val2=0;
03     input_vals(val1, val2); // kritischer Code
04     double res = division(val1, val2); // krit. Code
05     std::cout << "Ergebnis : " << res << std::endl;
06 }
07 catch(std::string msg) {
08     std::cerr << "Fehler : " << msg << std::endl;
09 }
10 std::cout << "Hier geht es weiter\n";
...
```

Wenn jetzt eine der Funktionen im try-Block eine Ausnahme wirft, wird `catch()` in den Zeilen 07 bis 09 diese jetzt abfangen. Als Typ wurde hier `std::string` verwendet, weil unsere Funktionen `input_vals()` und `division()` eine Instanz eines solchen Typs werfen, wie dies im Codeausschnitt vom Abschnitt 10.2.2 definiert wurde. In die-

sem Beispiel ist die einzige Fehlerbehandlung eher trivialer Natur, indem wir den String (die Fehlermeldung) einfach nur ausgeben. Wenn das Programm nicht in einer der Ausnahmebehandlungen beendet wurde, fährt die Ausführung des Programms hinter der Ausnahmebehandlung (hier Zeile 10) fort.

Das Programm beider Ausführung:

```
J.Wolf $./main
Dividend: a
Fehler : Fehler bei Eingabe: Dividend
J.Wolf $ ./main
Dividend: 10
Divisor: 0
Fehler : Division durch 0
J.Wolf $ ./main
Dividend: 10
Divisor: 3
Ergebnis : 3.33333
J.Wolf $
```

Feinschliff 1: Fehlercode werfen und auswerten

Bevor Sie erfahren, wie Sie mehrere `catch`-Anweisungen in Ihr Programm einbauen und verwenden können, will ich Ihnen noch eine übersichtlichere und beliebte Lösung liefern. Bei umfangreichen Projekten können oft mehr als nur zwei bis drei Ausnahmen geworfen werden. Immer einfach nur einen Text mit der Fehlermeldung zu werfen, abzufangen und auszugeben ist eher selten ausreichend.

Mehrere `catch`-Anweisungen zu schreiben und zu verwenden, wäre eine weitere Lösung, könnte aber schnell unübersichtlich (die Lesbarkeit leidet) und komplexer werden und die Wartung des Programms erheblich aufwendiger machen. In der Praxis werden daher gerne einfache Fehlernummern (oder auch Fehlercodes) geworfen und ausgewertet. Allerdings sind reine Angaben als Nummern auch nicht sehr hilfreich und wiederum fehleranfällig, weshalb sich hier `enum`-Konstanten (siehe Abschnitt 4.7, »Aufzählungstypen (C++11-Version)«) besser eignen würden. Bezogen auf unser Beispiel mit der Division könnte dies wie folgt definiert werden:

```
00 // kapitel10/ausnahmen03/main.cpp
...
01 enum class Err{ DIV0, INP1, INP2 };

02 double division(int x, int y) {
03     if(y==0) {
04         throw Err::DIV0;
05     }
```

```

06     return static_cast<double>(x)/y;
07 }

08 void input_vals( int& x, int& y) {
09     std::cout << "Dividend: ";
10     if( ! (std::cin >> x)) {
11         throw Err::INP1;
12     }
13     std::cout << "Divisor: ";
14     if( ! (std::cin >> y)) {
15         throw Err::INP2;
16     }
17 }
...
18 try {
19     // wie kapitel10/ausnahmen02/main.cpp
...
20 }
21 catch(Err errcode) {
22     switch( errcode ) {
23         case Err::DIV0 :
24             std::cerr << "Fehler: Division durch 0\n"; break;
25         case Err::INP1 :
26             std::cerr << "Fehler: Eingabe Dividend\n"; break;
27         case Err::INP2 :
28             std::cerr << "Fehler: Eingabe Divisor\n"; break;
29     }
30 }

```

In Zeile 01 erstellen wir die enum-Fehlercodes. In der Praxis sollten Sie diese Codes auch noch ausreichend dokumentieren. Beim try-Anweisungsblock ändert sich nichts. Wenn jetzt ein Fehler auftritt, wird nur noch ein konstanter Fehlercode geworfen (siehe Zeilen 04, 11 und 15). Mit dem catch() in den Zeilen 21 bis 29 fangen wir diesen Fehlercode ab, werten diesen in einer switch-Anweisung entsprechend aus und reagieren hier gegebenenfalls entsprechend darauf (hier wieder lediglich mit einer einfachen Ausgabe auf dem Bildschirm).

10.2.4 Mehrere Ausnahmen abfangen

Nicht immer ist es ausreichend, nur einen Typ mit catch() abzufangen. Zum Abfangen mehrerer Ausnahmen müssen Sie lediglich hinter dem letzten catch-Block einen neuen Block mit catch() einleiten. Hierbei können Sie beliebig viele catch-Anwei-

sungen anfügen. Natürlich können Sie beliebig viele Typen mit `throw` als Ausnahme werfen, aber in der Praxis werden eher selten einfache Basisdatentypen geworfen und gefangen. Hier schreibt man eher Fehlerklassen, welche es dann abzufangen gilt. Aber dazu mehr in Abschnitt 10.3, »Ausnahmeklasse implementieren«.

Zur Demonstration schreiben wir unser bisher verwendetes Codebeispiel wieder etwas um:

```
00 // kapitel10/ausnahmen04/main.cpp
...
01 double division(int x, int y) {
02     if(y==0) {
03         throw std::string("Division durch 0");
04     }
05     return static_cast<double>(x)/y;
06 }

07 void input_vals( int& x, int& y) {
08     std::cout << "Dividend: ";
09     if( ! (std::cin >> x)) {
10         throw -1;
11     }
12     std::cout << "Divisor: ";
13     if( ! (std::cin >> y)) {
14         throw "Divisor: Keine Zahl";
15     }
16 }
...
17 try {
18     ...
19     // wie kapitel10/ausnahmen03/main.cpp
20 }
21 catch(int val) {
22     std::cout << "Fehlercode erhalten:" << val << std::endl;
23 }
24 catch(std::string msg) {
25     std::cout << "Fehler: " << msg << std::endl;
26 }
```

Im Beispiel werden jetzt drei verschiedene Ausnahmen geworfen. Der `try`-Block ist wie in den Beispielen zuvor und wurde daher nicht abgedruckt. In den Zeilen 21 bis 26 finden Sie jetzt zwei `catch`-Anweisungen. Das erste `catch()` in den Zeilen 21 bis 23 fängt Integer-Werte ab, wie im Beispiel in Zeile 10 geworfen würde, wenn für den Dividend keine gültig Zahl eingegeben würde. Die andere `catch`-Anweisung in den

Zeilen 24 bis 26 fängt ein `std::string` ab, welches in diesem Beispiel in Zeile 03 geworfen würde, wenn eine Division durch 0 gemacht würde.

In Zeile 14 wird außerdem noch eine dritte Ausnahme von Typ `const char*` geworfen, auf die wir hier nicht vorbereitet sind. Würden Sie somit bei der Eingabe des Divisors eine ungültige Zahl eingeben, wird eine Ausnahme geworfen, für die wir kein `catch()` eingerichtet haben. Das Programm wird somit hierbei intern mit der Funktion `terminate()` beendet.

Kein passendes catch()

Wenn kein `catch` auf den geworfenen Typ der Ausnahme passt, wird intern die Funktion `terminate()` aufgerufen und das Programm beendet. Dasselbe gilt auch, wenn eine Ausnahme geworfen wird und überhaupt keine Ausnahmebehandlung implementiert wurde.

10.2.5 Unbekannte oder alternative Ausnahmen abfangen

Im Beispiel zuvor haben Sie gesehen, dass bei einer Ausnahme, die nicht behandelt wird, das Programm mit `terminate()` beendet wird. Die Reihenfolge der Abarbeitung von verschiedenen `catch`-Anweisungen erfolgt ähnlich wie bei einer `if-else`-Konstruktion von oben nach unten. Es werden der Reihenfolge nach alle `catch`-Anweisungen überprüft, ob der Typ zur geworfenen Ausnahme passt. In der Praxis können Sie hierbei auch eine (optionale) `catch`-Anweisung am Ende einrichten, welche sich um die unbekannten Typen von Ausnahmen kümmert (für die kein passendes `catch` eingerichtet wurde). Hierbei müssen Sie lediglich am Ende eine `catch`-Anweisung mit drei Punkten als Parameter einfügen.

Alternatives catch(...) mit Vorbehalt

Verwenden Sie das alternative `catch(...)` immer nur, wenn es unbedingt nötig ist. Das Problem dabei ist, dass hiermit wirklich jede Ausnahme abgefangen wird, was auch programminterne Fehler sein können. Hiermit würden Sie gegebenenfalls wichtige Fehlermeldungen von programminternen Ausnahmen unterdrücken.

Hierzu die Erweiterung von Listing *kapitel10/ausnahmen04/main.cpp* um das alternative und optionale `catch(...)`:

```
00 // kapitel10/ausnahmen05/main.cpp
...
01 double division(int x, int y) { ... }

02 void input_vals( int& x, int& y) {
```

```

...
03     std::cout << "Divisor: ";
04     if( ! (std::cin >> y)) {
05         throw "Divisor: Keine Zahl";
06     }
07 }
...
08 try {
    ...
09     // wie kapitel10/ausnahmen03/main.cpp
10 }
11 catch(int val) {
12     std::cout << "Fehlercode erhalten:" << val << std::endl;
13 }
14 catch(std::string msg) {
15     std::cout << "Fehler: " << msg << std::endl;
16 }
17 catch(...) {
18     std::cout << "Unbekannte Ausnahme wurde geworfen\n";
19 }

```

Jetzt wird in diesem Beispiel das optionale `catch(...)` in den Zeilen 17 bis 19 verwendet, um die Ausnahme aus Zeile 05 abzufangen. Auch wenn andere unbekannte Ausnahmen im `try`-Block geworfen werden, werden diese jetzt mit `catch(...)` abgefangen.

Alternatives catch am Ende

Auch wenn es klar sein sollte, das alternative `catch(...)` muss am Ende als letzte `catch`-Anweisung stehen. Würde diese weiter vorne stehen, so würden die anderen `catch`-Anweisungen dahinter nicht mehr ausgeführt. Allerdings sollte der Compiler ohnehin eine Meldung ausgeben, wenn das alternative `catch(...)` nicht am Ende steht.

10.2.6 Aufräumarbeiten bei der Ausnahmebehandlung

Im Grunde brauchen Sie sich um nichts zu kümmern, wenn bspw. eine Funktion oder Methode eine Ausnahme wirft. Wird eine Ausnahme mit `throw` geworfen, befindet sich die Codeausführung noch im `try`-Block. Bevor der Fehler jetzt mit `catch()` behandelt wird, werden noch sämtliche Aufräumarbeiten auf dem Stack vom Laufzeitsystem durchgeführt. Kurz und einfach: Hierbei werden alle lokalen, nicht statischen Objekte und sonstigen »Speicherleichen« automatisch freigegeben. Dieses

Aufräumen funktioniert auch über mehrere Funktionsebenen. Der kontrollierte Abbau nach einer Ausnahme wird als *Stack-Unwinding* bezeichnet.

Erst wenn der Stack komplett abgebaut und sauber gemacht wurde, wird zum passenden `catch` gesprungen, welches die Ausnahme abgefangen hat. Am Ende der Fehlerbehandlung innerhalb der passenden `catch`-Anweisung, werden selbstverständlich auch alle dort verwendeten Speicherobjekte (u.a. auch das Ausnahme-Objekt) zerstört und freigegeben (erneut *Stack-Unwinding*).

10.2.7 Ausnahmen weiterleiten

Manchmal werden Sie im Programm mehrere `try-catch`-Ausnahmebehandlungen einrichten wollen. Hier kann es gegebenenfalls sinnvoll sein, wenn eine Ausnahme, welche in einer bestimmten Funktion ausgelöst wurde und von einer anderen Funktion abgefangen wird, an eine ganz andere Funktion (bspw. `main()`) weitergegeben werden soll, wo ebenfalls eine `try-catch`-Ausnahmebehandlung eingerichtet wurde. Solch ein Weiterleiten einer Ausnahme kann dann mithilfe eines leeren `throw` gemacht werden.

Das folgende Beispiel soll den Sachverhalt relativ einfach demonstrieren:

```
00 // kapitel10/ausnahmen06/main.cpp
...
01 void input_vals(int&, int&); // Voraus-Deklaration
02 enum class Err{DIV0, INP1, INP2};

03 double division(int& x, int& y) {
04     try {
05         input_vals(x, y);
06     }
07     catch(Err errcode) {
08         std::cerr << "Division abgebrochen\n";
09         throw; // Weiterleiten
10     }
11     if(y==0) {
12         throw Err::DIV0;
13     }
14     return static_cast<double>(x)/y;
15 }

16 void input_vals( int& x, int& y) {
17     std::cout << "Dividend: ";
18     if( ! (std::cin >> x)) {
19         throw Err::INP1;
20     }
```

```

21     std::cout << "Divisor: ";
22     if( ! (std::cin >> y)) {
23         throw Err::INP2;
24     }
25 }
...
26 try {
27     int val1=0, val2=0;
28     double res = division(val1, val2);
29     std::cout << "Ergebnis : " << res << std::endl;
30 }
31 catch(Err errcode) {
32     switch( errcode ) {
33         case Err::DIVO :
34             std::cerr << "Division durch 0\n"; break;
35         case Err::INP1 :
36             std::cerr << "Eingabefehler: Dividend\n"; break;
37         case Err::INP2 :
38             std::cerr << "Eingabefehler: Divisor\n"; break;
39     }
40 }
...

```

Hier wurde unser Beispiel etwas umgeschrieben. Im try-Block der main-Funktion rufen wir jetzt nur noch die Funktion `division()` in Zeile 28 auf, welche eine Ausnahme auslösen könnte. In der Funktion `division()` finden Sie jetzt in den Zeilen 04 bis 10 eine weitere try-catch-Ausnahmebehandlung. Im try-Block wird hier in Zeile 05 die Funktion `input_vals()` aufgerufen, um die Werte für unsere Division einzugeben. Hier haben wir uns für eine Ausnahme gerüstet, welche in der Funktion `input_vals()` in den Zeilen 19 bzw. 23 geworfen werden kann, wenn eine ungültige Zahl für den Dividend oder Divisor eingegeben wurde.

Wurde eine falsche Eingabe gemacht, wird diese Ausnahme in der catch-Anweisung zwischen den Zeilen 07 bis 10 abgefangen und behandelt. Hier geben wir lediglich die Meldung aus, dass die Division abgebrochen wurde, und leiten exakt die zuvor geworfene Ausnahme (`Err::INP1` oder `Err::INP2`) mit einem leeren `throw` (ohne zusätzlichen Parameter) in Zeile 09 an die Ausnahmebehandlung der main-Funktion in den Zeilen 26 bis 40 weiter, wo sich die catch-Anweisung in den Zeilen 31 bis 40 um den Fehler kümmert – hier eine genauere Angabe zum Fehler ausgibt.

10.2.8 try-catch verschachteln

try-catch-Ausnahmebehandlungen können auch ineinander verschachtelt werden. Dies kann bspw. nützlich sein, um einen Fehler erst in der inneren Ausnahmebe-

handlung vorzubehandeln und dann im äußeren try-catch nachzubehandeln. Ein einfacher Codeausschnitt einer verschachtelten try-catch-Ausnahmebehandlung:

```
00 // kapitel10/ausnahmen07/main.cpp
...
01 try {
02     int val1=0, val2=0;
03     try {
04         input_vals(val1, val2);
05     }
06     catch(Err errcode) {
07         std::cerr << "Fehler bei der Eingabe: ";
08         throw; // Weiterleiten
09     }
10     double res = division(val1, val2);
11     std::cout << "Ergebnis : " << res << std::endl;
12 }
13 catch(Err errcode) {
14     switch( errcode ) {
15         case Err::DIVO : std::cerr << "Division 0\n"; break;
16         case Err::INP1  : std::cerr << "Dividend\n"; break;
17         case Err::INP2  : std::cerr << "Divisor\n"; break;
18     }
19 }
...
```

In den Zeilen **03** bis **09** wurde im try-Block eine weitere try-catch-Ausnahmebehandlung verschachtelt, welche sich um die Eingabe der Werte in Zeile **04** kümmert. Wird hier eine Ausnahme geworfen, wird die catch-Anweisung in den Zeilen **06** bis **09** aktiv. Hier machen wir eine einfache Ausgabe und leiten den Fehlercode gleich an den äußeren try-Block weiter, wo diese Ausnahme dann in deren catch-Anweisung in den Zeilen **13** bis **19** aufgefangen wird.

Natürlich müssen Sie nicht zwangsläufig die Ausnahme mit einem leeren `throw` weiterleiten, sondern es können auch andere Ausnahmen nach außen geworfen werden, sofern dort dann ein `catch` mit dem entsprechenden Typ vorhanden ist. Theoretisch müssen Sie auch gar keine Ausnahme nach außen werfen. Allerdings hätte dies in diesem Beispiel zur Folge, dass Zeile **10** mit der Division ausgeführt wird und dort eben dann die Ausnahme geworfen wird, weil eine Division durch 0 gemacht würde.

10.2.9 Sinnvoller Einsatz von Ausnahmebehandlungen

Jetzt kennen Sie die Grundlagen, um eine Ausnahmebehandlung zu implementieren. In der Praxis werden Sie eher selten einfache Basisdatentypen als Ausnahme

werfen und behandeln. Hierfür verwendet man gewöhnlich Fehlerklassen, welche im nächsten Abschnitt auch gleich behandelt werden.

Bevor Sie jetzt anfangen, eigene Fehlerklassen zu schreiben, sollten noch ein paar Worte bezüglich der tatsächlichen Praxis von Ausnahmebehandlungen verloren werden. Sicherlich stellen Sie sich hier die Fragen, wo, wann oder wie häufig sollte man diese Ausnahmebehandlung verwenden? Diese Fragen sind nicht einfach und deutlich zu beantworten. Definitiv sollte man allerdings nicht hergehen und den kompletten Quellcode in ein riesiges `try-catch`-Konstrukt verpacken. Allerdings ist auch nicht unbedingt sehr übersichtlich, wenn Sie vor lauter `try-catch`-Konstrukturen kaum noch den tatsächlichen Code ausmachen können.

Sie sollten auf jeden Fall eine Ausnahmebehandlung an den Stellen einrichten, an denen Sie üblicherweise Fehler erwarten könnten. Häufig sind dies auch Stellen, wo der Anwender etwas eingeben kann. Hier sollten Sie nie darauf vertrauen, dass dieser schon das Richtige eingeben wird.

Abstand sollten Sie vom optionalen `catch(...)` halten, weil dieses alle Ausnahmen »verschluckt« und eventuell programminterne Fehler auch unterdrückt, was zu unerwarteten Ergebnissen bei der weiteren Ausführung des Programms führen kann. Es ist dann sehr schwierig solche programminternen Fehler zu lokalisieren.

10.3 Ausnahmeklasse implementieren

In der Praxis schmeißt und fängt man eher selten einfache Datentypen, sondern, wie es sich für eine OOP-Sprache gehört, eher ganze Klassen – auch Fehlerklassen genannt. Wie normale Basisdatentypen können Sie auch Klassen als Fehlertyp werfen – was Sie ja im Grunde mit `std::string` in den Beispielen zuvor schon des Öfteren gemacht haben. Eine solche (Fehler-)Klasse für Ausnahmen kann auch komplett leer sein. Hier ein solcher Codeausschnitt:

```
01 class ErrorClass {}; // Definition: Fehlerklasse

02 void funktion() {
03     if(...)
04         throw ErrorClass(); // Fehlerklasse werfen
...
05 }
...
06 try {
07     funktion(); // Könnte Fehlerklasse werfen
08 }
09 catch( ErrorClass e ) { // Fehlerklasse auffangen
10     std::cerr << "ErrorClass wurde geworfen" << std::endl;
11 }
```

In der Praxis sollte eine Fehlerklasse für Ausnahmen nicht einfach nur leer sein, sondern auch Eigenschaften und Methoden enthalten, um Informationen zum geworfenen Fehler zu liefern oder gar um Daten zu sichern, welche für die Behandlung des Fehlers von Bedeutung sind.

10.3.1 Ausnahmeklasse auslösen und abfangen

Wenn ein Objekt mit `throw` geworfen wird, dann wird auch ein temporäres Objekt vom Typ des `throw`-Objekts erzeugt. Ist der Typ eine Klasse, so wird auch der Konstruktor aufgerufen. Bevor das temporäre geworfene Objekt im `catch`-Block verwendet werden kann, wird zunächst das Stack-Unwinding durchgeführt. Erst nach dem Abbau des Stacks kann das abgefangene temporäre Objekt im `catch`-Block bearbeitet werden. Am Ende des `catch`-Blocks wird auch das temporäre Objekt zerstört (bei der Fehlerklasse der Destruktor aufgerufen).

Es soll auch hier wieder alles recht einfach gehalten werden, und wir verwenden hierzu, wie schon im kompletten Kapitel, nochmals das Listing mit der Division. Jetzt soll aber eine richtige Fehlerklasse als Ausnahme geworfen werden:

```
00 // kapitel10/ausnahmen08/main.cpp
...
01 class MyError {
02     std::vector<std::string> msg;
03     int err_code;
04 public:
05     enum class Err{DIV0=0, INP1, INP2};
06     MyError(const Err err ):
07         err_code(static_cast<int>(err)){
08         msg.push_back("Division durch 0");
09         msg.push_back("Eingabe Dividend");
10         msg.push_back("Eingabe Divisor");
11     }
12     const std::string& what() {
13         return msg.at(err_code);
14     }
15 };
...
```

Dies ist eine einfach gehaltene Fehlerklasse mit den Eigenschaften `std::vector<std::string>` für die Fehlermeldungen und einem `enum` für den Fehlercode. Die `enum`-Konstanten mit dem Fehlercode kennen Sie ja bereits, welche wir hier im `public`-Bereich in Zeile 05 platziert haben.

Der Konstruktor (in Zeilen 06 bis 10) erwartet beim Aufruf auch gleich einen Fehlercode (aus Zeile 05) der geworfenen Ausnahme. Da `enum class` in C++11 streng typisiert

ist, müssen wir hier ein `static_cast<>` verwenden, um aus der `enum`-Konstante ein Integer für den benötigten Index zu bekommen. Im Konstruktor wird dann ein Vektor mit Strings und den entsprechenden Fehlermeldungen in der passenden Reihenfolge der `enum`-Fehlercodes gefüllt.

Die Methode `what()` liefert dann die zum Fehlercode passende Fehlermeldung zurück. Wir verwenden hier auch gleich `at()` statt des Indexoperators, weil `at()` im Falle einer falschen Indexnummer auch gleich die Ausnahme `out_of_range` (siehe den Abschnitt »`out_of_range`« in Abschnitt 10.4.1) auslösen würde. Zwar ist es bei richtiger Verwendung kaum möglich, hier den Indexbereich falsch anzugeben (da strenges C++11-enum), aber es wäre immer noch eine Manipulation mit `static_cast<>` wie `MyError(static_cast<MyError::Err>(5))` möglich.

Zugegeben, der Code ist ziemlich einfach gehalten, aber dieser zeigt sehr schön eine komplette Fehlerklasse. Die Fehlerklasse können Sie jetzt wie folgt verwenden:

```
00 // kapitel10/ausnahmen08/main.cpp
...
01 double division(int& x, int& y) {
02     if(y==0) {
03         throw MyError(MyError::Err::DIV0);
04     }
05     return static_cast<double>(x)/y;
06 }

07 void input_vals( int& x, int& y) {
08     std::cout << "Dividend: ";
09     if( ! (std::cin >> x)) {
10         throw MyError(MyError::Err::INP1);
11     }
12     std::cout << "Divisor: ";
13     if( ! (std::cin >> y)) {
14         throw MyError(MyError::Err::INP2);
15     }
16 }
...
17 try {
18     int val1=0, val2=0;
19     input_vals(val1, val2); // kritischer Code
20     double res = division(val1, val2); // krit. Code
21     std::cout << "Ergebnis : " << res << std::endl;
22 }
23 catch(MyError err) {
24     std::cerr << "Fehler : " << err.what() << std::endl;
25 }
...
```


Im Listing hat sich nicht allzu viel verändert. Nur werden jetzt an den entsprechenden Stelle (Zeilen 03, 10 und 14) entsprechende Fehlerklassen (mit dem Fehlercode) geworfen und mit `catch` in den Zeilen 23 bis 25 abgefangen. In Zeile 24 bekommen Sie jetzt auch eine Meldung zum Fehler, wenn bei der Division oder der Eingabe etwas falsch gemacht wurde.

Natürlich können Sie die Klasse auch mit den Daten füllen, die den Fehler ausgelöst haben. Dies kann Ihnen helfen, die Ausnahme noch deutlicher zu lokalisieren. Hierzu nochmals unsere Fehlerklasse `MyError`:

```
00 // kapitel10/ausnahmen09/main.cpp
...
01 class MyError {
02     std::vector<std::string> msg;
03     int err_code;
04     int valX, valY;
05 public:
06     enum class Err{DIVO=0, INP1, INP2};
07     MyError( Err err, int val1=0, int val2=0 ):
08         err_code(static_cast<int>(err)),valX(val1),valY(val2){
09         msg.push_back("Division durch 0");
10         msg.push_back("Eingabe Dividend");
11         msg.push_back("Eingabe Divisor");
12     }
13     const std::string& what() {
14         std::cerr << valX << "/" << valY << std::endl;
15         return msg.at(err_code);
16     }
17 };

17 double division(int& x, int& y) {
18     if(y==0) {
19         throw MyError(MyError::Err::DIVO, x, y);
20     }
21     return static_cast<double>(x)/y;
22 }
```

Zugegeben, in diesem Beispiel ist es jetzt ziemlich trivial. Hier wurden bspw. noch die beiden eingegebenen Werte als Eigenschaft (Zeile 04) an die Klasse `MyError` mitgegeben. Bei einer Division durch 0 können diese fehlerhaften Werte dann bei einer Ausnahme an die Klasse `MyError` übergeben und geworfen werden, wie dies hier in Zeile 19 gemacht würde. Hiermit kann man dann die Fehlermeldung von `what()` noch genauer formulieren und genau ausgeben, mit welchen Werten die Ausnahme aufgetreten ist.

10.3.2 Ausnahmeklasse ableiten

Mit Fehlerklassen lassen sich durch Ableitungen auch ganze Fehlerklassenhierarchien erstellen, um bspw. Fehlerklassen genauer zu spezifizieren. So könnte bspw. die allgemeine Fehlerklasse `MyError` zur Basisklasse erklärt werden, und daraus könnten noch Fehlerklassen wie `MathError` (für Fehler bei Berechnungen) oder `InputError` (für Fehler bei der Eingabe) usw. gebildet werden, um gezielter und spezieller auf solche Fehler eingehen zu können:

```
class MyError {
    // Allgemeine Fehlerklasse
};

class MathError : public MyError {
    // Fehlerklasse für mathematische Fehler
};

class InputError : public MyError {
    // Fehlerklasse für die Eingabe
};
```

Fangen Sie jetzt mit `catch` einen Typ der Basisklasse als Ausnahme ab (hier bspw. `catch(MyError &e)`), so werden **alle** Fehlerobjekte abgefangen, also auch die geworfenen Fehlerobjekte der abgeleiteten Klassen (wie im Beispiel `MathError` und `InputError`) und auch der Basisklasse (im Beispiel `MyError`).

Richten Sie hingegen ein `catch()` für eine abgeleitete Klasse ein (bspw. `catch(MathError &e)`), dann werden nur noch Fehlerobjekte von diesem abgeleiteten Typ (im Beispiel `MathError`) abgefangen und behandelt. In der Praxis wird das Objekt zwischen `throw` und `catch` kopiert, und man arbeitet daher nicht auf dem Original. Aber hier müssen Referenzen mit `&` verwendet werden, denn erst diese erlauben es, die Polymorphie auf den Fehlerklassen zu erhalten. Ohne das `&` würde hier nichts gehen.

10.4 Standardausnahmen von C++

In der Standardbibliothek von C++ sind ebenfalls verschiedene Fehlerklassen definiert, welche von verschiedenen Bibliotheksfunktionen oder Containerklassen geworfen werden können. So wirft bspw. `std::vector` eine Ausnahme `out_of_range`, wenn mit `at()` auf einen unerlaubten Bereich zugegriffen wurde. Die Fehlerklassen der Standardbibliothek lassen sich aber auch für benutzerdefinierte Zwecke einsetzen und werfen.

Die Standardfehlerklassen wurden in einer Klassenhierarchie implementiert, in der die Klasse `exception` die Basisklasse darstellt. In `exception` steht neben dem Standard- und Kopierkonstruktor sowie einer Zuweisung noch eine virtuelle Methode `what()` zur Verfügung, welche in einer Fehlermeldung die Ursache der geworfenen Ausnahme als C-String (!) zurückgibt. Die Definition ist in der Headerdatei `<stdexcept>` enthalten

Ausgehend von der Basisklasse `exception` werden die Standardfehlerklassen in die Klassen `logic_error` und `runtime_error` unterteilt. Des Weiteren finden sich noch verteilt auf verschiedene Bibliotheken direkt von `exception` abgeleitete Fehlerklassen (siehe Abbildung 10.1).

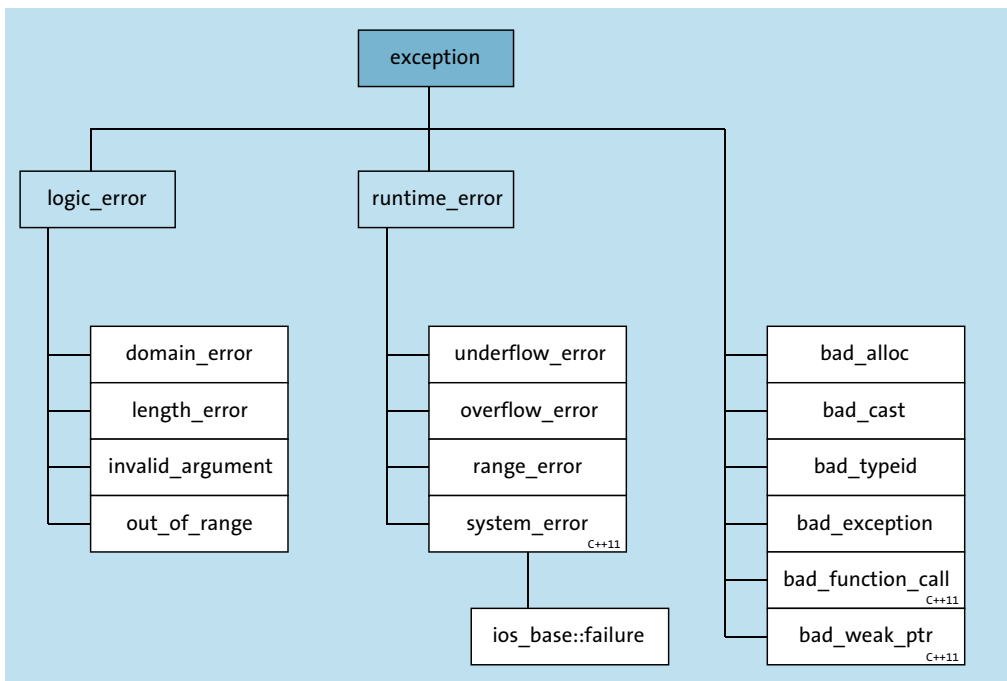


Abbildung 10.1 Die Klassenhierarchie der Standardfehlerklassen von C++

Fehlerklasse `regex_error`

Nicht aufgelistet wurde hier die Standardfehlerklasse `regex_error` (aus der Headerdatei `<regex>`), welche ebenfalls zu `runtime_error` gehört und geworfen wird, wenn es zu einem Fehler mit regulären Ausdrücken kommt. Die regulären Ausdrücke (neu im C++11-Standard dabei) werden noch gesondert in Kapitel 14, »Reguläre Ausdrücke (C++11)«, behandelt.

10.4.1 Logische Fehler (logic_error)

Logische Fehler (logic_error) sind vermeidbare Fehler mit logischen Ursachen, die Sie eigentlich schon vor dem Programmstart oder durch diverse Überprüfungen (von Funktionen, Argumenten usw.) finden könnten (Tabelle 10.1).

Fehlerklasse	Header	Bedeutung
domain_error	<stdexcept>	Wertebereichsfehler
length_error	<stdexcept>	maximale Größe überschritten
invalid_argument	<stdexcept>	unzulässiges Argument
out_of_range	<stdexcept>	ungültige Position

Tabelle 10.1 Logische Standardfehler, welche von der Standardklasse »logic_error« abgeleitet wurden

domain_error

Dieser Fehler definiert einen Typ, der geworfen wird, wenn bspw. bei einer mathematischen Funktion ein Argument oder Wert verwendet wurde, für den die Funktion nicht definiert wurde. Dies könnte bspw. der Fall sein, wenn der Standardfunktion `std::sqrt()` aus der Headerdatei `<cmath>` ein negativer Wert übergeben wird, was nicht zulässig ist (oder sein sollte). Es sollte allerdings erwähnt werden, dass diese Ausnahme in der Standardbibliothek von keiner Funktion geworfen wird. Diese Standardausnahme muss als Ausnahme im Programm geworfen werden. Ein einfaches Beispiel hierzu:

```
00 // kapitel10/ausnahmen10/domain_error.cpp
01 #include <cmath>
...
02 double safe_sqrt( double val ) {
03     if( val < 0.0 )
04         throw
            std::domain_error("Negative Zahl nicht erlaubt");
05     return std::sqrt(val);
06 }
...
07 try {
08     double val = safe_sqrt(-1);
09 }
10 catch( std::domain_error e ) {
11     std::cerr << e.what() << std::endl;
12 }
...
```

Hier wird das Ziehen einer Quadratwurzel als Ausnahmebehandlung in den try-Block gestellt. In der Funktion `safe_sqrt()` wird überprüft, ob das übergebene Argument nicht ein negativer Wert ist. Im Falle eines negativen Wertes wird die Standardausnahme `domain_error` in Zeile **04** geworfen und mit `catch` in Zeile **10** abgefangen.

length_error

Einige Komponenten der Standardbibliothek werfen diese Ausnahme, wenn ein Objekt erzeugt wurde, dass größer ist, als maximal erlaubt. Dies ist bspw. der Fall, wenn eine Klasse wie `std::vector` oder `std::string` mit `max_size()+1` überfüllt wurde. Im folgenden Beispiel lösen wir diese Ausnahme mit Absicht aus:

```
00 // kapitel10/ausnahmen11/length_error.cpp
...
01 try {
02     std::string text;
...
03     text.resize(text.max_size()+1);
04 }
05 catch( std::length_error e ) {
06     std::cerr << "Fehler: " << e.what() << std::endl;
07 }
```

Durch die Größenänderung in Zeile **03** mit `max_size()+1` provozieren wir mit Absicht, dass hier die Ausnahme `length_error` von `std::string` geworfen wird. Mit `catch` wurde hier auch gleich der passende Typ in Zeile **05** eingerichtet, um `length_error` abzufangen.

invalid_argument

Diese Ausnahme wird ebenfalls von einigen Komponenten der Standardbibliothek (bspw. `std::bitset`, `std::deque`, `std::string`, `std::vector` usw.) geworfen, wenn Sie ein falsches Argument verwendet haben. Auch hierzu ein einfaches Beispiel, in dem wir diese Ausnahme absichtlich provozieren:

```
00 // kapitel10/ausnahmen12/invalid_argument.cpp
01 #include <bitset>
...
02 try {
03     std::bitset<8> bitset(std::string("01011X10"));
04 }
05 catch (std::invalid_argument e) {
06     std::cerr << "Fehler : " << e.what() << std::endl;
07 }
...
```

Hier wurde in Zeile **03** ein `std::bitset` (siehe Abschnitt 12.2.3, »Container für Bit-Manipulationen – bitset«) aus einem String erstellt. Ein Bitset kennt allerdings nur 1 und 0, und das Zeichen `X` ist daher falsch, weshalb hier `invalid_argument` geworfen wird, das wir im eingerichteten `catch` in Zeile **05** auch gleich wieder auffangen.

out_of_range

Diese Ausnahme wird geworfen, wenn sich ein Wert nicht mehr in einer erlaubten Speicheradresse befindet. Einige Teile der Standardbibliotheken (wie bspw. `std::vector`, `std::string`, `std::deque` usw.) werfen diese Ausnahme, wenn mit der Methode `at()` auf einen ungültigen Bereich zugegriffen wird. Ein einfaches Beispiel hierzu:

```
00 // kapitel10/ausnahmen13/out_of_range.cpp
...
01 try {
02     std::string text("Ein einfacher Text");
...
03     text.at(30) = 'X';
04 }
05 catch (std::out_of_range e) {
06     std::cerr << "Fehler : " << e.what() << std::endl;
07 }
...
```

In diesem Beispiel wird in Zeile **03** die Ausnahme `out_of_range` geworfen, weil hier versucht wurde, der nicht definierten Position 30 den Buchstaben `X` zuzuweisen. Mit `catch` in Zeile **05** wird diese Ausnahme dann wieder abgefangen.

10.4.2 Laufzeitfehler (runtime_error)

Bei Laufzeitfehlern handelt es sich um eine Basisklasse von Ausnahmen, welche während der Kontrolle des Programms jederzeit auftreten und nicht verhindert werden können, oder – wie es der Name schon sagt – sind es Fehler, die nur zur Laufzeit ermittelt werden können (Tabelle 10.2).

Fehlerklasse	Header	Bedeutung
<code>underflow_error</code>	<code><stdexcept></code>	arithmetischer Unterlauf
<code>overflow_error</code>	<code><stdexcept></code>	arithmetischer Überlauf

Tabelle 10.2 Nicht vorhersehbare Fehler, welche zur Laufzeit des Programms auftreten können und von der Fehlerklasse »runtime_error« abgeleitet wurden

Fehlerklasse	Header	Bedeutung
<code>range_error</code>	<code><stdexcept></code>	Bereichsüberschreitung
<code>system_error(C++11)</code>	<code><system_error></code>	Fehlermeldung vom System
<code>ios_base::failure</code>	<code><system_error></code>	Fehler bei Ein-/Ausgabe

Tabelle 10.2 Nicht vorhersehbare Fehler, welche zur Laufzeit des Programms auftreten können und von der Fehlerklasse »runtime_error« abgeleitet wurden (Forts.)

underflow_error und overflow_error

Diese Ausnahmen sollten geworfen werden, wenn Integer nicht richtig dargestellt werden, weil der negative Wert zu klein bzw. der positive Wert zu groß ist. Für Gleitkommazahlen wird empfohlen, die Ausnahme `range_error` zu verwenden. In der Standardbibliothek wird kaum Gebrauch von `underflow_error` (ist mir nichts bekannt) oder `overflow_error` (`std::bitset` nutzt diese Ausnahme) gemacht. Ein Beispiel zu `overflow_error` mit `std::bitset` fällt mir hierzu ein:

```
00 // kapitel10/ausnahmen14/overflow_error.cpp
...
01 try {
02     std::bitset<80> set01;
03     set01[0] = 1;
04     set01[79] = 1;
05     unsigned long value = set01.to_ulong();
06 }

07 catch (std::overflow_error e) {
08     std::cerr << "Fehler : " << e.what() << std::endl;
09 }
...
```

Hier wird in Zeile **05** versucht, ein Bitset mit 80 einzelnen Bits in ein `unsigned long` zu konvertieren, was ja abhängig vom System 32- oder 64-Bit aufnehmen kann. Daher wird hier ein `overflow_error` geworfen und in den Zeilen **07** bis **09** abgefangen.

range_error

Diese Ausnahme sollte wie `underflow_error` und `overflow_error` bei zu großen oder zu kleinen Gleitkommazahlen verwendet werden, wenn diese nicht mehr richtig darstellbar sind. In der Standardbibliothek selbst wirft keine Komponente diese Ausnahme. Für Integer sollten sie `underflow_error` und `overflow_error` verwenden.

system_error (C++11)

Diese Ausnahme wird von einigen Funktionen der Standardbibliothek geworfen, welche auf eine sehr niedrige Schnittstelle des Betriebssystems zurückgreifen. Dies ist bspw. bei den Threads mit `std::thread` der Fall, welche neu mit C++11 eingeführt wurden. Gibt es hier bspw. Probleme bei der Erzeugung von neuen Threads mit dem Konstruktor, wird diese Ausnahme in Verbindung mit einem Objekt `std::error_code` geworfen. Hier enthält die Klasse `system_error` eine Methode `code()`, um dabei über weitere Methoden auf das Objekt `std::error_code` zurückgreifen zu können. Die Threads werden gesondert in Kapitel 15, »Multithreading (C++11)«, behandelt.

std::error_code (C++11)

Die Klasse `std::error_code` enthält einen Fehlercode einer bestimmten Kategorie. Das Betriebssystem und Bibliotheken auf niedrigerer Ebene generieren gewöhnlich auch numerische Codes, die das Ergebnis enthalten. Natürlich sind diese Informationen von Plattform zu Plattform unterschiedlich und daher nicht portabel. Die Klasse `std::error_code` enthält weitere Methoden, um noch mehr Informationen aus einem numerischen Fehlercode zu erhalten.

Das folgende Beispiel demonstriert die Ausnahme `system_error` und wie Sie Informationen von `std::error_code` auswerten können:

```
00 // kapitel10/ausnahmen15/system_error.cpp
...
01 #include <thread>
02 #include <system_error>
...
03 try {
04     std::thread().detach();
05 }
06 catch(std::system_error e) {
07     std::cout << "Fehler   : " << e.what() << '\n';
08     std::cout << "Code    : " << e.code().value() << '\n';
09     std::cout << "Kategorie: "
                << e.code().category().name() << '\n';
10     std::cout << "Meldung : " << e.code().message() << '\n';
11 }
...
```

Das Loslösen eines Nicht-Threads wie in Zeile 04 geht nicht und wirft sofort eine `system_error`-Ausnahme. Mit `catch` fangen wir die Ausnahme ab. Neben der bekannten Methode `what()` in Zeile 07 finden Sie hier noch weitere Informationen dank der neuen Klasse `std::error_code` mit `code().value()` (Fehlernummer), `code().cate-`

gory().name() (Fehlerkategorie) und code().category().message() (Fehlernachricht). Allerdings sind diese Informationen vom Betriebssystem abhängig und daher häufig auch ziemlich unterschiedlich.

Benutzerdefinierter std::error_code

An dieser Stelle soll auch noch darauf hingewiesen werden, dass es durchaus möglich ist, einen eigenen Fehlercode für std::error_code zu schreiben. Das Prinzip von <system_error> kann jederzeit um benutzerdefinierte Fehlercodes erweitert werden. In diesem Buch wird hierauf zwar nicht eingegangen, aber eine sehr gute Einführung dazu finden Sie auf der Webseite <http://blog.think-async.com/2010/04/system-error-support-in-c0x-part-4.html>.

ios_base::failure

Die Fehlerklasse ios_base::failure wurde von der Klasse system_error abgeleitet und wird für das Werfen von Ausnahmen bei Ein- und Ausgabe (auch der Datei-Ein-/Ausgabe) verwendet. Da diese Klasse auch von system_error abgeleitet wurde, finden Sie hier ebenfalls wieder die Methode code() mit an Bord, um weitere Informationen dank der Klasse std::error_code zu erhalten. Leider scheint die neue abgeleitete Form von ios_base::failure noch nicht überall implementiert zu sein. Stream-Klassen für die Ein- und Ausgaben werden in Kapitel 13, »Die Stream-Ein-/Ausgabeklassen von C++«, behandelt.

Hierzu ein einfaches Beispiel:

```
00 // kapitel10/ausnahmen16/ios_base_error.cpp
01 #include <system_error>
...
02 int val=0;
03 std::cin.exceptions(std::ios::failbit);
04 try {
05     std::cout << "Integer eingeben: ";
06     std::cin >> val;
07 }
08 catch(std::ios_base::failure e) {
09     std::cout << "Fehler   : " << e.what() << '\n';
10 }
...
```

In Zeile 03 müssen Sie die die Ausnahme erst mit der Methode exceptions() für std::cin aktivieren. Neben ios::failbit können Sie hier noch andere sogenannte Flags für die Stream-Ein-/Ausgabe verwenden. Mehr dazu allerdings erst in Kapitel 13.

Wenn Sie jetzt in Zeile **06** etwas eingeben, was nicht den Regeln für den Typ eines Integers entspricht, wird die Ausnahme geworfen und in den Zeilen **08** bis **10** abgefangen.

10.4.3 Weitere Standardfehlerklassen

Es gibt noch weitere, direkt von `exception` als Basisklasse abgeleitete Fehlerklassen, welche auf die verschiedenen Teile der Standardbibliothek verteilt sind und hier noch kurz erwähnt werden sollten (Tabelle 10.3).

Fehlerklasse	Header	Bedeutung
<code>bad_alloc</code>	<code><new></code>	Speicherzuweisungsfehler
<code>bad_cast</code>	<code><typeinfo></code>	Typumwandlungsfehler
<code>bad_typeid</code>	<code><typeinfo></code>	falscher Objekttyp
<code>bad_exception</code>	<code><exception></code>	Fehler bei Ausnahmebehandlung
<code>bad_function_call</code>	<code><functional></code>	Fehler bei Funktionsobjekt-Aufruf
<code>bad_weak_ptr</code>	<code><memory></code>	ungültiger <code>weak_ptr</code>

Tabelle 10.3 Weitere Fehlerklassen, die in der Standardbibliothek verstreut und direkt von der Basisklasse »`exception`« abgeleitet sind

`bad_alloc` (und `bad_array_new_length` (C++11))

Die Ausnahme `bad_alloc` wird geworfen, wenn bspw. ein mit `new` oder `new[]` angeforderter Speicher nicht reserviert werden konnte. Ein einfaches Beispiel, wie Sie hierfür eine Ausnahmebehandlung einrichten können:

```
00 // kapitel10/ausnahmen17/bad_alloc.cpp
01 #include <new>
...
02 int *iptr=nullptr;
...
03 try {
04     iptr = new int[10000000];
...
05 }
06 catch(std::bad_alloc e) {
07     std::cout << "Fehler   : " << e.what() << '\n';
08 }
...
```

Wenn in Zeile **04** im try-Block nicht mehr genügend zusammenhängender Speicher für `iptr` reserviert werden könnte, würde hier die Ausnahme `bad_alloc` geworfen und von `catch` in den Zeilen **06** bis **08** abgefangen und behandelt werden.

Neu in C++11 wurde außerdem noch `bad_array_new_length` von der Klasse `bad_alloc` abgeleitet und eingeführt. Diese Ausnahme wird geworfen, wenn

- ▶ die Array-Größe kleiner als 0 ist (bspw. `iptr = new int[-1]`),
- ▶ die Array-Größe größer ist, als die Implementierung erlaubt, oder
- ▶ die Anzahl der Elemente in der Initialisierungsliste größer ist, als die Anzahl der Elemente, die initialisiert werden können.

bad_cast

Diese Ausnahme wird geworfen, wenn eine `dynamic_cast`-Umwandlung zur Laufzeit fehlschlägt. Dies ist bspw. der Fall, wenn versucht wird, aus einem Objekt ein nicht vollständiges Zielobjekt zu machen. Einige Funktionen der Standardbibliothek werfen diese Ausnahme ebenfalls. Hierzu ein kurzes Beispiel:

```
00 // kapitel10/ausnahmen18/bad_cast.cpp
01 #include <typeinfo>

02 class Window { virtual void draw(){} };
03 class Button { virtual void draw(){} };
...
04 Window win01;
05 try {
06     Button& but01 = dynamic_cast<Button&>(win01);
...
07 }
08 catch(std::bad_cast e) {
09     std::cout << "Fehler   : " << e.what() << '\n';
10 }
```

Beim Versuch in Zeile **03**, aus einem `Window`-Objekt eine `Button`-Referenz zu machen, wird definitiv die Ausnahme `bad_cast` geworfen und mit `catch` in den Zeilen **08** bis **10** abgefangen und behandelt.

bad_typeid

Diese Ausnahme wird geworfen, wenn der `typeid`-Operator einen falschen bzw. ungültigen Objekttyp erhält. Das Beispiel hierzu:

```
00 // kapitel10/ausnahmen19/bad_typeid.cpp
01 #include <typeinfo>
...
```

```

02 class Window { virtual void draw(){} };
...
03 Window *winPtr=nullptr;
04 try {
05     std::cout << typeid(*winPtr).name() << '\n';
06 }
07 catch(std::bad_typeid e) {
08     std::cout << "Fehler   : " << e.what() << '\n';
09 }
...

```

Wegen des Versuchs, den Namen für den Zeiger eines polymorphen Typs in Zeile 05 auszugeben, wird hier die Ausnahme `bad_typeid` geworfen und von `catch` in den Zeilen 07 bis 09 abgefangen und behandelt.

bad_exception

Dieser spezielle Typ wurde erstellt, um als dynamische Ausnahme-Spezifikation in einer Funktion aufgelistet und geworfen zu werden. Allerdings ist die dynamische Ausnahme-Spezifikation mit C++11 zu »deprecated« (veraltet) erklärt worden und sollte somit nicht mehr verwendet werden.

bad_function_call (C++11)

Diese Ausnahme wird geworfen, wenn ein leeres Funktionsobjekt (siehe Abschnitt 12.5.5, »Funktionsobjekte«) verwendet wird. Die Klasse ist direkt von der Basisklasse `exception` abgeleitet und wurde mit dem C++11-Standard eingeführt. Folgendes Beispiel soll diese Ausnahme demonstrieren:

```

00 // kapitel10/ausnahmen20/bad_function_call.cpp
01 #include <functional>
...
02 std::function<int(double, double)> dval01 =
    std::divides<double>();
03 std::function<int(double, double)> dval02;
...
04 try {
05     std::cout << dval01(100, 3.1415) << std::endl;
06     std::cout << dval02(232, 10) << std::endl;
07 }
08 catch(std::bad_function_call e) {
09     std::cout << "Fehler   : " << e.what() << '\n';
10 }
...

```

In Zeile 02 sehen Sie die korrekte Verwendung eines vordefinierten Funktionsobjekts der Standardbibliothek. In Zeile 03 hingegen wurde zwar schon mal der Typ für das Funktionsobjekt angegeben, aber das Funktionsobjekt selbst fehlt hier noch – und die spätere Zuweisung wurde hier vergessen. Daher wirft der Aufruf in Zeile 06 die Ausnahme `bad_function_call`, welche dann in den Zeilen 08 bis 10 abgefangen und behandelt werden kann.

bad_weak_ptr (C++11)

Diese Ausnahme wird von den neu eingeführten Smart Pointern in C++11 von einem `shared_ptr`-Konstruktor geworfen, wenn ein Smart Pointer mit einem abgelaufenen `weak_ptr` aufgerufen wird. Die neuen Smart Pointer werden in Abschnitt 16.3, »Smart Pointer (C++11)«, behandelt.

10.4.4 Fehlermeldung mit `what()`

Bei allen Ausnahmen, die direkt oder indirekt von der Basisklasse `exception` abgeleitet wurden, können Sie sich im `catch`-Block mithilfe der virtuellen Methode `what()` eine Fehlermeldung der Ursache bzw. Ausnahme ausgeben lassen. Die Methode gibt gewöhnlich einen C-String zurück. Des Weiteren wird auch in allen von `exception` abgeleiteten Klassen ein Konstruktor mit einem solchen String als Parameter definiert, womit Sie beim Werfen einer solchen Ausnahme auch gleich einen benutzerdefinierten String mit angeben können, der anschließend im `catch`-Block von `what()` ausgegeben werden kann. Ein Beispiel, wie das mit dem benutzerdefinierten String funktioniert, haben Sie bereits in Listing *invalid_argument.cpp* mit der Ausnahme `out_of_range` gesehen.

<system_error> (C++11)

Die neu in C++11 eingeführte Headerdatei enthält neben `system_error` weitere Klassen wie `error_category`, `generic_category`, `system_category`, `error_condition`, `errc` und `error_code`, womit verschiedene Typen und Funktionen implementiert sind, welche verschiedene Informationen über Fehler- und Ausnahmezustände von Betriebssystemfunktionen und anderen APIs einer niedrigeren Ebene ausgeben können. Die Verwendung von `system_error` wurde bereits ansatzweise in Listing *system_error.cpp* demonstriert.

10.5 Spezielle Fehlerbehandlungsfunktionen

Keine Fehlerausnahmebehandlung ist perfekt, und auch die Fehlerbehandlung kann mal einen Fehler machen. In diesem Abschnitt lernen Sie noch ein paar Funktionen

und Kniffe kennen, wie Sie noch gezielter auf die eine oder andere Situation im Zusammenhang mit Ausnahmen reagieren können.

10.5.1 `terminate()` behandeln

Sie wissen mittlerweile, dass mit `std::terminate()` das Programm beendet wird, wenn kein passender `catch`-Block zum Abfangen einer Ausnahme vorhanden ist. Theoretisch könnten Sie `std::terminate()` auch manuell aufrufen (obgleich das wenig Sinn macht).

Hier wird ebenfalls `std::terminate` aufgerufen

Die Funktion `std::terminate()` wird auch aufgerufen, wenn bei der Ausführung eines Destruktors während des Stack-Unwindings eine Ausnahme geworfen wird oder ein statisches und nichtlokales Objekt während der Erzeugung oder der Zerstörung eine Ausnahme wirft.

Ebenfalls eine Ausnahme wird geworfen, wenn ein vorgegebener `unexpected_handler` aufgerufen wird. Dies ist der Fall, wenn `unexpected()` aufgerufen wurde, weil eine Funktion nicht das wirft, was in der Ausnahme-Spezifikation aufgeführt wurde. Allerdings sind Ausnahme-Spezifikationen seit C++11 zu **deprecated** (veraltet) erklärt worden und sollten künftig nicht mehr verwendet werden.

Ohne jetzt weitere Vorkehrungen zu treffen, würde `std::terminate()` standardmäßig die Funktion `abort()` zum Beenden aufrufen. Hier könnten Sie aber noch eingreifen und mit der Funktion `std::set_terminate()` eine Funktion einrichten, welche statt der Funktion `abort()` aufgerufen wird.

`std::terminate_handler`

Um hier noch etwas genauer zu sein: Beschrieben wird hier der Funktionszeiger `std::terminate_handler`, welcher eine Funktion installiert hat (oder darauf verweist), die mit `std::terminate()` aufgerufen wird. In der C++-Implementierung ist dies standardmäßig die Funktion `abort()`. Mit `std::set_terminate()` können Sie hier eine eigene Funktion installieren, welche stattdessen aufgerufen wird. Die Funktion muss hierbei `void` als Rückgabewert haben und darf auch keinen Parameter haben – also muss sie exakt dieselbe Syntax wie `void abort();` aufweisen.

Hierzu nun ein einfaches Beispiel, welches demonstrieren soll, wie Sie mit `std::set_terminate()` eine benutzerdefinierte Funktion einrichten können, die aufgerufen wird, wenn `std::terminate()` ausgeführt wird:

```
00 // kapitel10/ausnahmen21/set_terminate.cpp
01 #include <stdexcept>
```

```
02 #include <exception>
...
03 double division(int x, int y) {
04     if(y==0) {
05         throw std::string("Division durch 0");
06     }
07     return static_cast<double>(x)/y;
08 }

09 void before_terminate() {
10     std::cout << "Fehler bei den Ausnahmen\n";
11     // Tue was Wichtiges
12 }
...
13 std::set_terminate( before_terminate );
14 std::cout << division(100, 0) << std::endl;
...
```

In Zeile 13 wird die Funktion `before_terminate()` aus den Zeilen 09 bis 12 mit `std::set_terminate()` als neue Funktion eingerichtet, die aufgerufen wird, sobald `std::terminate()` ausgeführt wird. In Zeile 14 verwenden wir unsere bekannte Funktion `division()`, welche in den Zeilen 03 bis 08 definiert wurde und in diesem Fall in Zeile 05 eine Ausnahme vom Typ `std::string` werfen wird. Auf diese Ausnahme waren wir hier gar nicht gefasst und haben nicht mal einen `try-catch`-Block implementiert. Daher wird hier `std::terminate()` ausgeführt. Anstelle von `abort()` wird jetzt unsere neu eingerichtete Funktion `before_terminate()` aufgerufen und ausgeführt. Tabelle 10.4 zeigt Ihnen die Funktionen, die bei einem Fehler in der Ausnahmebehandlung verwendet werden können.

Funktion	Beschreibung
<code>terminate()</code>	Funktion wird aufgerufen, wenn die Ausnahmebehandlung fehlschlägt.
<code>terminate_handler</code>	Funktionszeiger auf eine Funktion, welche von <code>std::terminate()</code> aufgerufen wird. Die C++-Implementierung ruft <code>abort()</code> auf.
<code>get_terminate()</code> (C++11)	Gibt den aktuell installierten <code>terminate_handler</code> zurück.
<code>set_terminate()</code>	Ändert die Funktion, welche von <code>std::terminate()</code> aufgerufen wird.

Tabelle 10.4 Funktionen aus der Headerdatei `<exception>`, die bei einem Fehler in der Ausnahmebehandlung verwendet werden können

10.5.2 `std::uncaught_exception()`

Diese spezielle Funktion gibt nur dann `true` zurück, wenn das Säubern auf dem Stack (*Stack-Unwinding*) im Augenblick stattfindet. Genauer bedeutet dieser Zustand, dass eine Ausnahme geworfen wurde, aber diese noch nicht in einem `catch` behandelt und dass auch noch nicht `std::terminate()` oder `std::unexpected()` aufgerufen wurde.

10.5.3 Ausnahme-Objekte zwischenspeichern (C++11)

Neu in C++11 ist die Möglichkeit hinzugekommen, `exception`-Objekte, die geworfen und aufgefangen wurden, mit einem speziellen Zeiger (`std::exception_ptr`) zu verwalten. Abfangen und dem Zeiger `std::exception_ptr` zuweisen können Sie ein solches `exception`-Objekt mit `std::current_exception()`. Instanzen von `std::exception_ptr`, welche das `exception`-Objekt verwaltet, können Sie in einer anderen Funktion oder einem anderen Thread erneut mit `std::rethrow_exception()` werfen und dort mit `catch()` abfangen und behandeln. `std::exception_ptr` ist ein `nullptr`, wenn es auf kein `exception`-Objekt verweist.

Hierzu ein einfaches Beispiel, welches zeigen soll, wie Sie ein geworfenes `exception`-Objekt abfangen und mit `std::exception_ptr` verwalten können:

```
00 // kapitel10/ausnahmen22/exception_ptr01.cpp
01 #include <stdexcept>
02 #include <exception>
...
03 void do_it_yourself(std::exception_ptr excpPtr) {
04     try {
05         if( excpPtr != nullptr) {
06             std::rethrow_exception(excpPtr);
07         }
08     }
09     catch(std::exception e) {
10         std::cout << e.what() << std::endl;
11     }
12 }
...
13 std::exception_ptr excpPtr;
14 try {
15     std::string text("Ein einfacher Text");
16     text.at(30) = 'X';
17 }
18 catch(...) {
19     excpPtr = std::current_exception();
```



```

20 }
21 if( excpPtr != nullptr) {
22     do_it_yourself(excpPtr);
23 }
...

```

In Zeile 13 wird der Zeiger `std::exception_ptr` angelegt. Im try-Block in den Zeilen 14 bis 17 lösen wir in Zeile 16 absichtlich eine Ausnahme aus, indem wir außerhalb eines gültigen Bereichs mit `at()` zugreifen. Unser alternatives `catch()` aus den Zeilen 18 bis 20 fängt diese Ausnahme ab, weil sonst kein `catch()` dafür eingerichtet wurde. In Zeile 19 weisen wir dem `std::exception_ptr` die geworfene Ausnahme (ein `exception`-Objekt) mit `std::current_exception()` zu.

Außerhalb des try-catch-Blocks in Zeile 21 prüfen wir zunächst, ob unser Zeiger `excpPtr` vom Typ `std::exception_ptr` auch wirklich auf ein `exception`-Objekt verweist und nicht `nullptr` ist. Ist dies der Fall, übergeben wird diesen Zeiger an die Funktion `do_it_yourself()` in Zeile 22.

In der Funktion (die ja durchaus auch ein Thread sein kann, siehe Kapitel 15, »Multithreading (C++11)«) finden Sie jetzt einen try-catch-Block, der sich um die Ausnahme kümmert, die unser `std::exception_ptr` enthält. Im try-Block prüfen wir zur Sicherheit nochmals in Zeile 05, ob der Zeiger nicht `nullptr` ist und dass er tatsächlich auf ein `exception`-Objekt verweist. Ist dies der Fall, werfen wir die Ausnahme mit `std::rethrow_exception()` erneut aus und fangen diese im anschließenden catch-Block in den Zeilen 09 bis 11 ab und behandeln diese dort, hier wieder nur mit einer einfachen Ausgabe der Ausnahme.

Natürlich können Sie `std::exception_ptr` auch für eigene Klassen verwenden. Hierfür müssen Sie lediglich Ihre Fehlerklasse von der `exception`-Klasse ableiten und die virtuelle Methode `what()` implementieren. Hierzu ein Minimalbeispiel:

```

00 // kapitel10/ausnahmen23/exception_ptr02.cpp
...
01 class MyError : public std::exception {
02     public:
03         const char* what() const noexcept {
04             return("Ausnahme bei MyError");
05         }
06 };

07 void do_it_yourself(std::exception_ptr excpPtr) {
08     try {
09         if( excpPtr != nullptr) {
10             std::rethrow_exception(excpPtr);

```

```

11     }
12 }
13 catch(std::exception e) {
14     std::cout << e.what() << std::endl;
15 }
16 }
...
17 std::exception_ptr excpPtr;
18 try {
19     throw MyError();
20 }
21 catch(...) {
22     excpPtr = std::current_exception();
23 }
24 if( excpPtr != nullptr) {
25     do_it_yourself(excpPtr);
26 }

```

Durch die öffentliche Ableitung von der Basisausnahmeklasse `exception` in Zeile 01 und der Definition der virtuellen Methode `what()` in den Zeilen 03 bis 05 innerhalb unserer eigenen neuen Fehlerklasse `MyError` können wir dank der Vererbung (und somit Verwandtschaft mit `exception`) jetzt auch `MyError` werfen (siehe Zeile 19) und mit `std::exception_ptr` verwalten.

10.5.4 noexcept (C++11)

Neu in C++11 wurde das Schlüsselwort `noexcept` eingeführt. Da die Ausnahme-Spezifikation zu deprecated (veraltet) erklärt wurde, wurde ein Ersatz für ein leeres `throw()` benötigt. `noexcept` ist somit die (verbesserte) Alternative für ein leeres `throw()` (falls Sie bereits C++-Kenntnisse haben). Wenn Sie jetzt bspw. eine Funktion/Methode mit `noexcept` belegen, so bedeutet dies,

- ▶ dass die Funktion/die Methode/der Konstruktor/der Destruktor keine Ausnahme werfen darf und
- ▶ dass das Programm nicht auf die Ausnahme reagiert, wenn trotzdem eine geworfen wird. In diesem Fall wird dann `std::terminate()` aufgerufen.

Das Schlüsselwort `noexcept` wird bereits zur Übersetzungszeit ausgewertet. Daher stehen dem Compiler viele Möglichkeiten für Optimierungen zur Verfügung. Wenn Sie sich vielleicht ein wenig die neuen C++11-Schnittstellen der Standardbibliothek angesehen haben, werden Sie festgestellt haben, dass auch die Standardbibliothek regen Gebrauch von `noexcept` macht, um die Performance zu steigern. Zum Beispiel verschieben die Standardcontainer wie `std::string`, `std::vector` usw. ihre Elemente,

wenn der Move-Konstruktor mit `noexcept` gekennzeichnet ist. Ansonsten wird nur kopiert.

Hierzu ein einfaches Beispiel, wie Sie `noexcept` verwenden können:

```
00 // kapitel10/ausnahmen24/noexcept01.cpp
01 #include <stdexcept>
02 #include <exception>
...
03 double division(int x, int y) noexcept {
04     if(y==0) {
05         throw std::string("Division durch 0");
06     }
07     return static_cast<double>(x)/y;
08 }
...
09 try {
10     std::cout << division(100, 0) << std::endl;
11 }
12 catch(std::string e) {
13     std::cout << e << std::endl;
14 }
15 catch(...) {
16     std::cout << "Wird auch nie erreicht\n";
17 }
...
```

In Zeile 10 führen wir nochmals unsere klassische Division durch 0 im try-Block aus. In der Funktion (Zeilen 03 bis 08) werfen wir auch in Zeile 05 eine Ausnahme vom Typ `std::string()`. Aber obwohl hier jetzt ein `catch()` für `std::string` in den Zeilen 12 bis 14 und ein alternatives `catch()` für alle anderen Fällen in den Zeilen 15 bis 17 implementiert wurde, wird diese Aufnahme wegen des Schlüsselwortes `noexcept` in Zeile 03 nicht abgefangen, sondern es wird sofort `std::terminate()` aufgerufen und das Programm beendet. Hier könnten Sie höchstens noch mit `set_terminate()` einen `terminate_handler` einrichten, der beim Aufruf von `std::terminate` aufgerufen würde.

Von `noexcept` gibt es noch eine zweite Version mit `noexcept(ausdruck)`, bei der Sie einen konstanten Ausdruck zwischen die Klammern setzen. Je nachdem, ob der Ausdruck `true` oder `false` zurückgibt, werden Ausnahmen verweigert (`= true`) oder nicht (`= false`). Ein leeres `noexcept` entspricht somit einem `noexcept(true)`. Zum Beispiel:

```
01 const bool noexc = false;
...
02 double division(int x, int y) noexcept(noexc) {
```

```

03     if(y==0) {
04         throw std::string("Division durch 0");
05     }
06     return static_cast<double>(x)/y;
07 }

```

Hier haben wir eine einfache boolesche Konstante in Zeile 01 definiert und auf `false` gesetzt. In Zeile 02 verwenden wir diese Konstante als Parameter von `noexcept()`, und daher werden jetzt wieder die Ausnahmen geworfen und können abgefangen werden. Setzen Sie den Wert von Zeile 01 auf `true`, würden alle Ausnahmen, die in `division()` geworfen werden, wieder verweigert, und es würde `std::terminate()` aufgerufen.

noexcept auf Teufel komm heraus?

An dieser Stelle muss ich noch eine wichtige Anmerkung hinzufügen, für den Fall, dass Sie jetzt hergehen und künftig alles mit `noexcept` markieren, um die Performance zu steigern. Der Grundgedanke von Ausnahmen ist nämlich nach wie vor, dass diese bei einem Fehler geworfen werden und Sie als Programmierer somit auf den Fehler reagieren können. Daher an dieser Stelle einfach der gut gemeinte Ratschlag, `noexcept` nur zu verwenden, wenn Sie sich 100 %ig sicher sind und wissen, warum Sie dies tun wollen. Das Konzept von `noexcept` ist nicht so trivial, wie es vielleicht zunächst den Anschein hat. Eine umfassendere Einführung würde hier allerdings den Rahmen des Kapitels sprengen.

10.5.5 Ausnahme-Spezifikation `unexpected()` behandeln (veraltet/deprecated)

Die Ausnahme-Spezifikation ist mit Einführung von C++11 zu deprecated (veraltet) erklärt worden und sollte daher in neuen Projekten nicht mehr verwendet werden. Hierbei handelt es sich um eine Technik, mit der Sie in einer Funktionsdeklaration angeben können (bzw. konnten), welche Ausnahmen von einer Funktion geworfen werden dürfen. Wurde keine Spezifikation angegeben, können (konnten) alle Ausnahmen geworfen werden. Ein leeres `throw()` bedeutet(e), dass keine Ausnahmen erlaubt sind. Hierfür wurde in C++11 ja mit `noexcept` eine verbesserte Alternative eingeführt.

Wurde trotzdem eine Ausnahme geworfen, welche nicht in der Ausnahme-Spezifikation aufgelistet ist (war), wird (wurde) die Funktion `std::unexpected()` aufgerufen. Hier können (konnten) Sie dann mithilfe von `std::set_unexpected()` einen `unexpected_handler` einrichten, um dort die geworfene Ausnahme zu behandeln bzw. erneut zu werfen. Wurde kein `unexpected_handler` eingerichtet, wird (wurde) `std::terminate()` aufgerufen.

Mehr Informationen

Wollen Sie mehr darüber erfahren, warum die Ausnahme-Spezifikation nicht mehr verwendet werden soll, können Sie hier mehr darüber lesen: <http://www.gotw.ca/publications/mill22.htm>.

10.6 Gefahren bei der Ausnahmebehandlung

Die Behandlung von Fehlern mithilfe von Ausnahmen ist ein nützliches Hilfsmittel in C++. Trotzdem gibt es hier auch Fallstricke, in denen man sich verheddern kann. Einige der gängigsten will ich hier kurz erläutern.

10.6.1 Ausnahmen im Konstruktor

Es ist auch möglich, Ausnahmen aus einem Konstruktor einer Klasse zu werfen. Allerdings müssen Sie hier wissen, dass der Destruktor dann nicht mehr ausgeführt wird. Der Destruktor wird nur dann ausgeführt, wenn der Konstruktor komplett ausgeführt wurde. Wenn Sie also eine Ausnahme innerhalb eines Konstruktors werfen, dann müssen Sie sich hierbei um eventuell nötige Aufräumarbeiten kümmern, weil der Destruktor nicht mehr ausgeführt wird.

10.6.2 Ausnahmen im Destruktor

Im Destruktor sollten Sie keine Ausnahme werfen. Passiert dies während des Aufräumens des Stacks (*Stack-Unwindings*), ist der weitere Verlauf des Programms undefinierbar, weil das Programm hier in einem Dilemma steckt und nicht weiß, ob es den Stack weiter aufräumen soll oder die Ausnahme behandeln, welche im Destruktor geworfen wurde. Daher wird hier gewöhnlich das Programm sofort mit `std::terminate()` beendet.

10.6.3 Aufräumprozess

Wenn Ressourcen geworfen werden, wird das Aufräumen von noch offenen Ressourcen gerne vernachlässigt oder vergessen. Folgender Codeausschnitt demonstriert das Problem:

```
...
01 try {
02     NetworkStream stream;
03     stream = OpenConnection();
```

```

04  SendData(data, stream);
05  CloseConnection(stream);
06  }
07  catch(NetworkException& err) {
08      std::err << "Fehler bei der Datenübertragung\n";
09  }
...

```

Wenn in diesem Fall die Funktion `SendData()` in Zeile **04** eine Ausnahme `NetworkException` wirft, wird der `catch`-Handler ausgeführt und gibt eine passende Fehlermeldung aus. Problematisch ist hier allerdings, dass die Verbindung in Zeile **05** niemals mehr geschlossen wird. Das kann eine ziemliche Sicherheitslücke sein, je nachdem, woraus die Verbindung bestand. In diesem Fall würden sich zwei einfache Lösungen anbieten:

- Setzen Sie wirklich nur den kritischen Code in den `try`-Block, wenn dies möglich ist. Wenn hier `SendData()` die kritische Funktion ist, dann könnten Sie den Code wie folgt umschreiben:

```

...
01  NetworkStream stream;
02  stream = OpenConnection();
03  try {
04      SendData(data, stream);
05  }
06  catch(NetworkException& err) {
07      std::err << "Fehler bei der Datenübertragung\n";
08  }
09  CloseConnection(stream);
...

```

- Behandeln Sie mögliche Ressourcen-Lecks im passenden `catch`-Block, in dem die Ausnahme auch behandelt wird:

```

...
01  NetworkStream stream=0;
02  try {
03      stream = OpenConnection();
04      SendData(data, stream);
05      CloseConnection(stream);
06  }
07  catch(NetworkException& err) {
08      if( stream != 0 )
09          CloseConnection(stream);

```

```

10     std::err << "Fehler bei der Datenübertragung\n";
11 }
...

```

Verwenden Sie RAI

Noch eine bessere Möglichkeit ist es hier, sofern Sie den Quellcode `NetworkStream` haben, RAI zu verwenden und gleich die Klasse `NetworkStream` so zu implementieren, dass beim Zerstören des Objekts (beim Konstruktoraufwurf) die Verbindung geschlossen wird.

10.6.4 Reservierter Speicher vom Heap

Dasselbe wie eben für diverse Ressourcen gilt auch, wenn Sie zur Laufzeit dynamisch Speicher vom Heap mit `new` anfordern. Auch hier wird gerne ein `delete` innerhalb eines `try`-Blocks zur Freigabe des Speichers übersprungen, wenn zuvor eine Ausnahme geworfen wurde. Auch hier sollten Sie im entsprechenden `catch`-Block den Speicher gegebenenfalls mit `delete` wieder freigeben. Die bessere Alternative ist hier allerdings, stattdessen einen neu in C++11 eingeführten Smart Pointer wie `std::shared_ptr` zu verwenden. Hier wird der Speicher automatisch freigegeben, sobald der dazugehörige `std::shared_ptr` den Gültigkeitsbereich verlässt. Mehr zu den neuen Smart Pointern können Sie in Abschnitt 16.3, »Smart Pointer (C++11)«, nachlesen.

Kapitel 11

Template-Programmierung

Mit den Templates (engl. für Schablonen) lernen Sie eine weitere fundamentale Technik zur Typparametrierung kennen. Diese Technik ermöglicht Ihnen eine generische Programmierung und typsichere Container. So verwendet bspw. die Standardbibliothek selbst Templates, um typsichere Container wie `std::vector`, `std::list` usw. zur Verfügung zu stellen. Auch viele generische Algorithmen arbeiten damit. Ein guter Grund also, einen Blick darauf zu werfen, um selbst Templates schreiben zu können.

Eigene Templates zu erstellen, kann sehr nützlich sein, wenn Sie häufig ähnliche Funktionen oder ähnliche Klassen nur mit unterschiedlichen Typen implementieren müssen. Sie haben das Prinzip bereits am Standardcontainer `std::vector` gesehen. Das Prinzip von `std::vector` bleibt ja im Grunde immer dasselbe, es unterscheidet sich nur der Typ. Daher wurde `std::vector` als **Klassen-Template** implementiert, womit Sie diesen Container jetzt mit jedem beliebigen Typ verwenden können, indem Sie den Typ zwischen die spitzen Klammern schreiben (`std::vector<int>`, `std::vector<double>`, `std::vector<KlassenXY>` usw.). Gleiches gilt auch für häufig verwendete Algorithmen wie das Suchen oder Sortieren. Auch hier macht es einem die Standardbibliothek leicht und bietet **Template-Funktionen** an, bei denen der Compiler den Typ anhand der Argumente selbst herausfindet, anstatt diese Algorithmen mehrfach für jeden Typ speziell zu schreiben.

Wie Sie hier jetzt herauslesen konnten, können Sie Klassen-Templates und Funktions-Templates erstellen. Der Vorteil ist zunächst, dass Sie eine Klasse oder Funktion nur einmal schreiben müssen. Auch die Instanzen solcher Templates werden nur angelegt, wenn diese tatsächlich verwendet werden. Wenn Sie ohne Templates eine `funktionXY()` mit Parameter für `int`, `double` und `long` anlegen, wird für alle drei Versionen ein Maschinencode erzeugt. Erstellen Sie diese Funktion als Funktions-Template, wird nur ein Code von den Versionen erzeugt, die Sie tatsächlich im Programm verwenden. Das kann die ausführbare Datei schlanker halten. Auch die Überarbeitung von Code wird vereinfacht, denn anstatt alle Versionen einer bestimmten Funktionalität anzupassen, müssen Sie nur noch den Code des einen Templates

anpassen. Das spart Zeit und verhindert Fehler, welche bei der mehrfachen Codierung entstehen könnten.

Keine Reduzierung von Maschinencode

Wohlgemerkt, Templates reduzieren niemals den Umfang des Maschinencodes, sondern ersparen dem Programmierer lediglich, ähnliche Funktionen und Klassen für verschiedene Datentypen mehrmals schreiben zu müssen.

11.1 Funktions-Templates

Ein Funktions-Template sollte in der Lage sein, Argumente von verschiedenen Typen anzunehmen und auch, wenn nötig, verschiedene Rückgabewerte zurückzugeben. Hierzu soll eine einfache Funktion demonstriert werden, welche den kleinsten von zwei Werten zurückgibt:

```
01 int minVal(int v1, int v2) {
02     if( v1 < v2 ) {
03         return v1;
04     }
05     return v2;
06 }
```

Diese Funktion liefert den kleinsten von zwei übergebenen Werten an den Aufrufer zurück. Dank der impliziten Konvertierung des Compilers lässt sich diese Funktion auch mit `short` und `char` als Parametern verwenden. Wenn Sie jetzt aber eine Version für eine Gleitkommazahl verwenden wollen, müssen Sie eine weitere Version mit `float`, `double` und/oder `long double`, also Parameter und Rückgabewert implementieren:

```
01 double minVal(double v1, double v2) {
02     if( v1 < v2 ) {
03         return v1;
04     }
05     return v2;
06 }
```

Jetzt haben Sie zwei Versionen von `minVal()` und können diese Funktion mit `Integern` und `Gleitpunktzahlen` aufrufen. Wenn Sie diese Funktion auch mit `std::string` oder besser noch einer eigenen Klasse aufrufen wollen, müssen Sie weitere Versionen dafür implementieren.

std::min() und std::max()

Hier soll noch hinzugefügt werden, dass die C++-Standardbibliothek in der Headerdatei `<algorithm>` längst Funktions-Templates mit `std::min()` und `std::max()` anbietet, welche den kleinsten bzw. größten Wert von zwei übergebenen Werten zurückgibt. Mit Einführung von C++11 funktionieren `std::min()` und `std::max()` auch für mehrere Elemente, wenn man `{...}` verwendet.

Der Code hierfür ist eigentlich immer derselbe, nur die Typen sind anders. In solch einem Fall bietet sich jetzt ein Funktions-Template an, bei dem wir nur eine Version erstellen und den Compiler veranlassen, uns die benötigten weiteren Versionen zu generieren.

11.1.1 Funktions-Templates implementieren

Wenn Sie ein Funktions-Template definieren wollen, so müssen Sie dies mit folgendem Präfix einleiten:

```
template <typename T>
```

Der Parameter `T` wird hierbei für den formalen Datentyp verwendet, der bei der Definition der Funktion zum Einsatz kommt. `T` steht hierbei für einen beliebigen Datentyp, wie bspw. `int`, `double`, `long` oder auch eine andere Klasse. Sie müssen hier übrigens nicht zwangsläufig `T` für den Namen des formalen Datentyps verwenden und können auch jeden beliebigen (gültigen) Bezeichner nehmen. Allerdings findet man in der gängigen Praxis meistens den Buchstaben `T` vor.

template <class T> oder template <typename T>

In einigen Programmen kann es sein, dass Sie hier auch `template <class T>` anstatt `template <typename T>` vorfinden, was ebenfalls richtig ist. Beide Varianten sind absolut gleichbedeutend. Nur die Version mit `typename` wurde etwas später in C++ eingeführt und ist somit etwas jüngerer Datums. Ich persönlich bevorzuge `typename`, weil es sich besser herauslesen lässt, dass `T` ein beliebiger Typ (wie `int`, `double` usw.) sein kann, der nicht nur als Klasse definiert wird. Bei der Verwendung des Schlüsselwortes `class` entsteht eher der Eindruck, dass es sich bei `T` um eine Klasse handeln muss. Häufig wird hierbei auch `class` verwendet, wenn erwartet wird, dass `T` eine Klasse ist, und `typename`, wenn `T` ein beliebiger eingebauter Typ (`int`, `double` etc.) sein sollte.

template template Parameter

So ganz ohne schlechtes Gewissen kann ich die Hinweisbox darüber nicht stehen lassen. Es gibt nämlich tatsächlich eine Gelegenheit, bei der es nicht egal ist, ob Sie `typename` oder `class` verwenden, und zwar bei `template-template-Parameter` – also wenn Sie innerhalb eines `Template-Parameters` ebenfalls einen `Template-Parameter` verwenden:

```
template <template <typename> class    T> class CL01 { }; //Ok!!!
template <template <typename> typename T> class CL02 { }; //Fehler!!!
```

Hier **muss** im fett hervorgehobenen Teil `class` statt `typename` verwendet werden. Auch wenn Sie diese (Klassen-)Templates jetzt noch nicht verstehen und wir auch gar nicht mehr näher darauf eingehen werden, behalten Sie dies vielleicht trotzdem im Hinterkopf, falls Sie irgendwann einmal in Ihrem Programmiererleben auf einen `template-template-Parameter` stoßen oder ihn schreiben müssen – was mir persönlich noch nie untergekommen ist. Aber es sollte halt immerhin erwähnt werden.

Somit sieht die Definition unseres Funktions-Templates für `minVal()` wie folgt aus:

```
00 // kapitel11/function_template/minval.cpp
...
01 template <typename T>
02 T minVal(T v1, T v2) {
03     if( v1 < v2 ) {
04         return v1;
05     }
06     return v2;
07 }
...
```

Vergleichen Sie das Funktions-Template mit unseren ursprünglichen Definitionen, so wurde hier lediglich statt eines Datentyps beim Rückgabewert und Parameter der Buchstabe `T` (für den Typ) verwendet. Den formalen Datentyp `T` können Sie auch mit lokalen Daten innerhalb der Funktion verwenden. Auch ersetzt der Compiler bei der Generierung den formalen Datentyp durch den tatsächlichen Datentyp. Ein einfaches Beispiel hierzu:

```
...
01 template <typename T>
02 void mySwap(T& v1, T& v2) {
03     T tmp = v1;
04     v1 = v2;
05     v2 = tmp;
06 }
...
```

Dieses Funktions-Template tauscht Daten vom Typ `T` miteinander. In Zeile 03 wurde hier eine lokale Variable vom Typ `T` verwendet, welche als temporärer Speicher für den Austausch zweier Daten vom Typ `T` dient. Die zu tauschenden Werte müssen ebenfalls vom selben Datentyp sein, was hier ja mit den Parametern vom selben Typ (auch `T`) der Fall ist.

Ebenso könnten Sie innerhalb von Funktionen, welche aus einem Template erstellt wurden, ein weiteres Funktions-Template aufrufen. Das folgende Beispiel erweitert die Funktion `mySwap()` um die Funktion, dass die Werte nur dann getauscht werden, wenn der Wert im linken Parameter größer ist als der Wert im rechten Parameter:

```
00 // kapitel11/function_template/greaterThan.cpp
...
01 template <typename T>
02 bool greaterThan(T v1, T v2) {
03     return v1 > v2;
04 }

05 template <typename T>
06 void mySwap(T& v1, T& v2) {
07     if( greaterThan(v1, v2) ) {
08         T tmp = v1;
09         v1 = v2;
10         v2 = tmp;
11     }
12 }
...
```

Jetzt werden mit `mySwap()` nur die Werte vertauscht, wenn `greaterThan()` in Zeile 07 gleich `true` zurückgibt.

Nun haben Sie zwar das Funktions-Template definiert, aber der Compiler erzeugt aufgrund der Definition eines Funktions-Templates im Gegensatz zu normalen Funktionen noch keinen Maschinencode, weil er noch gar nicht wissen kann, für welchen Datentyp er eine Version generieren soll. Da der Datentyp des formalen Parameters (hier `T`) noch nicht bekannt ist, wird hier auch von einer **unvollständigen Definition** gesprochen.

Funktions-Templates über mehrere Module

Da es sich bei einem Funktions-Template allein um eine unvollständige Definition handelt, stellt sich gleich die Frage, wohin damit bei umfangreicheren Projekten mit mehreren Modulen? Eine häufige Empfehlung hierzu lautet, dass man die Definition des Funktions-Templates in eine Headerdatei stellt, damit das Template in allen Modulen zur Verfügung steht, wo die Headerdatei eingebunden wird.

Der Grund dafür ist, dass die endgültige Funktion (Maschinencode) ja erst beim Aufruf einer durch das Funktions-Template vorgegebenen Funktion erstellt wird, und dafür benötigt der Compiler den Code der Funktion.

11.1.2 Aufruf des Funktions-Templates

Nachdem Sie ein Funktions-Template definiert haben, können Sie dieses Template wie eine ganz normale Funktion aufrufen. Der Compiler kümmert sich darum, dass die richtige Version verwendet oder eben entsprechender Maschinencode generiert wird. Zum Beispiel:

```
00 // kapitel11/function_template/minval.cpp
...
01 std::cout << minVal(100, 101) << std::endl;
02 std::cout << minVal(1.832, 1.831) << std::endl;
03 std::cout << minVal(std::string("aab"), std::string("aaa"));

04 int val1=100, val2=200;
05 mySwap(val1, val2);
06 std::cout << val1 << "/" << val2 << std::endl;

07 double dval1=3.142, dval2=3.141;
08 mySwap(dval1, dval2);
09 std::cout << dval1 << "/" << dval2 << std::endl;
...
```

Im Beispiel von `minVal()` sehen Sie hier, wie diese Funktion einmal mit Integer-Werten (Zeile 01), Gleitkommazahlen (Zeile 02) und mit `std::string` (Zeile 03) aufgerufen wurde. Gleiches wurde auch mit `mySwap()` mit `int` und `double` gemacht (Zeilen 05 und 08). Sicherlich fragen Sie sich, wie das im Compiler vor sich geht?

Der Compiler sucht bei der Übersetzung zuerst nach einer Funktion, die zu dem verwendeten Datentyp des Funktionsaufrufes passt, und verwendet diese. Findet der Compiler aber keine passende Funktion, sucht er nach einem Funktions-Template. Findet dieser ein Funktions-Template, ersetzt er den formalen Datentyp (hier `T`) durch den echten Datentyp und generiert eine Funktion daraus. In unserem Beispiel generiert der Compiler bspw. drei Versionen von `minVal()`, einmal für ein Integer (`int`), einmal für eine Gleitkommazahl (`double`) und einmal eine Version mit `std::string`. Die Funktion wird also erst bei einem Funktionsaufruf mit dem entsprechenden Datentyp mithilfe des Funktions-Templates **instanziiert**.

Von der Funktion `mySwap()` hingegen werden nur zwei Versionen generiert, weil diese Funktion nur mit `int` und `double` aufgerufen wurde.

Funktions-Template mit eigener Klasse als Parameter

Funktions-Templates arbeiten hervorragend mit den eingebauten Typen von C++ (bspw. `char`, `int`, `double`, `long long` usw.) und in der Regel auch mit fast allen Standard-container-Klassen von C++, wie Sie im Beispiel mit `std::string` gesehen haben. Wenn Sie allerdings eigene Klassen verwenden, gelten auch hier alle Regeln, wie bei der Verwendung eigener Klassen als Funktionsargument – letztendlich generiert der Compiler ja aus dem Funktions-Template eine gewöhnliche Funktion mit entsprechenden Parametern. Wenn Sie folgendes Beispiel ausprobieren, wird sich der Compiler bei Ihnen beschweren:

```
...
01 class C {
02     int val;
03 public:
04     C(int v=0) : val(v) {}
05     void print() const { std::cout << val << std::endl; }
06 };
...
07 C c01(101);
08 C c02(102);
09 C c03 = minVal(c01, c02); // Fehler !!!
10 c03.print();
...
```

Die Compiler beschwert sich, dass er hier den Operator `operator<` vermisst, welcher in unserem Funktions-Template `minVal()` verwendet wird. Standardcontainer-Klassen bieten ja schon den Komfort, dass sie `operator<` für alle eingebauten C++-Typen bereits von Haus aus implementiert haben. Da im Funktions-Template ein Vergleich mit dem `operator<` verwendet wird, müssen Sie daher in der Klasse `C` diesen Operator auch überladen und implementieren. Die Operatorüberladung wird in Kapitel 8, »Operatoren überladen«, behandelt.

Spezialisierung

Nicht immer lässt sich das Problem wie hier mit einer einfachen Operatorüberladung lösen. Ist eine Klasse zu speziell und erfordert sie eine besondere Behandlung, die sich mit dem vorhandenen Funktions-Template nicht vereinbaren lässt, weil kein sinnvolles Ergebnis zurückgeliefert wird, können Sie auch eine Spezialisierung für dieses Funktions-Template definieren. Hierbei wird das Funktions-Template durch eine separat definierte Funktion (die Spezialisierung) überladen. Mehr zur Spezialisierung von Funktions-Templates können Sie in Abschnitt 11.1.3, »Funktions-Template spezialisieren«, nachlesen.

Folgender Codeausschnitt implementiert jetzt `operator<` als friend-Funktion in der Klasse `C` und sorgt dafür, dass auch vom Funktions-Template `minVal()` eine Version für unsere Klasse `C` instanziiert werden kann:

```
00 // kapitel11/function_template/minval2.cpp
...
01 class C {
02     int val;
03 public:
04     C(int v=0) : val(v) {}
05     friend bool operator<(C &c1, C &c2);
06     void print() const { std::cout << val << std::endl; }
07 };

08 bool operator<(C &c1, C &c2) {
09     return (c1.val < c2.val);
10 }
...
```

Typübereinstimmung

Wenn Sie Funktions-Templates verwenden, sollten Sie noch wissen, dass beim Auflösen des formalen Datentyps keine automatische Typkonvertierung durchgeführt wird. Wenn Sie bspw. ein Funktions-Template mit zwei formalen Datentypen mit `T` definiert haben, dann müssen beim Aufruf des Templates beide Typen miteinander übereinstimmen. Selbst im Prinzip einfachste implizite Anpassungen von Integer-Typen wie `short`, `int` und `long` führen zu einem Compilerfehler – der Compiler weigert sich schlicht, aus dem Funktions-Template eine Funktion zu generieren. Ein Beispiel:

```
01 int ival=123;
02 short sval=321;
03 int mval01 = minVal(ival, sval); // Fehler !!!
```

Der Compiler müsste hier eine Funktion `minVal(int, short)` generieren. Aber bei der Definition des Funktions-Templates für `minVal()` wurde dies nicht vereinbart und mit `T` eben nur ein formaler Datentyp vorgegeben. Wenn Sie ein Funktions-Template mit unterschiedlichen Datentypen als Parameter benötigen, so ist dies auch möglich, wie Sie im folgenden Abschnitt 11.1.3, »Funktions-Template spezialisieren«, noch erfahren werden.

In unserem Beispiel in Zeile **03** könnten Sie den Fehler nur beheben, indem Sie eine explizite Umwandlung des ersten oder zweiten Parameters wie folgt durchführen:

```
03 int mval01 = minVal(ival, static_cast<int>(sval));
```

Hiermit würde eine neue Funktion `minVal(int, int)` instanziiert werden. Natürlich könnten Sie stattdessen auch den ersten Parameter nach `short` umwandeln, um eine Funktion `minVal(short, short)` zu generieren, aber eine Umwandlung von `int` nach `short` ist doch immer etwas kritischer zu betrachten, weil dabei Informationen verloren gehen könnten.

Alternativ würde sich hier auch eine explizite Typangabe für das Template-Argument des formalen Datentyps (hier `T`) anbieten. Darauf wird in Abschnitt 11.1.5, »Explizite Template-Argumente«, noch gesondert eingegangen. Trotzdem will ich Ihnen die explizite Typangabe für das Template-Argument für unser Beispiel nicht vorenthalten:

```
03 int mval01 = minVal<int>(ival, sval);
```

Hiermit generiert uns der Compiler eine Funktion `minVal(int, int)`.

11.1.3 Funktions-Template spezialisieren

Nicht immer eignet sich ein Funktions-Template für Typen gleichermaßen. Bei dem einen oder anderen Typ kann es sein, dass einfach kein sinnvolles Ergebnis zurückgegeben wird, und bei einem anderen Typ (bspw. einer eigenen Klasse) kann es sein, dass bestimmte Anweisungen gar nicht mit dem Typ ausgeführt werden können. Betrachten Sie hierzu folgenden Codeausschnitt, der den Compiler veranlasst, einmal eine `std::string`-Version und einmal eine Version `const char*` aus dem Funktions-Template `minVal()` zu instanziiieren:

```
00 // kapitel11/function_template/minval_spezial.cpp
...
01 std::string str01("abba");
02 std::string str02("aabaaaa");
03 std::cout << minVal(str01, str02) << std::endl;
...
04 std::cout << minVal("aaa", "abaaa") << std::endl; // ???
```

Der Compiler generiert von beiden Versionen eine Funktion. Einmal `minVal(std::string, std::string)` und einmal `minVal(const char*, const char*)`. Im Beispiel von Zeile 03 mit `minVal(std::string, std::string)` wird, wenn Sie sich ein wenig mit der Stringbibliothek und dem darin implementierten `operator<()` befassen haben, der lexikografisch kleinere String zurückgegeben. Die Vergleichsoperatoren von `std::string` arbeiten lexikografisch, das heißt so sortiert, wie dies bspw. bei einem Lexikon üblicherweise der Fall ist. Was ist aber, wenn Sie dies hier gar nicht im Sinn gehabt hatten? Was ist, wenn Sie stattdessen wirklich den String zurückhaben wollen, der kleiner ist – also weniger Buchstaben enthält?

Auch die Generierung und Verwendung von `minVal(const char*,const char*)` in Zeile 04 ist eher fragwürdig, weil ein direkter Vergleich, wie er im Funktions-Template vorgenommen wird, nur die kleinere der beiden Adressen zurückgibt, in der die C-Strings gespeichert sind.

Hier haben Sie also zunächst einen Fall, in dem es nicht zum gewünschten Ergebnis kommt, und einen zweiten Fall, in dem kein sinnvolles Ergebnis geliefert wird. Für solche und weitere Fälle können Sie eine **Spezialisierung** erstellen. Hierfür müssen Sie lediglich dieselbe Funktion separat mit den entsprechenden Typen definieren und überladen.

Allerdings können (sollten) Sie nicht hergehen und eine ganz gewöhnliche Funktionsüberladung implementieren. Gerade bei umfangreicheren Projekten mit mehreren Modulen wirft dies ein Problem auf. Wird nämlich eine Spezialisierung in einem anderen Modul als Funktions-Template definiert, weiß der Compiler nicht mehr, ob hier eine Instanziierung des Funktions-Templates oder einer Spezialisierung vorliegt und meldet gewöhnlich einen Fehler, nämlich eine doppelte Definition.

template<>

Damit Spezialisierungen nicht mit den Funktions-Templates kollidieren, bietet der Standard eine eigene Syntax dafür an, wie Spezialisierungen eingeleitet werden müssen. Hierzu können Sie folgendes Präfix vor die Spezialisierung stellen:

```
template <>
```

Der Datentyp, für den dieses Funktions-Template jetzt verwendet werden soll, wird nach dem Funktionsnamen zwischen spitze Klammern geschrieben. Wenn sich der Datentyp aus den Parametern der Funktion von alleine ergibt, so können Sie diese Angabe zwischen den spitzen Klammern nach dem Funktionsnamen auch weglassen.

Bezogen auf unser Funktions-Template `minVal()` könnte die Spezialisierung für die Typen `std::string` und `const char*` jetzt wie folgt aussehen:

```
00 // kapitel11/function_template/minval_spezial.cpp
...
01 template <typename T>
02 T minVal(T v1, T v2) {
03     if( v1 < v2 ) {
04         return v1;
05     }
06     return v2;
07 }
```

```

08 template<>
09 std::string minVal <std::string>(std::string s1,
                                std::string s2) {
10     if(s1.size() < s2.size()) {
11         return s1;
12     }
13     return s2;
14 }

15 template<>
16 const char* minVal <const char*> (const char *s1,
                                const char *s2) {
17     int val1 = std::char_traits<char>::length(s1);
18     int val2 = std::char_traits<char>::length(s2);
19     if( val1 < val2 ) return s1;
20     return s2;
21 }

```

Die Spezialisierung von `std::string` in den Zeilen 08 bis 14 wird mit `template<>` eingeleitet. Der Datentyp für das Funktions-Template wird hier mit `<std::string>` nach dem Funktionsnamen angegeben. In der Funktion wird für den Vergleich die Methode `size()` in Zeile 10 verwendet. Diese Methode liefert die Anzahl der Zeichen zurück, welche sich im String befinden. Je nachdem, welcher der beiden Strings weniger Buchstaben enthält, der wird zurückgegeben.

Ähnlich sieht es auch mit unserer Spezialisierung für `const char*` in den Zeilen 15 bis 21 aus. Für die Länge des C-Strings verwenden wir hierzu in den Zeilen 17 und 18 die statische Methode `length()` aus `<string>`, welche die Länge eines nullterminierten C-Strings zurückgibt. Diese Funktion ist die Alternative zur Uralt C-Funktion `strlen()` aus der Headerdatei `<cstring>`. Auch hier wird entsprechend der C-String zurückgegeben, der weniger Buchstaben enthält – eben der kleinste C-String.

Spezialisierungen werden bevorzugt behandelt

Dank des Schlüsselwortes `template<>` vor einer Spezialisierung können Sie sich sicher sein, dass der Compiler immer erst die Spezialisierung aufruft. Haben Sie also ein Funktions-Template und eine Spezialisierung definiert, wird immer die Spezialisierung verwendet. Aber wenn eine globale Funktion mit dem entsprechenden Datentyp der Parameter definiert wurde, verwendet der Compiler die globale Funktion. Anstatt also eine Spezialisierung zu schreiben, könnten Sie auch einfach eine globale Funktion mit dem passenden Datentyp als Parameter schreiben.

Spezialisierung von Funktions-Template verwenden?

An dieser Stelle muss noch angemerkt werden, dass die Spezialisierung von Funktions-Templates auch zu vielen Kontroversen geführt hat, und auch Probleme mit sich bringt. Ich will hier nicht mehr näher darauf eingehen, aber der folgende Artikel von Herb Sutter beschreibt dies sehr gut: <http://www.gotw.ca/publications/mill17.htm>

11.1.4 Mehrere Template-Parameter verwenden

Natürlich sind Templates nicht nur auf einen formalen Datentyp beschränkt. Ganz im Gegenteil, sofern es Sinn macht, können beliebig viele formale Datentypen definiert werden. Logischerweise sollten sich die formalen Datentypen und die entsprechenden Bezeichner unterscheiden. Hierzu ein einfaches Funktions-Template mit zwei formalen Datentypen:

```
00 // kapitel11/function_template/moreformalT01.cpp
01 #include <typeinfo>
...
02 template <typename T, typename U>
03 void func(T t, U u) {
04     std::cout << "Inhalt : " << t << std::endl;
05     std::cout << "Inhalt : " << u << std::endl;
06     if( typeid(t).name() == typeid(u).name() ) {
07         std::cout << t << " und " << u
                   << " sind vom selben Typ\n";
08     }
09 }
...
10 func("Test", 123.123); // func(const char*, double)
11 func('A', 12345L);     // func(char, long)
12 func(2000, 3.1415f);   // func(int, float)
13 func(987, 123);        // func(int, int)
14 func(123.123, 133.122); // func(double, double)
...
```

Bei diesem Funktions-Template werden zwei formale Datentypen (T und U) verwendet. Häufig sieht man hier auch T1 und T2, aber ich finde, dass es mit T und U deutlicher zu unterscheiden und besser zu lesen ist. Dieses Template können Sie jetzt mit beliebigen Typen aufrufen und instanziiieren lassen, wie Sie in den Zeilen 09 bis 13 auch sehen können. Hinter dem Aufruf finden Sie als Kommentar, was für Funktionen der Compiler hier instanziiieren muss. Die Überprüfung mit typeid in den Zeilen 05 bis 07 dient nur der Demonstration, um gegebenenfalls mitzuteilen, dass eine

Funktion mit zwei gleichen Typen (wie dies beim Aufruf der Zeilen 12 und 13 der Fall ist) instanziiert wurde – was logischerweise auch gültig ist.

Es sollte hierbei nicht unerwähnt bleiben, dass Sie selbstverständlich trotz formaler Datentypen im Funktions-Template auch eingebaute Datentypen mit den üblichen Features verwenden können:

```
template <typename T>
void func(T t, double dval) {
...
}
```

Ebenso ist es möglich, eine bereits generische Funktion als Funktions-Template erneut zu überladen oder gar zu spezialisieren. Dies hört sich komplizierter an, als es ist, daher auch hierzu ein einfaches Beispiel:

```
00 // kapitel11/function_template/moreformalT02.cpp
...
01 template <typename T, typename U>
02 void func(T t, U u) {
03     std::cout << "func(T t, U u)\n";
04     std::cout << "Inhalt : " << t << std::endl;
05     std::cout << "Inhalt : " << u << std::endl;
06 }

07 template <typename T>
08 void func(T t, double dval) {
09     std::cout << "func(T t, double dval)\n";
10     std::cout << "Inhalt : " << t << std::endl;
11     std::cout << "Inhalt : " << dval << std::endl;
12 }

13 template <>
14 void func(double dval1, double dval2) {
15     std::cout << "func(double, double)\n";
16     std::cout << "Inhalt : " << dval1 << std::endl;
17     std::cout << "Inhalt : " << dval2 << std::endl;
18 }

...
19 func("Test", 123.123); // func(T t, double dval)
20 func('A', 12345L);     // func(T t, U u)
21 func(2000, 3.1415f);   // func(T t, U u)
22 func(987, 123);        // func(T t, U u)
23 func(123.123, 133.122); // func(double, double)
...
```

In den Zeilen 01 bis 06 sehen Sie unsere bereits bekannte generische Funktion mit mehreren Parametern. Dann, in den Zeilen 07 bis 12, finden Sie eine weitere überladene Version mit einem zweiten festen Parameter. Zur Demonstration wurde diese Funktion nochmals in den Zeilen 13 bis 18 für `double` spezialisiert (es handelt sich aber immer noch um ein Template). Alternativ hätten Sie stattdessen auch einfach eine globale Funktion `void func(double, double)` definieren können.

Die Spezialisierung in den Zeilen 13 bis 18 wird auch nur dann aufgerufen, wenn `func()` mit zwei `double`-Werten aufgerufen wird, wie dies in Zeile 23 der Fall ist. Die überladene Version von den Zeilen 07 bis 12 wird instanziiert und verwendet, wenn der Funktionsaufruf mit einem `double` als zweitem Parameter erfolgt, wie dies in Zeile 19 der Fall ist. In Zeile 21 wird die Gleitkommazahl mit dem Suffix `f` als `float` gekennzeichnet, daher wird hier die generische Funktion aus den Zeilen 01 bis 06 verwendet. Bei allen anderen Funktionsaufrufen wird ebenfalls die Version mit zwei formalen Datentypen verwendet.

Templates mit variabler Argumentanzahl

Neu in C++11 wurde auch die Möglichkeit eingeführt, Templates mit einer variablen Anzahl an Argumenten zu verwenden. Solche Templates werden auch als *Variadic Templates* bezeichnet. Mehr zu dieser neuen Möglichkeit finden Sie in Abschnitt 11.3, »Templates mit variabler Argumentanzahl (C++11)«.

11.1.5 Explizite Template-Argumente

Die Instanziierungen der Funktions-Templates, wie Sie sie bisher aufgerufen haben, waren alle implizit. Das bedeutet, sobald Sie ein Funktions-Template mit einem bestimmten Typ aufgerufen haben, wurde der formale Datentyp des Templates durch den entsprechenden Typ ersetzt und ein Maschinencode dafür erzeugt. Rufen Sie bspw. unser bekanntes Funktions-Template mit `minVal(123, 456)` auf, erzeugt der Compiler daraus eine Funktion `minVal(int, int)`.

Sie haben aber auch die Möglichkeit, eine solche Instanziierung explizit zu machen – also dem Compiler vorzugeben, von welchem Typ er die formalen Datentypen ersetzen und einen Maschinencode erzeugen soll. Um jetzt explizit die Argumente anzugeben, müssen Sie nur die Template-Argumente in spitzen Klammern hinter dem Funktions-Template-Namen angeben. Rufen Sie das Funktions-Template jetzt mit `minVal<double>(123, 456)` auf, erzeugt der Compiler durch die explizite Angabe von `double` zwischen den spitzen Klammern tatsächlich auch eine Funktion `minVal(double, double)`.

Das folgende Listing demonstriert zwei Fälle, in denen die explizite Angabe von Template-Argumenten sinnvoll sein kann:

```

00 // kapitel11/function_template/explicitTemplate.cpp
...
01 template <typename T, typename U>
02 T half(U u) {
03     return u / 2;
04 }
05 template <typename T>
06 T minVal(T v1, T v2) {
07     if( v1 < v2 ) {
08         return v1;
09     }
10     return v2;
11 }
...
12 int ival = 3;
13 double dval = half (ival); // Fehler !!!
14 std::cout << dval << std::endl;
15 std::cout << minVal(ival, dval) << std::endl; // Fehler !!!
...

```

Mit dem Aufruf in Zeile 13 kann der Compiler nichts anfangen und weiß nicht, was er für den Rückgabewert verwenden soll. Der Datentyp des Rückgabewertes kann hier nicht aus dem Funktionsaufruf abgeleitet werden. In diesem Beispiel können Sie anhand der Zuweisung sehen, dass es hier ein `double` sein soll. Aber der Compiler muss ja instanziierten und kann das mit der Zuweisung nicht erkennen. Hier müssen Sie also den Datentyp explizit angeben.

Den zweiten Aufruf in Zeile 15 kennen Sie bereits aus einem früheren Beispiel, wo die Typübereinstimmung beschrieben wurde. Auch hier meldet der Compiler einen Fehler, weil es hier nur einen formalen Datentyp mit `T` gibt und beim Funktionsaufruf zwei verschiedene Typen verwendet wurden. Bei Funktions-Templates werden keine impliziten Konvertierungen durchgeführt. Hier können Sie entweder einen `static_cast<>` innerhalb der Parameter verwenden oder eben auch den Datentyp explizit angeben.

Hier nun die Lösung, wie Sie die Typen für den oder die formalen Parameter explizit angeben, womit Sie den Compiler veranlassen, eine Funktion exakt mit diesem Typ zu instanziierten:

```

00 // kapitel11/function_template/explicitTemplate.cpp
...
12 int ival = 3;
13 double dval = half<double, int>(ival);
14 std::cout << dval << std::endl;
15 std::cout << minVal<double>(ival, dval) << std::endl;
...

```

Mit der expliziten Angabe der Template-Argumente in Zeile 13 veranlassen Sie den Compiler, eine Funktion mit `double` als Rückgabewert für `T` und `int` als Parameter für `U` zu instanzieren. In diesem Beispiel hätten Sie auch das `int` in den spitzen Klammern weglassen und nur `half<double>(ival)` verwenden können und so den Compiler den Datentyp für den Parameter wieder selbst ermitteln lassen können. Das `double` würde für den Rückgabewert verwendet. Auch hier werden die formalen Datentypen in derselben Reihenfolge abgearbeitet, wie diese im Funktionskopf des Templates stehen.

Beim zweiten Fall in Zeile 15 wird jetzt ebenfalls durch die explizite Angabe von `double` eine Funktion mit `double`-Parametern (`double minVal(double,double);`) erzeugt – egal, mit welchen Typen Sie die Funktion aufgerufen haben. Dies bedeutet auch, dass eine implizite Konvertierung nach den C++-Regeln möglich sein sollte, was hier mit `int` nach `double` kein Problem darstellt. Sie könnten aber jetzt nicht hergehen und die Funktion mit einem `const char*` aufrufen.

11.1.6 Methoden-Templates

Es ist auch möglich, Methoden-Templates ohne Klassen-Templates (siehe Abschnitt 11.2, »Klassen-Templates«) zu erstellen. Dies kann bspw. nützlich sein, wenn Sie eine Klasse geschrieben haben, in der viele überladene Methoden vorhanden sind, wie dies im folgenden Beispiel der Fall ist:

```
01 class Server {
...
02 public:
03     void send_data(int val) {...}
04     void send_data(double val) {...}
05     void send_data(long long val) {...}
06     void send_data(std::string val) {...}
07     void send_data(const char* val) {...}
08 };
...
09 Server s01;
10 s01.send_data(100);           // int
11 s01.send_data(123.123);      // double
12 s01.send_data(12341234LL);   // long long int
13 s01.send_data(std::string("Text")); // std::string
14 s01.send_data("Text");       // const char*
...
```

Hier finden Sie in der Klasse `Server` die Methode `send_data()` mehrfach überladen für verschiedene Datentypen vor. Wenn hier mehrfach derselbe Code für die überlade-

nen Methoden verwendet wird, können Sie sich die Tipparbeit sparen indem Sie eine Methoden-Template für `send_data()` verwenden. Eine mögliche Implementierung könnte wie folgt aussehen:

```
01 class Server {
...
02 public:
03     template <typename T> void send_data(T val);
04 };
...
05 template <typename T> void Server::send_data(T val) { ... }
...
```

Hier sehen Sie die Deklaration des Methoden-Templates in Zeile 03 und die Definition in Zeile 05.

11.2 Klassen-Templates

Die Template-Technik, wie Sie diese mit den Funktions-Templates eben kennengelernt haben, ist nicht nur auf Funktionen beschränkt, sondern lässt sich auch mit Klassen verwenden. Um genau zu sein, wäre C++ ohne Klassen-Templates wohl nicht so populär und beliebt wie es heute ist. Die Standardbibliothek definiert nämlich sehr viele Klassen-Templates, wovon Sie selbst (un-)wissentlich bereits `std::vector` oder `std::string` verwendet haben. Bei `std::vector` haben Sie den Typ ja in den spitzen Klammern angegeben. `std::string` hingegen ist eine Instanziierung aus dem Typ `char` und intern wie folgt definiert:

```
typedef basic_string<char> string;
```

Neben den Standardcontainer-Klassen wie `std::vector`, `std::list` usw. sind auch die Stream-Klassen für die Ein-/Ausgabe (siehe Kapitel 13, »Die Stream-Ein-/Ausgabeklassen von C++«) letztlich meist als Instanziierung von `char` implementiert.

Zum Weiterlesen

Bevor Sie sich nach diesem Kapitel hinsetzen und eigene Klassen-Templates wie einen Stapel, Listen usw. schreiben, empfehle ich Ihnen, erst einmal Kapitel 12, »Container, Iteratoren, Algorithmen und Hilfsmittel«, durchzusehen, wo wir auf die in C++ vorhandenen Standardcontainer eingehen werden. Ich bin mir sicher, hier ist fast immer das dabei, was Sie suchen. Wenn nicht, dann wissen Sie anschließend, wie Sie selbst ein Klassen-Template erstellen können.

11.2.1 Klassen-Templates implementieren

Zur Einleitung will ich Ihnen zeigen, wie Sie selbst einen, wenn auch ganz primitiven, Behälter (Containerklasse) erstellen können, welcher (fast) jeden beliebigen Typ aufnehmen kann. Die Einleitung eines Klassen-Templates erfolgt auch hier wieder mit dem aus Funktions-Templates bereits bekannten

```
template <typename T>
```

Auch hier steht `T` wieder für den formalen Datentyp, den der Compiler bei der Instanziierung, also der Erzeugung des Maschinencodes, durch den gewünschten Typ ersetzt. Der Name `T` kann wieder ein beliebiger gültiger Bezeichner sein – `T` ist aber auch hier eine sehr gängige Verwendung. Ebenfalls können Sie wieder `<class T>` statt `<typename T>` verwenden. Auf den Unterschied zwischen `class` und `typename` wurde bereits in Abschnitt 11.1.1, »Funktions-Templates implementieren«, eingegangen.

Hierzu nun die Definition eines kompletten Klassen-Templates:

```
00 // kapitel11/class_template/myContainer01.cpp
...
01 template <typename T>
02 class myContainer {
03     T data;
04 public:
05     void setData(const T& d) {data=d;}
06     T getData() const {return data;}
07 };
...
```

In den Zeilen **01** bis **07** haben Sie ein Klassen-Template `myContainer<T>` definiert, welches sich jetzt fast wie `std::vector<T>` verwenden lässt – natürlich fehlen hier noch viele Funktionalitäten, aber als theoretischer Vergleich taugt es schon. Würden Sie jetzt ein Objekt wie `myContainer<int>` anlegen, würde der Compiler alle formalen Datentypen (hier `T`) durch `int` ersetzen und einen Maschinencode dazu erstellen. Mehr zur Instanziierung von Objekten als Klassen-Template erfahren Sie in Kürze in Abschnitt 11.2.3, »Objekte aus Klassen-Templates erzeugen«.

Klassen-Templates mit struct und union

Klassen-Templates lassen sich neben `class` auch für und mit `struct` und `union` definieren.

11.2.2 Methoden von Klassen-Templates implementieren

Im eben erstellten Beispiel des Klassen-Templates `myContainer<T>` konnten Sie auch sehr schön sehen, dass es ohne Weiteres möglich ist, den formalen Datentyp bei den Parametern oder dem Rückgabewert von Methoden zu setzen.

Werteübergabe als Referenz oder Zeiger bevorzugen

Hier fehlt noch die Empfehlung, dass man die Übergabe von Werten, wie im Beispiel mit `setData(const T&)`, als Referenz (oder Zeiger) realisiert und nicht als Kopie übergibt. Denn wird das Klassen-Template mit Objekten verwendet, könnten doch relativ viele Daten zusammenkommen, die dann unnötigerweise auf das Stack gelegt und wieder abgebaut werden müssten. Allerdings sollten Sie diese Empfehlung unter Vorbehalt betrachten. Wenn Sie nämlich Multithreads verwenden sollten, gilt dies nicht mehr. Mehr dazu können Sie in Abschnitt 15.2.1, »Argumente für den Thread«, nachlesen.

In der Praxis dürfte wohl eher selten die Definition der Methode `inline` innerhalb der Klassen-Template-Definition geschrieben werden. Wie Sie eine Methode außerhalb einer Klasse definieren, wissen Sie ja bereits, aber im Fall einer Methode eines Klassen-Templates ist dies etwas spezieller, und daher muss hier gesondert darauf eingegangen werden. Die Syntax für eine Methode eines Klassen-Templates mit formalen Datentypen sieht demnach wie folgt aus:

```
template <typename T>
void Klassenname<T>::methode( parameter ) { ... }
```

Bezogen auf unser Klassen-Template `myContainer<>` sieht diese Definition der Methode außerhalb der Klassen-Template-Definition jetzt wie folgt aus:

```
00 // kapitel11/class_template/myContainer02.cpp
...
01 template <typename T>
02 class myContainer {
03     T data;
04 public:
05     void setData(T& d);
06     T getData() const;
07 };

08 template <typename T>
09 void myContainer<T>::setData(T& d) {
10     data=d;
11 }
```

```

12 template <typename T>
13 T myContainer<T>::getData() const {
14     return data;
15 }
...

```

Es ist wichtig, dass Sie bei Methoden für Klassen-Templates zusätzlich noch den formalen Datentyp in spitzen Klammern (hier <T>) hinter dem Klassennamen angeben, wie hier in den Zeilen 09 und 13 zu sehen ist. Eine Angabe wie `myContainer<T>` entspricht der Angabe eines Datentyps. Ein leeres `myContainer` wäre lediglich ein Template-Name.

Vorläufige Definition

Auch hier gilt wie schon bei den Funktions-Templates, dass die Definition einer Methode auch nur eine vorläufige Definition darstellt, weil der Typ ja immer noch unvollständig ist, da hier ja nur ein formaler Datentyp enthalten ist. Die eigentlichen Methoden werden auch hier erst dann erzeugt, wenn Sie ein Objekt des Klassen-Templates instanziiieren.

Methoden überschreiben (spezialisieren)

Wie auch schon bei den Funktions-Templates (siehe Abschnitt 11.1.3, »Funktions-Template spezialisieren«) lassen sich Methoden von Klassen-Templates bei Bedarf mit `template<>` und dem gewünschten Datentyp überschreiben (spezialisieren). Statt `myContainer<T>` muss hier dann der tatsächliche Datentyp für den formalen Datentyp `T` angegeben werden. Eine solche Spezialisierung für `std::string` für die Methoden `setData()` und `getData()` lässt sich bspw. so realisieren:

```

00 // kapitel11/class_template/myContainer02.cpp
...
01 template <typename T>
02 void myContainer<T>::setData(T& d) {
03     data=d;
04 }

05 template <typename T>
06 T myContainer<T>::getData() const {
07     return data;
08 }

09 template <>
10 void myContainer<std::string>::setData(std::string& str){

```

```

11  std::cout << "Spezialisierung von std::string\n";
12  data = str;
13  }

14  template <>
15  std::string myContainer<std::string>::getData() const {
16      std::cout << "Spezialisierung von std::string\n";
17      return data;
18  }
...

```

In den Zeilen 09 bis 13 und 14 bis 18 finden Sie zwei Spezialisierungen für `myContainer<std::string>`, welche verwendet und erzeugt würden, wenn Sie ein Objekt `myContainer<std::string>` instanziierten würden.

11.2.3 Objekte aus Klassen-Templates erzeugen

Wie auch schon bei den Funktions-Templates werden auch Klassen-Templates erst bei der ersten Verwendung mit dem entsprechenden Typ instanziiert. Das Instanzieren bei Klassen-Templates haben Sie ja im Grunde schon des Öfteren in diesem Buch verwendet, bspw. `std::vector<T>`. Neben dem Klassennamen müssen Sie zwischen den spitzen Klammern den Typ für den formalen Datentyp angeben, von dem der Compiler eine Klasse generieren soll.

Hierzu nun ein paar klassische Anwendungsbeispiele unserer Minimal-Beispielklasse `myContainer<T>`:

```

00  // kapitel11/class_template/myContainer02.cpp
...
01  class Simple {
02      int x;
03  public:
04      Simple(int _x=0) : x(_x) {}
05      int get_x() const {return x;}
06  };
...
07  double dval1 = 123.123;
08  double dval2 = 234.234;
09  myContainer<double> Dcont[3];
10  Dcont[0].setData(dval1);
11  Dcont[1].setData(dval2);
12  std::cout << Dcont[0].getData() << std::endl;
13  std::cout << Dcont[1].getData() << std::endl;

```

```

14 Simple Sval(100000);
15 myContainer<Simple> Scont;
16 Scont.setData(Sval);
17 std::cout << Scont.getData().get_x() << std::endl;

18 std::string str("Text");
19 myContainer<std::string> StrCont;
20 StrCont.setData(str);
21 std::cout << StrCont.getData() << std::endl;
...

```

In Zeile 09 wird der Compiler eine Klasse für `myContainer<double>` generieren und dann drei Objekte `Dcont` als C-Style-Array anlegen. Der Compiler ersetzt hierbei den formalen Datentyp `T` im Klassen-Template durch `double`. In den Zeilen 10 bis 13 übergeben wir unserem Container einige Werte und geben diese wieder aus.

Natürlich funktioniert dies auch mit eigenen Klassen, wie dies mit der Instanziierung in Zeile 15 demonstriert werden soll. Hier wird der Compiler eine Klasse für `myContainer<Simple>` generieren. Der Compiler ersetzt hier den formalen Datentyp `T` durch die Klasse `Simple` (Zeile 01 bis 06). Anschließend wird die Übergabe und Ausgabe eines einzelnen Wertes demonstriert.

Bei der letzten Instanziierung in Zeile 19 wird eine weitere Form der Klasse von unserem Compiler generiert. Hier wird der formale Datentyp durch `std::string` ersetzt. Als Methoden kommen hier in den Zeilen 20 und 21 außerdem die von uns erstellten Spezialisierungen für `std::string` zum Einsatz.

Doppelte Fehlerüberprüfung

Ein weiterer Vorteil beim Instanzieren von Templates im Allgemeinen ist es, dass der Compiler hier zweimal auf Fehler überprüfen muss. Einmal beim Kompilieren der Template-Definition und einmal bei der Generierung von Maschinencode.

11.2.4 Klassen-Templates mit mehreren formalen Datentypen

Wie auch Funktions-Templates können Sie bei Klassen-Templates im Grunde beliebig viele weitere formale Datentypen verwenden. Sie müssen lediglich die formalen Datentypen bei der Definition des Klassen-Templates zwischen den spitzen Klammern, getrennt durch ein Komma, mit dem Schlüsselwort `typename` (bzw. `class`) auflisten. Im folgenden Beispiel sehen Sie eine primitive Version von `std::pair` als eigenes Klassen-Template mit mehreren formalen Datentypen:

```

00 // kapitel11/class_template/myPair01.cpp
...
01 template <typename T, typename U>
02 class myPair {
03     T data01;
04     U data02;
05 public:
06     myPair(const T& t,const U& u) : data01(t), data02(u) {}
07     // ...
08     void print() const;
09 };

10 template <typename T, typename U>
11 void myPair<T, U>::print() const {
12     std::cout << data01 << " : " << data02 << std::endl;
13 }
...
14 std::string month("Januar");
15 int temp = -5;
16 myPair<std::string, int> temperatur(month, temp);
17 temperatur.print();
...

```

Mit diesem Klassen-Template `myPair<T, U>` können Sie eine Klasse erstellen, welche zwei (fast) beliebige Typen verwalten kann. Im Beispiel erzeugen wir eine Instanz mit `std::string` und `int` in Zeile 16. Zunächst veranlassen wir den Compiler, entsprechenden Maschinencode dafür zu generieren, und dann erzeugen wir das Objekt `temperatur`. Zur Demonstration wurde auch eine Methode in den Zeilen 10 bis 13 geschrieben, weil Sie hier, wenn außerhalb der Klasse definiert, immer beide formale Datentypen hinter dem Klassennamen in spitzen Klammern angeben müssen.

Hierzu noch ein Tipp, falls Sie ein Klassen-Template, welches mehrere formale Datentypen enthält, nicht mit bspw. `myPair<double, double>` instanziiieren wollen, wo der Compiler `T` durch `double` und `U` durch `double` ersetzt. Wenn Sie hier eine Instanz der Klasse mit `myPair<double>` generieren wollen, müssen Sie lediglich den Template-Kopf wie folgt anpassen:

```

00 // kapitel11/class_template/myPair02.cpp
...
01 template <typename T, typename U=T>
02 class myPair {
03     T data01;
04     U data02;
05 public:

```

```

06  myPair(const T& t, const U& u) : data01(t), data02(u) {}
...
07  };
...
08  double x=1.5, y=2.5;
09  myPair<double> coordinates(x, y);
10  coordinates.print();
...

```

Hiermit setzen Sie in Zeile **01** `U` auf den Standardwert `T`, falls in den spitzen Klammern nur ein Datentyp geschrieben wurde und das zweite Argument fehlt, wie es hier in Zeile **09** der Fall ist. Durch die Instanziierung in Zeile **09** ersetzt der Compiler den formalen Datentyp `T` durch `double`, und da der zweite Typ für den zweiten formalen Datentyp nicht angegeben wurde, wird hier durch `U=T` das `U` ebenfalls durch `double` ersetzt. Mehr über das Setzen des Standardwertes von Template-Parametern können Sie in Abschnitt 11.2.6, »Klassen-Templates mit Default-Wert«, nachlesen.

Templates mit variabler Argumentanzahl

Neu in C++11 wurde auch die Möglichkeit eingeführt, Templates mit einer variablen Anzahl an Argumenten zu verwenden. Solche Templates werden auch als *Variadic Templates* bezeichnet. Mehr zu dieser neuen Möglichkeit finden Sie in Abschnitt 11.3, »Templates mit variabler Argumentanzahl (C++11)«.

11.2.5 Klassen-Templates mit Non-Type-Parameter

Ein Template-Parameter muss nicht zwangsläufig ein formaler Datentyp sein. Es dürfen auch echte Datentypen verwendet werden (auch Zeiger und Referenzen sind erlaubt). Solche Typen werden auch als Non-Type-Parameter (= Nicht-Typ-Parameter) bezeichnet. Hier ein solcher Non-Type-Parameter:

```

template<typename T, int val>
...

```

Hier sehen Sie mit `int val` einen solchen Non-Type-Parameter, welcher sich wie eine Art Konstante verwenden lässt. Dieser Non-Type-Parameter **muss** allerdings ein ganzzahliger Typ sein. In der Praxis wird diesem Non-Type-Parameter häufig auch noch ein Default-Wert (siehe Abschnitt 11.2.6, »Klassen-Templates mit Default-Wert«) mitgegeben, der verwendet wird, wenn bei der Instanziierung kein Wert für diesen Parameter angegeben wird:

```

template<typename T, int val=10> // mit Default-Wert
...

```

Auch hier wollen wir zur Demonstration des Non-Type-Parameters wieder eine vereinfachte Version des neu in C++11 eingeführten fixen Arrays `std::array` erstellen. Hier der Code dazu mit anschließender Erläuterung:

```
00 // kapitel11/class_template/nonType.cpp
...
01 template <typename T, int n=1>
02 class fixedArray {
03     T data[n]{0}; // C++11
04 public:
05     T& operator[](int index);
06     int mySize() const {return n;}
07     void printAll() const {
08         for( auto it : data ) { // C++11
09             std::cout << it << std::endl;
10         }
11     }
12 };

13 template <typename T, int n>
14 T& fixedArray<T, n>::operator[](int index) {
15     if(index >= n) {
16         std::cout << "Zugriff auf Pos. " << index
17             << " von " << n << std::endl;
18         throw std::out_of_range("fixedArray:Pufferüberlauf");
19     }
20     else if(index < 0) {
21         std::cout << "Zugriff auf Pos. " << index
22             << " von " << n << std::endl;
23         throw
24             std::out_of_range("fixedArray:Pufferunterlauf");
25     }
26     return data[index];
27 }
...
28 try {
29     fixedArray<int, 10> vals;
30     for(int i=0; i < vals.mySize(); ++i) {
31         vals[i] = i+i;
32     }
33     vals.printAll();
34 }
```



```

31   fixedArray<double> dvals;
32   for(int i=0; i < dvals.mySize(); ++i) {
33       dvals[i] = i+i;
34   }
35   dvals.printAll();
36   dvals[5] = 3.1415; // Bereichsüberschreitung !!!
37 }
38 catch(std::out_of_range& e) {
39     std::cerr << e.what() << std::endl;
40 }
...

```

In Zeile 01 finden Sie neben dem formalen Datentyp `T` den Non-Type-Parameter `n`, welcher hier auch gleich den Default-Wert 1 erhält, weil ein Array mit 0 keinen Sinn macht und ohnehin nicht zulässig ist. In Zeile 03 wird dieser Non-Type-Parameter dann verwendet, um ein Array mit `n` Elementen zu erzeugen. Erzeugt wird das Klassen-Template hier auch erst, wenn es aufgerufen wird. In Zeile 26 wird bspw. der formale Datentyp durch `int` und der Non-Type-Parameter `n` durch 10 ersetzt. Der Compiler instanziiert somit daraus quasi ein `int`-Array mit zehn Elementen. Allerdings bietet unser fixes Array mehr als ein herkömmliches C-Array.

So liefert die Methode `mySize()` die Größe des Arrays (was immer der Non-Type-Parameter ist) zurück. Zusätzlich gibt die Methode `printAll()` alle Elemente auf dem Bildschirm aus. Und das Beste ist, dass unser Array eine Bereichsüber- bzw. Unterschreitung erkennt, weil wir hier den `operator[]` überladen haben, welcher den Bereich, auf den zugegriffen wird, überprüft und gegebenenfalls die Ausnahme `std::out_of_range` auswirft. Wichtig ist auch bei dem `operator[]` und den Methoden, die Sie außerhalb des Klassen-Templates definieren, dass Sie auch den Non-Type-Parameter neben dem formalen Datentyp in den spitzen Klammern mit angeben müssen, wie Sie in Zeile 13 sehen.

In Zeile 31 wird der formale Datentyp `T` durch `double` ersetzt. Da für den non-Type-Parameter keine weiteren Angaben in den spitzen Klammern gemacht wurden und wir einen Default-Wert mit 1 gesetzt haben, wird ein `double`-Array mit einem Element instanziiert. In Zeile 36 machen wir mit Absicht eine Bereichsüberschreitung, was von `catch` in den Zeilen 38 bis 40 abgefangen wird.

11.2.6 Klassen-Templates mit Default-Wert

Wie Sie bereits gesehen haben, können Sie Template-Parameter, wie Funktionsparameter, mit einem Default-Wert (Standardwert) versehen. Wenn bei der Instanziierung eines Templates dann das entsprechende Argument fehlt, wird der Default-Wert dafür verwendet. Die Default-Werte werden gewöhnlich bei der Definition des

Templates angegeben. Default-Werte bei Templates betreffen allerdings nicht nur literale Werte, sondern auch formale Datentypen können gleich mit einem Default-Typ vorbelegt werden, der verwendet wird, wenn bei der Instanziierung kein Typ angegeben wird.

Default-Werte auch für Funktions-Templates

Default-Werte können auch bei Funktions-Templates verwendet werden.

Um nochmals auf das Klassen-Template `fixedArray` zurückzukommen, so könnten Sie bspw. Folgendes verwenden:

```
01 template <typename T=int, int n=10>
02 class fixedArray {
...
03 };
```

Hier verwenden wir für den formalen Datentyp `T` den Default-Wert `int` und für den Non-Type-Parameter `n` den Default-Wert `10`. Diese Werte werden immer dann verwendet, wenn der Anwender des Klassen-Templates bei der Instanziierung keine Angaben macht. Hier gelten jedoch dieselben Regeln wie schon bei den Default-Werten von Funktionen (siehe Abschnitt 5.5.6, »Default-Parameter (Standardparameter)«). Wenn also ein Parameter einen Default-Wert bekommt, müssen alle anderen rechts nachfolgenden Parameter ebenfalls einen Default-Wert bekommen. Hier bedeutet dies konkret, wenn der formale Parameter `T` den Default-Wert `int` bekommen hat, muss der Non-Type-Parameter daneben auch einen Default-Wert bekommen.

Von dem so definierten Klassen-Template `fixedArray` mit den beiden Default-Werten lässt sich jetzt auf drei verschiedene Arten eine Klasse instanziiieren:

```
01 fixedArray<> vals;           //fixedArray<int, 10>
02 fixedArray<double> dvals;    //fixedArray<double, 10>
03 fixedArray<char,8> bytes;    //fixedArray<char, 8>
04 fixedArray<100> whatever;    // Fehler !!!
```

In Zeile **01** wird der Compiler eine Klasse mit den Default-Werten generieren, welche ein `int`-Array mit zehn Werten als Eigenschaft aufnehmen kann. In Zeile **02** hingegen wird der formale Datentyp `T` vom Klassen-Template durch `double` ersetzt, und der Compiler generiert hier ein `double`-Array mit zehn (Default-Wert-)Elementen als Eigenschaft. In Zeile **03** hingegen verwenden wir einen Typ für den formalen Datentyp und auch einen Wert für den Non-Type-Parameter, wodurch der Compiler eine Klasse generiert, welche ein `char`-Array mit acht Elementen als Eigenschaft speichern kann.

Alias-Template (C++11)

Eine Instanziierung wie in Zeile **04** vom Beispiel zuvor mit `fixedArray<100>` ist nicht möglich, weil, wenn bei einer Instanziierung ein Argument weggelassen wird, so müssen alle nachfolgenden Argumente weggelassen werden. Es gibt trotzdem eine Möglichkeit, mithilfe von `using` solche Werte in spitzen Klammern zu verwenden:

```
01 template <int MAX>
02 using ifixedArray = fixedArray<int, MAX>;    // C++11
...
03 ifixedArray<100> bigArray01;
...
```

In den Zeilen **01** und **02** finden Sie ein sogenanntes Alias-Template, was neu mit C++11 eingeführt wurde. Hiermit können Sie jetzt, wie in Zeile **03** zu sehen ist, eine Klasse für ein `int`-Array als Eigenschaft generieren lassen, dessen Anzahl Sie beim Aufruf, wie hier mit 100, zwischen den spitzen Klammern festlegen können. Hierbei handelt es sich um eine echte Template-Technik, welche hier in Zeile **01** mit dem Non-Type-Parameter für `MAX` eingeleitet wurde und in Zeile **02** verwendet wird, um ein `fixedArray` vom Typ `int` mit `MAX` Elementen zu generieren. Dieses Klassen-Template kann jetzt mit dem Aliasnamen `ifixedArray` verwendet werden (siehe Zeile **03**). Wohlgermerkt, hiermit wird nach wie vor das Klassen-Template `fixedArray<T, n>` verwendet. Nur erfolgt der Aufruf mit einem Aliasnamen, um die Verwendung zu erleichtern.

Formale Datentypen

Für solche Alias-Templates lassen sich auch formale Datentypen verwenden.

Aliasnamen

Auf ähnliche Weise können Sie auch ein Alias mit `using` erstellen, um beim Aufruf zur Generierung einer Klasse gar keine spitzen Klammern mehr anzugeben:

```
01 using i10fixedArray = fixedArray<>;          // C++11
02 using d10fixedArray = fixedArray<double>;    // C++11
...
03 i10fixedArray intArray;    //fixedArray<int,10>
04 d10fixedArray doubleArray; //fixedArray<double,10>
...
```

Hier verwenden Sie `using` in den Zeilen **01** und **02**, um einen Aliasnamen für `fixedArray<int, 10>` und `fixedArray<double, 10>` verwenden zu können, was Sie dann in den Zeilen **03** und **04** auch im Einsatz sehen. Solche einfachen Aliasnamen können

Sie aber auch noch mit der alten C++03-Technik mit Hilfe von `typedef` realisieren. Alias-Templates allerdings nicht mehr.

Standardbibliothek

Auch die Standardbibliothek erleichtert Ihnen das Leben mit solchen Aliasnamen. So verdanken Sie es einem `typedef`, dass Sie `std::string` nicht mit `std::basic_string<char>` verwenden müssen, weil hier folgender `typedef` benutzt wurde:

```
typedef basic_string<char> string;
```

11.2.7 Klassen-Templates spezialisieren

Wie auch schon mit Funktions-Templates können Sie Klassen-Templates spezialisieren, wenn ein Datentyp nicht so recht mit einem Klassen-Template funktionieren will oder kein vernünftiges Ergebnis zurückliefert. Im Gegensatz zu Funktions-Templates können Sie Klassen-Templates sogar nur teilweise spezialisieren (partielle Spezialisierung).

Klassen-Template partiell spezialisieren

Um das Thema hier nicht zu umfangreich werden zu lassen, soll an dieser Stelle nochmals das Klassen-Template `myPair` aus Abschnitt 11.2.4, »Klassen-Templates mit mehreren formalen Datentypen«, hierfür verwendet werden. Und zwar soll hierbei das Klassen-Template partiell spezialisiert werden, wenn beim Aufruf einer der beiden formalen Datentypen `std::string` enthält.

Folgende Aufrufe sollten also partiell spezialisiert werden:

```
01 myPair<int, std::string> data01(86153, "Augsburg");
02 myPair<std::string, int> data02("Friedberg", 86163);
```

Und so kann die partielle Spezialisierung dazu aussehen:

```
00 // kapitel11/class_template/specialization.cpp
...
01 template <typename T, typename U=T>
02 class myPair {
03     T data01;
04     U data02;
05 public:
06     myPair(const T& t, const U& u) : data01(t), data02(u) {}
...
07 };
```

```

08 template <typename T>
09 class myPair<T, std::string> {
10     T data01;
11     std::string data02;
12 public:
13     myPair(const T& t, const std::string& str) :
14         data01(t), data02(str) {
15         std::cout << "myPair<T, std::string>\n";
16     }
17 ...
18 };

17 template <typename T>
18 class myPair<std::string, T> {
19     std::string data01;
20     T data02;
21 public:
22     myPair(const std::string& str, const T& u) :
23         data01(str), data02(u) {
24         std::cout << "myPair<std::string, T>\n";
25     }
26 ...
27 };

...
26 myPair<int, std::string> data01(86153, "Augsburg");
27 myPair<std::string, int> data02("Friedberg", 86163);
...

```

In den Zeilen **01** bis **07** sehen Sie unser ursprüngliches Klassen-Template mit zwei formalen Datentypen. In den Zeilen **08** bis **16** finden Sie die partielle Spezialisierung, die der Compiler verwendet, wenn das Klassen-Template mit `myPair<T, std::string>` aufgerufen und daraus eine Klasse instanziiert werden soll, wie dies in Zeile **26** mit `myPair<int, std::string>` der Fall wäre.

Für die partielle Spezialisierung muss im Klassen-Template natürlich der formale Datentyp (hier wieder `T`) zwischen die spitzen Klammern (Zeile **08**) gestellt werden, der nicht spezialisiert wird und erst beim Aufruf seine Typen erhält. Nach dem Klassennamen (Zeile **09**) werden jetzt in den spitzen Klammern alle Template-Argumente (hier `T`) und die Datentypen (hier `std::string`) aufgelistet, für die diese partielle Spezialisierung gelten soll.

Dass es auch in einer anderen Reihenfolge möglich ist, soll die partielle Spezialisierung in den Zeilen **17** bis **25** zeigen, welche der Compiler für die Instanziierung verwendet, wenn `myPair<std::string, int>` aufgerufen wird, wie dies in Zeile **27** der Fall ist.

Unser Beispiel hat aber einen Schönheitsfehler, wenn Sie das Klassen-Template mit `myPair<std::string, std::string>` aufrufen wollen. Der Compiler kann jetzt keine Instanziierung machen, weil er nicht weiß, ob er `myPair<T, std::string>` oder `myPair<std::string, T>` verwenden soll. Gäbe es hier nur eine der beiden partiellen Spezialisierungen, so wäre das Problem gelöst, und es würde jetzt die eine partielle Spezialisierung verwendet werden. Würde dann die andere partielle Spezialisierung aufgerufen, würde der Compiler eine Instanz aus `myPair<T, U>` generieren.

Um im vorliegenden Fall keine der partiellen Spezialisierungen streichen zu müssen, würde sich eine vollständige Spezialisierung von `myPair<std::string, std::string>` anbieten.

Was kann nicht partiell spezialisiert werden?

Im Gegensatz zu einem Klassen-Template ist es nicht möglich, ein Funktions-Template partiell zu spezialisieren. Auch eine Methode eines Klassen-Templates kann nicht partiell, sondern muss vollständig spezialisiert werden.

Klassen-Template voll spezialisieren

Wenn bei Klassen-Templates keines der Template-Argumente so recht passen will, dann können Sie auch eine vollständige Spezialisierung des Klassen-Templates erstellen. Im Beispiel zuvor war ja die Rede von einer vollständigen Spezialisierung von `myPair<std::string, std::string>`. Realisieren können Sie eine vollständige Spezialisierung wie folgt:

```
00 // kapitel11/class_template/specialization.cpp
...
01 template <>
02 class myPair<std::string, std::string> {
03     std::string data01;
04     std::string data02;
05 public:
06     myPair(const std::string& t, const std::string& str) :
07         data01(t), data02(str) {
08         std::cout << "myPair<std::string, std::string>\n";
09     }
...
09 };
...
10 myPair<std::string, std::string> data04("Augsburg", "Friedberg");
```

Zum vollständigen Spezialisieren eines Klassen-Templates bleibt die spitze Klammer, wie auch bei den Funktions-Templates, leer (siehe Zeile 01). Der oder (wie hier)

die Datentypen, für welche das Klassen-Template jetzt spezialisiert werden soll, muss nach dem Klassennamen in den spitzen Klammern geschrieben werden, wie hier in Zeile 02 mit `<std::string, std::string>` zu sehen ist. Durch die vollständige Spezialisierung des Klassen-Templates `myPair<T, U>` durch `myPair<std::string, std::string>` wird jetzt nach dem Aufruf in Zeile 10 vom Compiler die Spezialisierung der Zeilen 01 bis 09 für die Generierung verwendet.

11.2.8 Klassen-Template als Parameter an Funktion/Methode

Ein Thema habe ich bewusst bei den Funktions-Templates noch unterschlagen. Und zwar geht es um die Möglichkeit, ein Klassen-Template als Objekt einer Funktion bzw. Methode zu übergeben. Da der Compiler erst beim Aufruf eine entsprechende Klasse aus einem Klassen-Template generiert, muss logischerweise auch der Funktionsparameter ein Template-Argument sein. Da der Datentyp des Funktionsparameters wegen des formalen Datentyps des Klassen-Templates unterschiedlich sein kann, muss eine Funktion auch ein Funktions-Template sein.

Im Grunde ist die Definition dieses Funktions-Templates relativ einfach. Gewöhnlich verwendet man auch hier eine Referenz (bzw. auch `const`-Referenz) vom Typ des Klassen-Templates. Hierzu ein kurzer Codeausschnitt, der dies demonstrieren soll:

```
00 // kapitel11/class_template/func_class_param01.cpp
...
01 template <typename T>
02 class myContainer {
03     T data;
04 public:
05     myContainer(const T& t) : data(t) {};
06     void print() { std::cout << data << std::endl;}
07 };

08 template <typename T>
09 void function(myContainer<T>& myObj) {
...
10     myObj.print();
11 }
...
12     std::string str("Ein Text");
13     myContainer<std::string> data01(str);
14     function(data01);
...
```

In Zeile 08 bis 11 sehen Sie die Funktion, welche als Parameter ein Objekt des Klassen-Templates `myContainer<T>` erwartet. Das Template-Argument `T` entspricht hier dann dem Datentyp, mit dem Sie das Objekt des Klassen-Templates instanziiert haben. In Zeile 13 wird bspw. eine Klasse mit `myContainer<std::string>` instanziiert. Somit generiert auch der Compiler eine Funktion mit `function(myContainer<std::string>&)` für uns, dank des Funktionsaufrufes in Zeile 14.

11.3 Templates mit variabler Argumentanzahl (C++11)

Neu in C++11 ist hinzugekommen, Templates mit beliebig vielen Argumenten zu erstellen. Diese neue Technik wird häufig auch als *Variadic Templates* bezeichnet. Die Syntax dieser neuen Technik sieht allerdings zunächst etwas ungewöhnlich aus, weil hier in der Typliste die drei Punkte ... (auch Ellipse genannt) verwendet werden.

Anwendungsgebiet

In der Praxis dürfte wohl die Implementierung von solchen *Variadic Templates* ganz besonders für diejenigen interessant sein, die sich mit dem Erstellen von Bibliotheken beschäftigen. Wer jetzt anschließend allerdings hergeht und hiermit ein Klassen-Template erstellen will, welches eine variable Anzahl an Typen aufnehmen kann, dem möchte ich vorher noch den sequenziellen Datentyp `std::tuple` (siehe Abschnitt 12.6.2, »Tupel, das bessere pair (C++11)«) ans Herz legen, der ebenfalls mit C++11 eingeführt wurde, und auch meistens die Lösung sein sollte, nach der Sie suchen.

Hierbei werden mit ... die Parameter zu einem *Parameter-Pack* zusammengepackt und auch wieder entpackt. Dies hängt davon ab, auf welcher Seite der Operator ... steht. Das Ganze in der Theorie zu lesen, hilft hier nicht weiter, daher soll Ihnen ein Listing den Sachverhalt demonstrieren:

```
00 // kapitel11/variadic_template/variadic01.cpp
...
01 #include <typeinfo>          // operator typeid
02 #include <unordered_map>     // std::unordered_map
03 std::unordered_map <std::size_t, std::string> type_names;

04 template <typename T>
05 void output(T val) {
06     std::cout << type_names[typeid(val).hash_code()] << ":";
07     std::cout << val << std::endl;
08 }
```



```

09 template<typename First, typename ... Rest>
10 void output(First first, Rest ... rest) {
11     output(first);
12     output(rest ...);
13 }
...
14 type_names[typeid(const char*).hash_code()]="const char*";
15 type_names[typeid(char).hash_code()] = "char";
16 type_names[typeid(int).hash_code()] = "int";
17 type_names[typeid(double).hash_code()] = "double";
18 type_names[typeid(bool).hash_code()] = "bool";

19 output("ende", 2, 3.14, 'A', false);
20 output(3.1415);
...
```

Die Verwendung von `type_names` in den Zeilen **03**, **06** und **14** bis **18** können Sie ignorieren, sie dient nur der Informationsausgabe, um welchen Typ es sich hierbei handelt. Hierfür wurde die neu in C++11 eingeführte Funktion `hash_code()` aus `<typeinfo>` verwendet.

Betrachten Sie zunächst den Funktionsaufruf von `output()` in Zeile **19** mit beliebigen Parametern. Die Funktion wurde überladen und ist daher in doppelter Ausführung vorhanden. Aber dazu gleich mehr. Mit dem Funktionsaufruf wird das *Variadic Template* in den Zeilen **09** bis **13** aufgerufen.

Zuerst müssen wir die übergebene Liste in zwei Teilen zwischen dem ersten und den restlichen Elementen aufteilen. In Zeile **09** haben Sie den Typ `First` für das erste Element und `Rest` als Parameter-Pack für die restlichen Elemente festgelegt. Beachten Sie hier die Position des `...`-Operators, der links von `Rest` steht, was bedeutet, dass der Operator die einzelnen Parameter zu einem Parameter-Pack **verpackt** hat.

Im Funktionskopf von Zeile **10** finden Sie jetzt die einzelnen Parameter. In `First first` haben Sie immer jeweils das erste Element in der Liste, und in `Rest ... rest` finden Sie die restlichen Elemente. Hier sehen Sie auch gleich, dass jetzt der `...`-Operator rechts von `Rest` steht, und dies bedeutet, dass hier das Parameter-Pack wieder **entpackt** wird. In `first` findet sich jetzt das erste Element in der Liste, welches hier als `const char*` ermittelt wird und in Zeile **11** die überladene Version `output()` in den Zeilen **04** bis **08** mit dem C-String "ende" aufruft. Die überladene Version sollte hierbei ein Funktions-Template oder eine passende Überladung sein, damit der Compiler eine entsprechende Version generieren kann. In der Funktion geben wir in Zeile **06** den Typ und in Zeile **07** den Inhalt aus, ehe die Funktion wieder mit dem Aufrufer in Zeile **12** fortfährt.

In Zeile 12 starten Sie jetzt einen rekursiven Aufruf (Rekursion = Selbstaufruf), indem Sie das restliche Parameter-Pack ohne `first` erneut für die Funktion in den Zeilen 09 bis 13 aufrufen. Der Trick ist dabei, dass `rest` bei jedem erneuten Selbstaufruf um das erste Element `first` gekürzt wird. Der erneute Funktionsaufruf des Parameter-Packs würde somit wie folgt aussehen:

```
output(2, 3.14, 'A', false);
```

Jetzt beginnt das Spielchen wieder von vorne, nur dass jetzt 2 als `int`-Wert für `first` ermittelt wird, und der Rest als Parameter-Pack wäre `<double, char, bool>`. In jedem weiteren Selbstaufruf passiert jetzt dasselbe und die Funktion wird immer ohne das Element `first` erneut aufgerufen:

```
output(3.14, 'A', false);
output('A', false);
output(false);
```

Die Rekursion wird beendet, wenn `rest` nur noch einen Parameter enthält. In dem Fall wird nur noch das Funktions-Template aufgerufen. Ebenfalls nur das Funktions-Template wird aufgerufen, wenn Sie die Funktion `output()` mit nur mit einem Parameter aufrufen, wie dies in Zeile 20 mit dem `double`-Wert demonstriert wurde.

Schleifen nicht möglich

Die Frage werden Sie sich sicherlich hier gestellt haben, warum ich hier keine Schleifen wie `for` oder `while` verwendet haben? Das ist nicht möglich, weil das Parameter-Pack zur Übersetzungszeit behandelt werden muss.

Das Programm bei der Ausführung:

```
const char* : ende
int : 2
double : 3.14
char : A
bool : 0
double : 3.1415
```

operator<< vorhanden?

Dass unser Beispiel hier so glatt funktioniert, liegt auch daran, dass `operator<<` für alle Typen, die wir verwendet haben, definiert war. Ist dies nicht der Fall, müssen Sie natürlich den `operator<<` in Ihrer Klasse definieren. Ansonsten würde sich der Compiler bei Ihnen beschweren.

sizeof...-Operator

Einen Operator gibt es dann mit `sizeof...()` für Parameter-Packs doch noch. Mit diesem Operator können Sie zur Übersetzungszeit die Anzahl der vorhandenen Elemente des Parameter-Packs herausfinden. Hier der Operator im Einsatz:

```
00 // kapitel11/variadic_template/variadic02.cpp
...
01 template <typename ... Args>
02 auto countArgs(Args ... args) -> int {
03     return (sizeof ... (args) );
04 }
...
05 std::cout << countArgs("eins", 2, 3.14) << std::endl;
```

Parameter-Pack in Tupel konvertieren

Früher oder später werden Sie auf den neuen sequenziellen Typ `std::tuple` stoßen und dessen komfortablen Zugriff auf die einzelnen Elemente mit `get<N>` zu schätzen wissen. Schließlich ist `std::tuple` auch das prominenteste Beispiel für den Einsatz von *Variadic Templates*. Auf `std::tuple` wird noch in Abschnitt 12.6.2, »Tupel, das bessere pair (C++11)«, etwas umfassender eingegangen. Wenn Sie bspw. ein Parameter-Pack haben und daraus ein `std::tuple` machen wollen, so ist dies im Grunde mit `std::make_tuple` keine große Sache. Folgender Codeausschnitt demonstriert Ihnen, wie Sie aus einem Parameter-Pack ein Tupel machen können:

```
00 // kapitel11/variadic_template/variadic03.cpp
...
01 #include <tuple>
...
02 template <typename ... Args>
03 auto conv2tuple(Args ... args) ->
    decltype(std::make_tuple(args...)) {
04     return std::make_tuple(args...);
05 }
...
06 auto myTuple = conv2tuple("ende", 2, 3.14, 'A', false);
07 std::cout << std::get<0>(myTuple) << std::endl;
08 std::cout << std::get<1>(myTuple) << std::endl;
09 std::cout << std::get<2>(myTuple) << std::endl;
10 std::cout << std::get<3>(myTuple) << std::endl;
11 std::cout << std::get<4>(myTuple) << std::endl;
```

Das Prinzip ist einfach. In Zeile **06** rufen Sie die Funktion `conv2tuple()` mit verschiedenen Parametern auf, wo diese zu einem Parameter-Pack verpackt werden. In der

Funktion wird dieses Paket wieder entpackt, wobei wir hier auch gleich den Rückgabewert mit `decltype` vom Compiler ermitteln lassen. In Zeile **04** machen wir aus der Liste von Parametern mithilfe von `std::make_tuple()` ein `std::tuple` und weisen den Rückgabewert in Zeile **06** `myTuple` zu. In den Zeilen **07** bis **11** verwenden wir die Funktion `get<>`, um die Elemente des Tupels auszugeben – und nein, auch bei einem Tupel kann nicht mit Schleifen wie `for` oder `while` durchlaufen werden, weil es sich letztendlich auch wieder um *Variadic Templates* handelt.

»Variadische« Klassen-Templates oder doch lieber ein Tupel?

In der Tat ist es durchaus möglich, »variadische« Klassen-Templates zu generieren. Hierfür müssen Sie allerdings die Klassen rekursiv implementieren, was ziemlich komplex werden kann. Hier bietet das neue `std::tuple`, welches ebenfalls in C++11 eingeführt wurde, viel mehr Flexibilität und Komfort. Wer sich trotzdem dafür interessiert, der kann sich ja mal dieses Beispiel auf der folgenden Webseite ansehen:

<http://thenewcpp.wordpress.com/2012/02/15/variadic-templates-part-3-or-how-i-wrote-a-variant-class>

Kapitel 12

Container, Iteratoren, Algorithmen und Hilfsmittel

Maßgeblich für den großen Erfolg von C++ sind die zahlreichen Bibliotheken, die einem das Leben erleichtern und wegen denen man das Rad nicht immer wieder neu erfinden muss, sondern gleich langzeiterprobte Komponenten für seine Projekte einsetzen kann. Da die Standardbibliothek von C++ riesig ist, wird in diesem Kapitel gesondert auf das Konzept der verschiedenen Standardcontainer-Klassen (Behälterklassen) eingegangen.

Die Basis zu den Standardcontainer-Klassen und den Algorithmen, wie Sie heute in der Standardbibliothek von C++ enthalten sind, stammt aus der Standard Template Library (kurz STL), welche in den 1980er Jahren bei Hewlett-Packard mit dem Schwerpunkt auf Datenstrukturen und Algorithmen entwickelt wurde, basierend auf generischer Programmierung (also Klassen- und Funktions-Templates). Heute wird allerdings der Begriff STL nicht mehr verwendet und kommt auch in der Standardbibliothek gar nicht mehr namentlich vor.

Die Verwendung und Erstellung von Templates haben Sie bereits in Kapitel 11 kennen und (hoffentlich) schätzen gelernt. Mithilfe dieser Technik wurden auch die Standardcontainer-Klassen und Algorithmen entwickelt. Die Verwendung solcher fertigen Templates kann das Programmiererleben enorm erleichtern. Einige Vorteile, solche vorgefertigten Templates zu verwenden, sind folgende:

- ▶ stabiles und besseres Laufzeitverhalten der Programme, weil die Templates bereits jahrelang erprobt, verbessert und wohl frei von Fehlern sind
- ▶ Es handelt sich um eine standardisierte Bibliothek, und somit sind die Programme leichter portierbar, weil sich jeder namenhafte Compilerhersteller am Standard orientiert.
- ▶ Auch bei Problemen findet man relativ schnell Hilfe oder Lösungsvorschläge, weil es sich hier nicht um eine spezielle, sondern um eine weitverbreitete Bibliothek handelt und somit das Wissen darüber auch viel weiter verbreitet ist.

12.1 Grundlagen

An dieser Stelle will ich Ihnen kurz das Konzept oder besser das harmonische Zusammenspiel von Containerklassen (oft auch Behälterklassen genannt), Iteratoren und Algorithmen erklären.

Standardcontainer-Klassen (Behälterklassen)

Häufig benötigt man in Programmen Objekte mit mehreren gleichartigen Typen. Dies können einfache Ganzzahl- oder Gleitkommatypen sein, aber auch komplexe Objekte. Zur Verwaltung solcher Daten kennt man in der Informatik verschiedene Datenstrukturen wie Vektoren, Listen, Stapel, Bäume usw. Die Standardbibliothek bietet hierfür sogenannte (Standard-)Containerklassen (bspw. `std::vector`, `std::list` usw.) an, die ebenfalls als Klassen-Template implementiert sind.

Solche Container sind Objekte, welche andere Datentypen und Objekte speichern können. Der Container kümmert sich auch um die Speicherverwaltung für die Elemente. Für den Zugriff auf die einzelnen Elemente stellt die Containerklasse verschiedene Methoden und Iteratoren zur Verfügung. Hierbei wird zwischen *sequenziellen* und *assoziativen Containern* unterschieden. Bei sequenziellen Containern sind die Objekte linear angeordnet, und bei den assoziativen Containern muss mithilfe von Schlüsseln auf die Elemente zugegriffen werden. Mehr zu den verschiedenen Containern der Standardbibliothek erfahren Sie in Abschnitt 12.2, »Standardcontainer-Klassen«.

Iteratoren

Die Iteratoren sind wie Zeiger, intelligente Zeiger, mit denen Sie auf die einzelnen Elemente eines Containers schrittweise zugreifen bzw. sie wiederholt durchlaufen können. Iteratoren sind häufig als gewöhnliche Zeiger oder reine Zugriffsobjekte mit zeigerähnlichen Eigenschaften implementiert. Es gibt unterschiedliche Kategorien von Iteratoren, wobei nicht jeder Container alle Iteratoren zur Verfügung stellt. So fehlt dem `std::list`-Container sinngemäß der wahlfreie Zugriff (bspw. mithilfe der Methode `at()`), welchen der `std::vector`-Container implementiert hat. Mehr zu den verschiedenen Iteratoren finden Sie in Abschnitt 12.4, »Iteratoren«.

Algorithmen

(Template-)Algorithmen wiederum sind spezielle Funktionen, um mit Iteratoren auf Container zuzugreifen. Die Standardbibliothek bietet hierfür eine Menge nützlicher Algorithmen zum Suchen, Kopieren, Vergleichen, Sortieren usw. an. Die Algorithmen wurden allerdings nicht abhängig von speziellen Containern als Methode implementiert, sondern als globales Funktions-Template und können somit völlig unabhängig von Containern und Iteratoren auch für eingebaute Datentypen, eigene

Klassen(-Templates) und gewöhnliche Zeiger verwendet werden. Die Algorithmen werden in Abschnitt 12.5, »Algorithmen«, nochmals genauer behandelt.

Ohne Container, Iteratoren und Algorithmen

Zunächst will ich Ihnen das Zusammenspiel von Container, Iterator und Algorithmus auf niedrigster Ebene mit rohen Zeigern demonstrieren, einem Code, den man so heute nicht mehr in C++ schreibt und der eher in die Klamottenkiste gehört. Hierzu das Beispiel, mit einer anschließenden Erläuterung:

```
00 // kapitel12/cont_iter_algo01.cpp
...
01 int *find(int *begin, int *end, const int& val) {
02     for(int *it=begin; it!=end; ++it) {
03         if( *it == val) {
04             return it;
05         }
06     }
07     return end;
08 }
...
09 const int MAX=20;
10 int ival=1;
11 int Container[MAX]={0};
12 int *begin = Container;
13 int *end = Container + MAX;
14 for(int *it=begin; it!=end; ++it)
15     *it = ival++;

16 int search_val=-1;
17 std::cout << "Welche Nummer suchen Sie: ";
18 std::cin >> search_val;

19 int *pos = find(begin, end, search_val);
20 if(pos == end ) {
21     std::cout << "Wert nicht vorhanden\n";
22 }
23 else {
24     std::cout << "Wert gefunden an Pos.: "
                << (pos-begin) << '\n';
25 }
```

Ich will hier gar nicht allzu viele Worte zum Code verlieren, weil dieser keine Schule machen soll. In Zeile 11 verwenden wir ein rohes C-Array vom Typ `int` als **Container**.

Als **Iteratoren** auf das erste und hinter das letzte Element in diesem Behälter verwenden wir in den Zeilen 12 und 13 zwei rohe Zeiger und weisen dort jeweils die Adresse des ersten Elements bzw. der Position vor dem letztem Element zu. In den Zeilen 14 und 15 initialisieren wir den Container mit Werten von 1–20. In den Zeilen 16 bis 25 können Sie nach einem bestimmten Wert im Container suchen. Für die Suche habe wir eine globale Funktion namens `find()` geschrieben (Zeilen 01 bis 08), die hier auch gleich unser **Algorithmus** ist. Den Algorithmus rufen wir in Zeile 19 mit der Anfangsadresse und der Adresse vom letzten Element im Container mit dem gesuchten Element als Parameter auf. Im Algorithmus selbst wird der Container mit den Zeigern Element für Element durchlaufen, und es wird überprüft, ob der Wert gefunden wurde. Wurde kein Wert gefunden, wird `end` zurückgegeben – ein Code für Nostalgiker eben.

Standardalgorithmus implementieren

Hier verwenden wir also rohe Datenstrukturen als Container, Zeiger als Iteratoren und eine globale Funktion als Algorithmus, in Zeiten der generischen Programmierung nicht sehr elegant. Wenn wir schon bei diesem Beispiel sind, kann hier gleich die Aussage weiter oben bewiesen werden, dass Algorithmen völlig unabhängig implementiert werden können. Mit Absicht wurde hier auch gleich die globale Funktion `find()` geschrieben, wovon es in der Headerdatei `<algorithm>` auch tatsächlich ein gleichnamiges Funktions-Template `std::find()` mit denselben Parametern gibt. Im Beispiel müssen Sie daher nur die Headerdatei `<algorithm>` einbinden und das `std::` vor `find()` schreiben, dann können Sie die globale Funktion `find()` entfernen:

```
00 // kapitel12/cont_iter_algo02.cpp
01 #include <algorithm>
...
19 int *pos = std::find(begin, end, search_val);
```

Mehr war nicht nötig. Jetzt wird der echte Algorithmus `std::find()` für unsere Suche nach dem Element verwendet. Dies ist ohne großen Aufwand möglich, weil die Algorithmen als Funktions-Template in der Standardbibliothek implementiert wurden.

Container, Iteratoren und Algorithmen mit der Standardbibliothek

Das Beispiel in Listing `cont_iter_algo01.cpp` wurde speziell für den Typ `int` geschrieben und ist daher nicht sehr flexibel. In Kapitel 11 haben Sie erfahren, wie Sie eigene Templates erstellen können und somit für jeden Typ gerüstet sind. Sie könnten jetzt hergehen und ein eigenes Klassen-Template für den Container erstellen, ein weiteres Klassen-Template für die Iteratoren und ein Funktions-Template für den Algorithmus. Das wäre sicherlich eine gute Übung, aber im Grunde würden Sie das Rad neu erfinden. Für solche Dinge gibt es und verwendet man in der Regel meistens auch die Standardbibliothek. Dasselbe Beispiel aus Listing `cont_iter_algo01.cpp` soll jetzt mithilfe der Standardbibliothek implementiert werden:

```

00 // kapitel12/cont_iter_algo03.cpp
01 #include <vector>
02 #include <algorithm>
...
03 const int MAX=20;
04 int ival=1;
05 std::vector<int> Container(MAX);

06 for( std::vector<int>::iterator it=Container.begin();
      it != Container.end(); ++it ) {
07     *it = ival++;
08 }

09 int search_val=-1;
10 std::cout << "Welche Nummer suchen Sie: ";
11 std::cin >> search_val;

12 std::vector<int>::iterator pos =
    std::find(Container.begin(),Container.end(),search_val);
13 if(pos == Container.end() ) {
14     std::cout << "Wert nicht vorhanden\n";
15 }
16 else {
17     std::cout << "Wert gefunden an Pos.: "
        << (pos-Container.begin()) << '\n';
18 }

```

Hier finden Sie in Zeile **05** den Container `std::vector`, welcher `MAX-int`-Elemente verwalten kann. Die Iteratoren auf das erste und vor dem letzten Element sind dabei schon bei der Instanziierung des Containers als Methode mit `begin()` und `end()` dabei, wie Sie dies in der `for`-Schleife in Zeile **06** und beim Aufruf des Algorithmus `std::find()` in Zeile **12** sehen können. Beim Aufruf in Zeile **12** können Sie auch sehr schön sehen, wie hier der Container mit dem Algorithmus zusammenarbeitet und wie in Zeile **17** selbst eine klassische Arithmetik auf den Iteratoren durchgeführt werden kann.

C++11: auto statt `std::vector<int>::iterator`

Der passende Iterortyp wird im Beispiel mit `std::vector<int>::iterator` ermittelt, erzeugt und verwendet. Wem dies zu umständlich ist, der kann stattdessen das neu in C++ eingeführte Schlüsselwort `auto` verwenden. Somit würde sich die Schleife in Zeile **06** wie folgt verkürzen lassen:

```

06 for(auto it=Container.begin();it!=Container.end();++it) {
07     *it = ival++;
08 }

```

Dass hier der Programmieraufwand erheblich vereinfacht und verkürzt wurde, ist schon sehr hilfreich. Noch wichtiger ist allerdings, dass Sie auf die Art viele Programmierfehler vermeiden und den Verwaltungsaufwand minimieren können, als wenn Sie alles selbst ohne die Standardbibliothek geschrieben hätten.

Dass hier alles **in einer Form** als Klassen-Template (Container) und Funktions-Template (Algorithmen) vorliegt und die Iteratoren dazwischen hervorragend mit weiteren Operationen harmonisieren, macht diese Konzept so genial.

Wenn Sie die Abbildung 12.1 betrachten, sind Daten (Container) und Funktionen (Algorithmen) voneinander getrennt. Die Iteratoren sind hier Bindeglieder zwischen Daten und Funktionen. Mithilfe der Iteratoren (sollten) die Algorithmen mit den Containern zusammenarbeiten. Somit entspricht dieses Konzept der OOP, wo Sie ebenfalls gewöhnlich Daten und Funktionen voneinander trennen. Vereinfacht kann man zusammenfassen, dass alles einfach wie in einer Fertigmischung bereitliegt und nur noch in eine Form gegeben werden muss, ohne dass Sie sich noch allzu viele Gedanken um die Zutaten machen müssen.

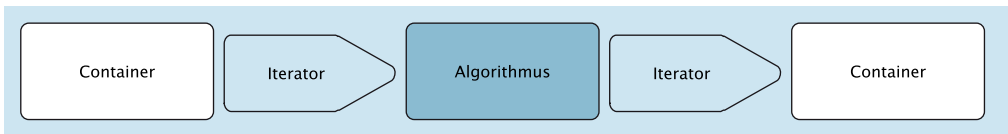


Abbildung 12.1 Das Grundkonzept von Containern, Iteratoren und Algorithmen der Standardbibliothek von C++, bildlich dargestellt.

12.2 Standardcontainer-Klassen

Wie bereits erwähnt, bietet die Standardbibliothek für fast jede Datenstruktur eine passende Containerklasse an. Aufteilen lassen sich diese in **sequenzielle** und **assoziative** Containerklassen. Da jede dieser Containerklassen als Klassen-Template realisiert wurde, können sie mit Objekten beliebiger Typen verwendet werden. Auch um die Speicherverwaltung müssen Sie sich nicht mehr kümmern.

12.2.1 Sequenzielle Containerklassen

Sequenzielle Containerklassen sind Container, bei denen die einzelnen Elemente linear angeordnet und gespeichert werden. Die Standardbibliothek bietet hierzu mitt-

lerweile fünf verschiedene Arten von Containerklassen an, nämlich `std::vector`, `std::list`, `std::deque` und – seit C++11 mit dabei – `std::array` und `std::forward_list`.

Welchen Container wann verwenden?

Welchen Container Sie jetzt für welchen Zweck verwenden können oder sollen, hängt davon ab, wie Sie dem Container neue Elemente hinzufügen bzw. sie entfernen wollen und dem daraus resultierenden Laufzeitverhalten. So wird in der Praxis eine verkettete Liste mit `std::list` eingesetzt, wenn viele neue Elemente irgendwo eingefügt bzw. gelöscht werden. Werden immer nur Elemente am Ende hinzugefügt oder benötigen Sie einen schnellen wahlfreien Zugriff, dann ist `std::vector` die erste Wahl. Wenn Elemente vorwiegend am Anfang oder am Ende hinzugefügt werden, dann bietet sich `std::deque` an, womit ebenfalls ein wahlfreier Zugriff wie mit `std::vector` möglich ist.

`std::deque` vs. `std::vector`

`std::vector` ist ideal, wenn Sie sich ziemlich sicher über die Länge des Arrays sind. Falls Sie aber unsicher sind, dann ist der Container `std::deque` die effizientere Wahl. Wenn der Speicher von `std::vector` nicht mehr ausreicht und erweitert werden muss, müssen alle Elemente in einen neuen Speicherblock kopiert werden. `std::deque` hingegen reserviert nur einen neuen Speicherblock und verlinkt (oder auch verkettet) lediglich die Liste mit alten Elementen mit dem neuen Block. Mit `std::deque` ist somit kein Kopieren nötig. Einen interessanten Artikel von Herb Sutter dazu finden Sie hier: <http://www.gotw.ca/gotw/054.htm>

Neben `std::vector` und `std::deque` für einen wahlfreien Zugriff gibt es noch `std::array`, welches neu mit C++11 als Alternative für ein C-Array eingeführt wurde. `std::array` wurde bereits kurz in Abschnitt 4.1.3, »Standardcontainer `std::array` (C++11)«, behandelt. Es ist ein Container mit dem Laufzeitverhalten eines rohen C-Arrays und der komfortablen Verwendung von `std::vector`, allerdings mit einer fixen Länge.

Ebenfalls neu in C++11 wurde mit `std::forward_list`, neben `std::list`, eine einfach verkettete Liste eingeführt. Der Vorteil von `std::list` ist, dass Sie hiermit die Elemente sowohl vorwärts als auch rückwärts iterieren können, weil dieser Container als doppelt verkettete Liste implementiert wurde, in der jeweils der Vorgänger und der Nachfolger gespeichert werden müssen. Mit `std::forward_list` hingegen wird nur die Adresse des Vorgängers gespeichert (also eine einfach verkettete Liste), und somit können hiermit die Elemente nur in eine Richtung vorwärts durchlaufen werden. Daher kann der Container `std::forward_list` auch wesentlich effizienter als

`std::list` sein, weil hier keine zwei Adressen (genauer ein bidirektionaler Iterator) verwaltet werden müssen.

Tabelle 12.1 bietet Ihnen nochmals einen Überblick über die sequenziellen Containerklassen der Standardbibliothek, und Abbildung 12.2 visualisiert sie.

Container	Beschreibung
<code>std::vector</code>	Ein Array (Feld) mit dynamischer Größe. Das Einfügen und Löschen am Ende erfolgt unabhängig von der Elementanzahl konstant in O(1) . Für alle anderen Elemente ist der Zugriff linear und steigt proportional zur Elementanzahl in O(n) . Der wahlfreie und lesende Zugriff auf die einzelnen Elemente ist mit O(1) ebenfalls extrem schnell. Benötigt die Headerdatei <code><vector></code> .
<code>std::array</code>	Ähnlich wie <code>std::vector</code> nur ein Array (Feld) mit fester Größe, kann als bessere Alternative für ein rohes C-Array verwendet werden. Wurde mit dem C++11-Standard eingeführt und benötigt die Headerdatei <code><array></code> .
<code>std::list</code>	Doppelt verkettete Liste, mit der das Einfügen dazwischen wesentlich schneller (im besten Fall in O(1)) als bei <code>std::vector</code> vor sich geht. Allerdings ist hiermit kein wahlfreier Zugriff möglich. Benötigt die Headerdatei <code><list></code> .
<code>std::forward_list</code>	Ähnlich wie <code>std::list</code> , aber nur als einfach verkettete Liste implementiert, kann daher nur vorwärts durchlaufen werden. Da weniger Verwaltungsaufwand im Gegensatz zu <code>std::list</code> nötig ist, werden die Aktionen zum Einfügen und Entfernen von Elementen schneller durchgeführt. Dieser Container wurde mit dem neuen C++11-Standard eingeführt und benötigt die Headerdatei <code><forward_list></code> . An einigen Punkten weicht <code>std::forward_list</code> ein wenig von der Standardschnittstelle für Container und damit von den anderen Schnittstellen ab, weshalb Sie diese Form der Liste nur in Sonderfällen verwenden sollten.
<code>std::deque</code>	<code>std::deque</code> ist der Containerklasse <code>std::vector</code> sehr ähnlich, nur ermöglicht dieser Container zusätzlich noch Löschen und Einfügen am Anfang in O(1) .

Tabelle 12.1 Sequenzielle Containerklassen der Standardbibliothek

Laufzeitklassen wie $O(1)$ und $O(n)$

Die Laufzeit von Algorithmen oder Zugriffe von Containerklassen werden gewöhnlich in verschiedene Laufzeitklassen aufgeteilt. So steht bspw. $O(1)$ für ein konstantes Laufzeitverhalten, welches unabhängig von der Elementanzahl ist. Bei $O(n)$ steht das n für die Anzahl der Elemente. Hier ist das Laufzeitverhalten linear. Das bedeutet, die benötigte Zeit steigt proportional zur Elementanzahl.

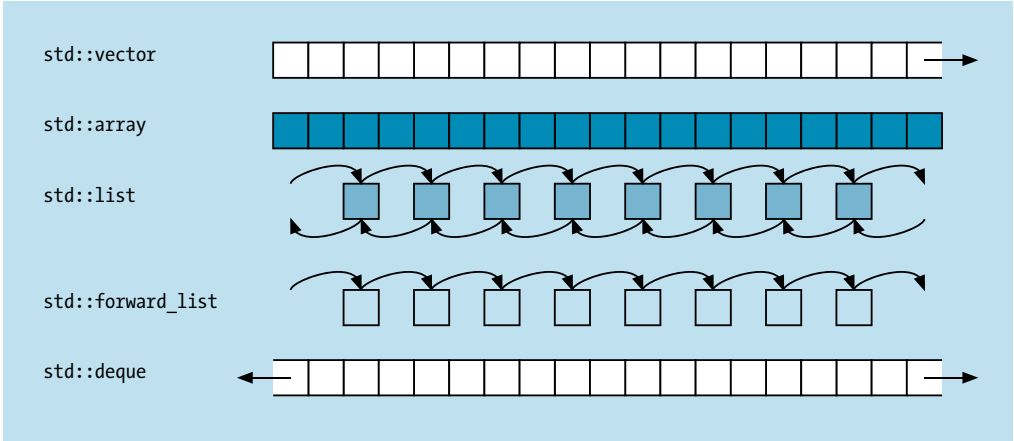


Abbildung 12.2 Sequenzielle Containertypen im Überblick

Adapterklassen

Aus den eben vorgestellten sequenziellen Containern werden die sogenannten Adapterklassen erstellt. Da eine solche Adapterklasse sequenzielle Behälter als Template-Argument verwendet, sind es im Grunde ebenfalls sequenzielle Behälter (Tabelle 12.2).

Container	Beschreibung
std::stack	Diese Klasse ist gewöhnlich eine Anpassung von <code>std::deque</code> und wird dazu verwendet, die Elemente auf einem Stapel abzulegen und von dort auch wieder zu entnehmen. Damit wird das Prinzip eines Tellerstapels realisiert, was auch unter dem LIFO-Prinzip (Last In First Out) bekannt ist. Dieser Container unterstützt keine Iteratoren. Diese Adapterklasse benötigt die Headerdatei <code><stack></code> .

Tabelle 12.2 Adapterklassen werden aus sequenziellen Containern gebildet.

Container	Beschreibung
<code>std::queue</code>	Diese Klasse wird gewöhnlich auch mit <code>std::deque</code> als Template-Argument realisiert und dient als Warteschlange. Hierbei wird ein Element auf der einen Seite abgelegt und von der anderen Seite entnommen. Das Prinzip ist wie beim Anstellen an der Supermarktkasse und auch als FIFO-Prinzip (First In First Out) bekannt. Auch dieser Container unterstützt keine Iteratoren. Diese Adapterklasse benötigt die Headerdatei <code><queue></code> .
<code>std::priority_queue</code>	Diese Adapterklasse entspricht fast der Adapterklasse <code>std::queue</code> , nur dass hier noch ein extra Prioritätsattribut vorhanden ist. Damit wird garantiert, dass immer das Element mit höchster Priorität am Anfang steht und schneller entnommen werden kann. Diese Adapterklasse benötigt die Headerdatei <code><queue>(!)</code> .

Tabelle 12.2 Adapterklassen werden aus sequenziellen Containern gebildet. (Forts.)

`std::list`, `std::vector` oder `std::deque`

Aus welchen Klassen die Adapterklassen tatsächlich adaptiert wurden, wird vom Standard nicht vorgeschrieben. Daher lautet die Betonung hier auch auf »gewöhnlich«. So kann bspw. `std::stack` durchaus aus den Klassen `std::list`, `std::vector` oder `std::deque` adaptiert sein. Alle drei Klassen bieten die für den Container nötigen Operationen `back()`, `push_back()` und `pop_back()`. Wohingegen die Adapterklasse `std::queue` eben nur von der Klasse `std::list` oder `std::deque` abstammen kann, weil hier die idealen Operationen zum Hinzufügen und Entnehmen von Elementen am Anfang und am Ende vorhanden sind. Bei `std::priority_queue` kommen entweder `std::vector` oder `std::deque` in Frage. Soweit in der Theorie. In der Praxis verwenden alle drei Adapterklassen das Template-Argument `std::deque`, weil dieser Container einfach am wirtschaftlichsten mit dem Speicher umgeht.

Sequenzielle Containerklassen verwenden

Die sequenziellen Container sind alle recht ähnlich aufgebaut und verfügen über eine Menge an gemeinsamen Eigenschaften und Methoden. Um eine Containerklasse zu verwenden, stehen Ihnen viele unterschiedliche Konstruktoren zur Verfügung. Hierzu ein paar Möglichkeiten, wie Sie verschiedene sequenzielle Container anlegen können:

```

01  std::deque<double> dbldeque;
02  std::vector<short> svec01(100);
03  std::vector<short> svec02(svec01);
04  std::list<short> slist01(svec01.begin(), svec01.end());
05  C obj01;
06  std::list<C> manyC( 100, obj01 );

07  std::stack<std::string> aStack01;
08  std::stack<std::string> aStack02(aStack01);
09  std::stack<C> aStack(100); // Fehler !!!

```

In Zeile **01** legen Sie einen leeren `std::deque`-Container für Objekte vom Typ `double` an. Mit Zeile **02** erzeugen Sie einen `std::vector`-Container mit 100 `short`-Elementen. Den Vektor aus Zeile **02** kopieren wir mithilfe des Kopierkonstruktors in Zeile **03**. Es ist auch möglich, Teilmengen eines Containers für einen anderen Container zu verwenden, wie Sie dies in Zeile **04** sehen können. Die Teilmenge wird über Iteratoren angegeben. In Zeile **06** wird ein `std::list`-Container `manyC` mit 100 Objekten vom Typ `C` erzeugt. Alle 100 Elemente werden dabei mit dem Objekt `obj01` aufgefüllt.

In Zeile **07** wird ein leerer `std::stack`-Behälter für `std::string`-Elemente angelegt. Auch bei den Adapterklassen gibt es einen Kopierkonstruktor, wie dies in Zeile **08** zu sehen ist. Nur wenn versucht wird, einen Stack mit einer bestimmten Anzahl von Elementen anzulegen, wie in Zeile **09**, funktioniert dies nicht mehr – und würde auch gar keinen Sinn bei einer solchen Datenstruktur machen. Adapterklassen definieren ohnehin nur einen Standard- und den Kopierkonstruktor.

Bevorzugte Methoden für den Zugriff auf Containerklassen

Ohne bei den einzelnen Methoden zu sehr ins Detail zu gehen, will ich Ihnen hier schnell die gängigsten Methoden für den Hausbrauch von sequenziellen Containern vorstellen. Die üblichen Routinen zur Verwendung von Containern sind: Elemente hinzufügen, Zugriff auf die Elemente, Anzahl der Elemente im Container ermitteln und Elemente wieder entfernen. Hierzu werden häufig folgende Methoden verwendet:

► Elemente hinzufügen

Bei den Containerklassen werden hier gerne `push_back()` und `push_front()` verwendet, um ein Element hinten bzw. vorne hinzuzufügen. Dass die Container `std::vector` und `std::array` kein `push_front()` haben und `std::forward_list` kein `push_back()`, sollte, so denke ich, logisch sein. Alle Containerklassen, außer `std::array`, bieten außerdem eine Methode `insert()` (bzw. `insert_after()` bei `std::forward_list`) an, um ein Element an einer bestimmten Position einzufügen. Die Adapterklassen hingegen verwenden nur die Methode `push()`, um ein Element hinzuzufügen.

► Zugriff auf ein Element

Der Zugriff auf die Elemente der Containerklasse kann mit `front()` für das erste und `back()` für das letzte Element erfolgen (gilt auch für die Adapterklasse `std::queue`). `std::forward_list` hat sinngemäß kein `back()`. Wahlfreien Zugriff mit dem Operator `[]` oder der Methode `at()` haben Sie hingegen nur mit den Containerklassen `std::array`, `std::vector` und `std::deque`. Mit den Adapterklassen `std::stack` und `std::priority_queue` erfolgt der Zugriff mit der Methode `top()`.

► Anzahl der Elemente im Container

Wie viele Elemente aktuell in der Container- oder Adapterklasse sind bzw. ob überhaupt Elemente enthalten sind, können Sie mit den Methoden `size()` (ausgenommen `std::forward_list`) bzw. `empty()` in Erfahrung bringen.

► Elemente entfernen

Das letzte Element können Sie mit der Methode `pop_back()` (nicht vorhanden bei `std::array` und `std::forward_list`) und das erste Element mit der Methode `pop_front()` (nicht vorhanden bei `std::array` und `std::vector`) entfernen. Bei allen Adapterklassen hingegen wird die Methode `pop()` verwendet.

Hierzu nun ein einfaches Beispiel, in dem einige dieser Methoden in der Praxis demonstriert werden:

```
00 // kapitel12/sequContainer.cpp
...
01 class DataClass {
02     int data;
03 public:
04     DataClass( int d=0 ) : data(d) {}
05     int getData() const { return data; }
06     friend bool operator<( const DataClass& d1, const DataClass& d2) {
07         if(d1.getData() < d2.getData()) {
08             return true;
09         }
10         return false;
11     }
12 };
...
13 std::deque<DataClass> deqData;
14 deqData.push_back( DataClass(101) );
15 deqData.push_front( DataClass(103) );
16 deqData.emplace( deqData.begin(), 202 ); // C++11
17 deqData.emplace_back( 303 );             // C++11

18 for(auto it=deqData.begin(); it != deqData.end(); ++it) {
19     if(it->getData() == 101 ) {
```

```

20     deqData.insert( it, 3, DataClass(100) );
21     break; // Schleife nach der Arbeit beenden !!!
22 }
23 }

24 for(size_t i = 0; i<deqData.size(); ++i) {
25     std::cout << deqData.at(i).getData() << " ";
26 }
27 std::cout << std::endl;

28 std::sort(deqData.begin(), deqData.end());
29 for(size_t i = 0; i<deqData.size(); ++i) {
30     std::cout << deqData.at(i).getData() << " ";
31 }
32 std::cout << std::endl;

33 std::list<int>  ilistData;
34 for(auto it=deqData.begin(); it != deqData.end(); ++it ) {
35     ilistData.push_back( it->getData() );
36 }
37 ilistData.sort();

38 for( const auto &it : ilistData ) {
39     std::cout << it << " ";
40 }
41 std::cout << std::endl;
...

```

In den Zeilen 01 bis 12 wurde eine einfache Klasse für unsere Beispiel geschrieben, welches lediglich ein Integer als Eigenschaft enthält. Außerdem wurde der `operator<()` für die Sortierung überladen. Dazu aber gleich mehr.

Hier wurde in Zeile 13 zunächst ein leerer `std::deque`-Container angelegt, welcher Daten unserer Klasse `DataClass` aufnehmen kann. In den Zeilen 14 bis 17 sehen Sie einige Möglichkeiten, wie Sie neue Elemente hinzufügen können, zunächst die Klassiker `push_back()` und `push_front()` und dann die neu in C++11 hinzugekommenen Methoden `emplace()` und `emplace_back()`. Der Vorteil der neuen `emplace`-Versionen ist, dass das Objekt nicht kopiert werden muss, wenn es in den Container kommt. Es wird kein temporärer Speicher erzeugt, sondern das Objekt wird direkt an Ort und Stelle konstruiert, an der es im Container landen soll. Vorausgesetzt natürlich, Sie verwenden die richtigen Argumente der Klasse (wie hier mit `int`).

In den Zeilen 18 bis 23 soll die Methode `insert()` demonstriert werden. Diese Methode liegt übrigens mehrfach überladen und in vielen Versionen vor. Im Beispiel

wird der Container durchlaufen, bis ein Element mit dem Wert 101 gefunden wurde. Mit `insert()` fügen wir dann in Zeile 20 drei gleiche neue Objekte mit dem Wert 100 vor den Wert 101 ein, worauf der Iterator `it` verweist. Die Ausgabe der einzelnen Elemente im Container sieht demnach wie folgt aus:

```
202 103 100 100 100 101 303
```

Die Containerklasse `std::deque` hat keine Methode zum Sortieren der einzelnen Elemente, weshalb wir in Zeile 28 den Algorithmus dafür verwenden. Damit der Algorithmus `std::sort()` auch wirklich sortiert, muss der `operator<()` für den Typ definiert sein. In unserem Fall haben wir den Operator als `friend`-Funktion in unserer Klasse `DataClass` implementiert. Die anschließende Ausgabe der Daten in den Zeilen 29 bis 32 bestätigt jetzt auch die Sortierung des Containers:

```
100 100 100 101 103 202 303
```

In Zeile 33 legen wir einen leeren `std::list`-Container für Integer-Werte an. Diesen Container wollen wir in den Zeilen 34 bis 36 verwenden, um die Eigenschaft der Klasse `DataClass`, die wir im Container `deqData` gespeichert haben, zu extrahieren. Hierfür durchlaufen wir lediglich den Container `deqData` und fügen von jedem Objekt lediglich die Eigenschaft (den Integer-Wert) mit `push_back()` am Ende der Liste hinzu. Alternativ würde sich der Algorithmus `std::copy()` dafür anbieten, aber dieser müsste hier etwas angepasst werden, weil wir hier aus `DataClass` ein `int` machen. Die Algorithmen behandeln wir in Abschnitt 12.5, »Algorithmen«. Die Containerklasse `std::list` hingegen enthält eine eigene Methode `sort()`, welche wir in Zeile 37 verwenden. Da wir hier nur noch `int`-Werte sortieren müssen, müssen wir uns um den `operator<()` keine Gedanken machen. Bei den eingebauten Datentypen übernimmt das der Compiler für uns. Die Ausgabe, hier im C++11-Stil, in den Zeilen 38 bis 41 lautet daher:

```
100 100 100 101 103 202 303
```

Zugegeben, das Beispiel zeigt noch lange nicht die Vielfalt der Containerklassen, aber es lässt sich sehr schön die Flexibilität und das Zusammenspiel mit Iteratoren und Algorithmen erkennen.

Sequenzielle Containerklasse austauschen

Sie haben hier wohl bereits gemerkt, dass sich die Schnittstellen der Containerklassen recht ähnlich sind und es häufig ohne größeren Aufwand möglich ist, einfach eine Containerklasse durch eine andere auszutauschen. Natürlich sind hier auch Kenntnisse nötig, was möglich ist und was nicht. So gibt es sequenzielle Container mit und ohne wahlfreien Zugriff. Allerdings entscheidet man sich gewöhnlich schon vorab, welchen Container man für sein Projekt verwendet. Und dies hängt gewöhnlich vom Anwendungsfall (und somit der Laufzeit) ab. Aber das wurde bereits beschrieben.

Sortieren mit `std::list` und `std::forward_list`

Es wird Ihnen am Listing bereits aufgefallen sein, dass `std::list` (wie auch `std::forward_list`) eine eigene Methode `sort()` mitbringt. Einige Container liefern hierbei eigens auf den Container spezialisierte Algorithmen als Methode mit. Im Fall von `sort()` können Sie jetzt die Methode `std::list::sort()` oder eben den globalen Algorithmus `std::sort()` zum Sortieren verwenden.

Hierzu noch ein einfaches Beispiel, welches die Adapterklassen im Einsatz zeigen soll:

```
00 // kapitel12/adaptContainer.cpp
...
01 class DataClass {
02     int data;
03 public:
04     DataClass( int d=0 ) : data(d) {}
05     int getData() const { return data; }
06 };
...
07 std::stack<DataClass> stapel;
08 stapel.push(101);
09 stapel.push(303);
10 stapel.push(404);

11 while( stapel.empty() != true ) {
12     std::cout << stapel.top().getData() << " ";
13     stapel.pop();
14 }
15 std::cout << std::endl;
```

Dies ist ein einfaches Beispiel, welches den `std::stack`-Container demonstriert. In Zeile **07** wird ein leerer Stapel angelegt, auf dem in den Zeilen **08** bis **10** drei `DataClass`-Objekte abgelegt werden. In den Zeilen **11** bis **14** nehmen wir ein Element nach dem anderen wieder vom Stack und geben den Wert auf dem Bildschirm aus. Die Ausgabe erfolgt in umgekehrter Reihenfolge, als die Elemente auf den Stack gelegt wurden, wie es für einen Stack nun mal üblich ist.

Auf `empty()` prüfen!

Es ist wichtig, dass Sie vor Methodenaufrufen wie `pop()` sichergehen, dass sich überhaupt noch etwas im Behälter befindet, wie das im Listing in Zeile **11** mit `empty()` gemacht wird. Ein `pop()` auf einen leeren Container hat unvorhersehbare Folgen.

12.2.2 Assoziative Containerklassen

Bei den assoziativen Containerklassen unterscheidet man zwischen geordneten assoziativen Containern und ungeordneten assoziativen Containern (auch als *Hashmaps* bekannt). Zu den assoziativen geordneten Containerklassen bietet die Standardbibliothek `std::set`, `std::multiset`, `std::map` und `std::multimap` an. Die Container sind als Baumstruktur realisiert, wobei der Zugriff auf die Daten in $O(\log n)$ geschieht. Das bedeutet, die Zugriffszeit erfolgt logarithmetisch und hängt von der Anzahl der Schlüssel ab.

Wenn keine Sortierung von assoziativen Containern benötigt wird, bietet der neue C++11-Standard die neuen ungeordneten Containerklassen `std::unordered_set`, `std::unordered_multiset` und `std::unordered_map`, `std::unordered_multimap` an. Hierbei generiert der Compiler dann eine Symboltabelle, um auf die einzelnen Elemente direkt zugreifen zu können. Daher ergibt sich hier eine Zugriffszeit von $O(1)$ und ist somit unabhängig von der Anzahl der Schlüssel proportional. Allerdings ist hierbei $O(1)$ nicht garantiert. Tabelle 12.3 und Abbildung 12.3 bieten Ihnen einen Überblick über die geordneten und ungeordneten assoziativen Containerklassen.

Container	Beschreibung
<code>std::set</code> <code>std::multiset</code>	Bei diesen Containern (auch Mengen genannt) werden die eingefügten Elemente sofort in einer sortierten Reihenfolge abgelegt. In diesem Container werden einzigartige Elemente als Schlüssel gespeichert. Bei <code>std::set</code> darf ein Schlüssel nur einmal vorkommen, wohingegen bei <code>std::multiset</code> mehrere gleiche Schlüssel vorkommen dürfen. Bei diesem Container sind die einzigartigen Elemente als Schlüssel gespeichert. Benötigt die Headerdatei <code><set></code> .
<code>std::map</code> <code>std::multimap</code>	Diese Container (auch assoziative Felder genannt) können im Gegensatz zu den Mengen zwei Elemente speichern, wobei ein Element der Schlüssel und das zweite Element ein beliebiges Datenelement sein kann. Hier werden also die Elemente zusammen mit dem Schlüssel gespeichert. Auch hier darf bei <code>std::map</code> der Schlüssel nur einmal vorkommen. Mit <code>std::multimap</code> hingegen darf es wieder mehrere gleiche Schlüssel geben. Hierbei wird die Headerdatei <code><map></code> benötigt.

Tabelle 12.3 Die geordneten und ungeordneten assoziativen Containerklassen

Container	Beschreibung
<code>std::unordered_set</code> <code>std::unordered_multiset</code>	Das sind die ungeordneten (nicht sortierten) Gegenstücke zu <code>std::set</code> und <code>std::multi_set</code> , die mit sogenannten Hash-Tabellen realisiert werden und erst mit dem C++11-Standard eingeführt wurden. Hierfür wird die Headerdatei <code><unordered_set></code> benötigt.
<code>std::unordered_map</code> <code>std::unordered_multimap</code>	Und das sind die ungeordneten (nicht sortierten) Gegenstücke zu <code>std::map</code> und <code>std::multi_map</code> , die ebenfalls mit Hash-Tabelle realisiert werden und erst mit dem C++11-Standard eingeführt wurden. Hierfür wird die Headerdatei <code><unordered_map></code> benötigt.

Tabelle 12.3 Die geordneten und ungeordneten assoziativen Containerklassen (Forts.)

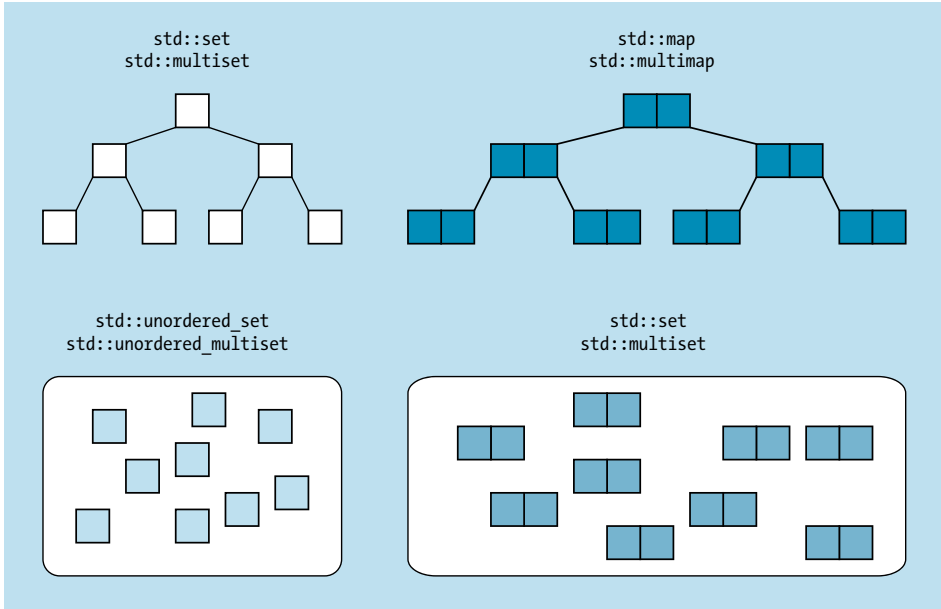


Abbildung 12.3 Assoziative Containertypen im Überblick

Assoziative Containerklassen für Mengen verwenden

Das Einfügen neuer Elemente erfolgt hier gerne mit der Methode `insert()`. Bei `std::set` wird im Gegensatz zu `std::multiset` das neue Element nur dann hinzugefügt, wenn das Element nicht existiert. Ein Element entfernen können Sie mit `erase()`. Zum Suchen wird gewöhnlich die Methode `find(T)` verwendet. Die Anzahl der gleichen Elemente erhalten Sie mit `count(T)` zurück, was bei `std::set` nur 0 oder 1 sein kann. `std::multiset` kann hingegen einen Wert größer als 1 zurückgeben.

Das folgende Listing soll die Containerklassen für Mengen kurz demonstrieren:

```

00 // kapitel12/assoContainer.cpp
...
01 class DataClass {
02     int data;
03 public:
04     DataClass( int d=0 ) : data(d) {}
05     int getData() const { return data; }
06     friend bool operator<( const DataClass& d1, const DataClass& d2 ) {
07         if(d1.getData() < d2.getData()) {
08             return true;
09         }
10         return false;
11     }
12 };
...
13 std::deque<DataClass> deqData;
14 deqData.push_back( 101 );
15 deqData.push_front( 103 );
16 deqData.insert( deqData.end(), 3, 100 );

17 std::set<DataClass> set01( deqData.begin(), deqData.end());
18 for(auto it=set01.begin(); it != set01.end(); ++it) {
19     std::cout << it->getData() << " ";
20 }
21 std::cout << std::endl;

22 std::multiset<DataClass> set02( deqData.begin(), deqData.end());
23 for(auto it=set02.begin(); it != set02.end(); ++it) {
24     std::cout << it->getData() << " ";
25 }
26 std::cout << std::endl;

27 int ival=0;
28 std::cout << "Wert zum Löschen eingeben: ";
29 if( !( std::cin >> ival ) ) return -1;
30 auto it = set02.find( DataClass(ival));
31 if( it != set02.end() ) {set02.erase(it);}

32 for(auto it=set02.begin(); it != set02.end(); ++it) {
33     std::cout << it->getData() << " ";
34 }
35 std::cout << std::endl;

```

In Zeile 17 erzeugen wir einen `std::set`-Container aus den Daten, die zuvor in den `std::deque`-Container (Zeilen 13 bis 16) gepackt wurden. Essenziell für `std::set` ist in diesem Beispiel, dass der Operator `<()` vorhanden ist, was mit der Klasse `DataClass` gegeben ist. Bei den eingebauten Typen (wie `int`, `double` etc.) und diversen Standardklassen (wie bspw. `std::string`) ist der Operator ja bereits integriert. Die Ausgabe in den Zeilen 18 bis 21 zeigt auch gleich, dass alle Elemente sortiert eingefügt wurden:

```
100 101 103
```

Hierbei können Sie auch sehr schön erkennen, dass die identischen Elemente nicht eingefügt wurden. Im Container `std::deque` waren ja drei `DataClass(100)`-Objekte enthalten. In `std::set` ist nur noch eines davon enthalten.

Für identische Elemente müssen Sie auf `std::multiset` zurückgreifen, was wir dann in Zeile 22 und der Ausgabe der Zeilen 23 bis 26 auch demonstrieren. Hier ist die Ausgabe dann:

```
100 100 100 101 103
```

In den Zeilen 27 bis 31 wird noch gezeigt, wie Sie ein Element in der Menge mit der Methode `find()` suchen (hier in Zeile 30) und mit `erase()` löschen können (Zeile 31). Die Ausgabe in den Zeilen 32 bis 34 soll dann demonstrieren, dass etwas gelöscht wurde, wenn ein entsprechendes Element gefunden wurde:

Wert zum Löschen eingeben: 101

```
100 100 100 103
```

Assoziative Containerklassen für Felder verwenden

Auch die Konstruktoren und Methoden von `std::map` und `std::multimap` sind dieselben wie schon zuvor bei `std::set` und `std::multiset`. Nur das Prinzip ist etwas unterschiedlich, weil hier jedes Element aus einem Paar aus Schlüssel/Element besteht. Der Zugriff auf den Schlüssel erfolgt hierbei mit `iterator->first`, und auf das Element können Sie mit `iterator->second` zugreifen. Um ein Element mit `insert()` hinzuzufügen, können Sie das Hilfsmittel `pair<Key, Val>` (siehe Abschnitt 12.6.1, »Template pair«) aus der Headerdatei `<utility>` oder gleich die neue C++11-Methode `emplace()` verwenden (bei g++ erst ab 4.8 dabei). Bei diesen Behältern ist ein wahlfreier Zugriff mit `operator[Key]` bzw. der Methode `at(Key)` möglich. Sie müssen aber beachten, dass bspw. mit `operator[key]` ein neues leeres Element eingefügt wird, wenn dieses noch nicht existiert.

Hierzu ein kurzes Listing, welches Ihnen ein Gefühl für die Verwendung von `std::map` und `std::multimap` geben soll:


```

00 // kapitel12/assoContainer2.cpp
...
01 class DataClass {
02     int data;
03 public:
04     DataClass( int d=0 ) : data(d) {}
05     int getData() const { return data; }
06     friend bool operator<( const DataClass& d1, const DataClass& d2) {
07         if(d1.getData() < d2.getData()) {
08             return true;
09         }
10         return false;
11     }
12 };
...
13 std::map<int, DataClass> valData;
14 valData.insert(std::make_pair(1234, 101));
15 valData.insert(std::make_pair(3213, 100));
16 valData.emplace( 4325, DataClass(202)); // C++11
17 valData.emplace( 3311, DataClass(199)); // C++11

18 for(auto it=valData.begin(); it != valData.end(); ++it) {
19     std::cout << "Schlüssel: " << it->first << std::endl;
20     std::cout << "Element  : " << it->second.getData() << std::endl;
21 }

22 std::multimap<std::string, int> numbers;
23 numbers.insert( std::make_pair("Wolf", 5557354));
24 numbers.insert( std::make_pair("Wolf", 5557345));
25 numbers.insert( std::make_pair("Galileo", 555987));
26 numbers.insert( std::make_pair("Meier", 555883));
27 // C++11-like
28 for( const auto& elem : numbers) {
29     std::cout << "Schlüssel: " << elem.first << std::endl;
30     std::cout << "Element  : " << elem.second << std::endl;
31 }

```

In Zeile 13 legen wir einen leeren `std::map`-Behälter an, welcher ein `int` als Schlüssel und eine Klasse `DataClass` als Element speichern kann. In den Zeilen 14 bis 17 übergeben dem Container einige Werte mit `insert()`. Hierbei benötigen wir das Hilfsmittel `std::pair<>`. Anstatt hier allerdings auf manuelle Angaben wie `std::pair<int, DataClass>(1234, 101)` zurückzugreifen, verwenden wir `std::make_pair(1234, 101)`, wo

sich der Compiler darum kümmert, das Richtige zu `std::pair<>` zu erstellen. Einfacher geht es mit der neu in C++11 eingeführten Methode `emplace()`, bei der sich der Compiler um die Konstruktion des Schlüssel-/Element-Paares kümmert. In den Zeilen **18** bis **21** geben wir die einzelnen Schlüssel (`first`) und Elemente (`second`) im `std::map`-Container aus. Die Sortierung erfolgt hierbei anhand des Schlüssels. Hier die Ausgabe dazu:

```
Schlüssel: 1234
Element  : 101
Schlüssel: 3213
Element  : 100
Schlüssel: 3311
Element  : 199
Schlüssel: 4325
Element  : 202
```

operator<()

Auch hier ist `operator<()` essenziell, damit die Sortierung des Schlüssels funktioniert. Wenn Sie im Beispiel also keinen eingebauten Typ für den Schlüssel verwenden (bspw. mit `map<DataClass, int>`), dann müssen Sie den `operator<()` für diesen Typ (hier `DataClass`) überladen und implementieren.

Im zweiten Teil des Listings demonstrieren wir den `std::multimap`-Container, welcher im Gegensatz zu `std::map` auch mehrere gleiche Schlüssel haben darf. Unser `std::multimap`-Container verwendet `std::string` als Schlüssel und `int` als Element. Im Beispiel soll es ein Telefonbucheintrag sein, wo eine Person ja durchaus mehrere Telefonnummern haben kann. Auch hier fügen wir in den Zeilen **23** bis **26** neue Schlüssel-/Element-Paare hinzu. Sortiert werden die Elemente lexikografisch, weil so die Implementierung von `operator<()` mit `std::string` ist. Auch hier könnten Sie `emplace()` für das Einfügen verwenden. Die Ausgabe der Schlüssel-/Element-Paare in den Zeilen **28** bis **31** haben wir diesmal mit der neuen `for`-Schleife von C++11 realisiert. Die Ausgabe sieht wie folgt aus:

```
Schlüssel: Galileo
Element  : 555987
Schlüssel: Meier
Element  : 555883
Schlüssel: Wolf
Element  : 5557354
Schlüssel: Wolf
Element  : 5557345
```

unordered-Versionen

Die neu in C++11 eingeführten Versionen `std::unordered_set` und `std::unordered_multiset` für Mengen und `std::unordered_map` und `std::unordered_multimap` für Felder lassen sich im Prinzip wie die Versionen für die Nicht-`unordered`-Versionen verwenden. Zwar gibt es bei den `unordered`-Versionen noch spezielle Hash-Funktionen, aber ansonsten können Sie im eben erstellten Beispiel die Versionen untereinander tauschen. Der entscheidende Unterschied zwischen den `unordered` und den geordneten Containern liegt in der Art und Weise, wie die Elemente gespeichert werden, und man braucht keinen `operator<`, stattdessen eben eine Hash-Funktion.

Wo bei den geordneten Containern die Elemente meistens noch in einer baumartigen Struktur sortiert gespeichert werden, sind die Elemente bei den `unordered`-Containern ungeordnet. Hierbei ist ein Element zunächst einfach nur irgendwo im Container. Ähnlich wie bei einer Tasche, in die Sie alles hineinlegen. Sie wissen, was drin ist, aber zunächst nicht wo. Sie können dies gerne testen, indem Sie in Listing *kapitel12/assoContainer2.cpp* einfach `std::map` und `std::multimap` durch `std::unordered_map` und `std::unordered_multimap` austauschen. Die Reihenfolge, in der die einzelnen Elemente jetzt gespeichert werden, kann variieren.

Ungeordnete Container ordnen die Elemente gewöhnlich in einer Hash-Tabelle an. Dies kann man sich als ein Array von verketteten Listen vorstellen. Das Prinzip lässt sich recht einfach erklären: Der Schlüssel bzw. das Schlüssel-/Wert-Paar (bei den `multi`-Versionen) kommt immer in einen *Bucket* (dt. Eimer). In welchen Bucket, wird durch eine Hash-Funktion entschieden. Das Ziel ist dabei, dass jedes Element eine eigene Position hat und Sie somit (theoretisch) direkten Zugriff mit $O(1)$ auf ein Element hätten. Allerdings können hier $O(1)$ nicht garantiert werden, weil dies von der Hash-Funktion abhängt. Entscheidend für eine gute Performance ist somit eine gute Hash-Funktion. Abbildung 12.4 soll den Sachverhalt etwas genauer demonstrieren.

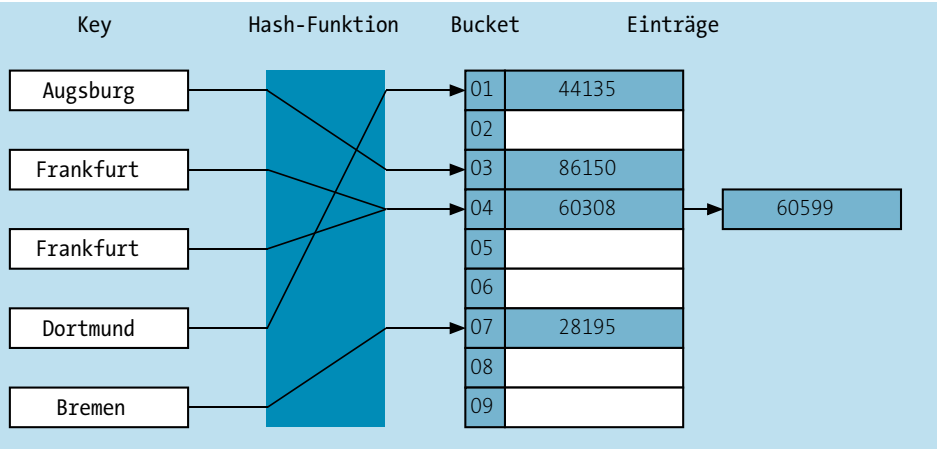


Abbildung 12.4 Ungeordnete Felder sind Hash-Tabellen, hier als Beispiel einer Postleitzahlen-Verwaltung

Im idealsten Fall sollte eine Hash-Funktion so gewählt werden, dass für gleiche Schlüssel immer derselbe Platz (hier Bucket) in Form einer verlinkten Liste verwendet wird, wie Sie dies in Abbildung 12.4 mit den beiden Postleitzahlen von Frankfurt sehen können. Für unterschiedliche Schlüssel hingegen sollte im Idealfall ein unterschiedlicher Platz im Bucket verwendet werden.

Keine Laufzeitgarantie

Wo Sie bei den Containern `std::set`, `std::multiset`, `std::map` und `std::multimap` noch garantieren können (da meistens als binärer Baum implementiert), dass das Laufzeitverhalten $O(\log n)$ ist, so ist dies bei den `unordered`-Versionen nicht mehr der Fall. Im besten Fall kann hier die Laufzeit $O(1)$ sein, also unabhängig von der Anzahl der Elemente. Betrachten Sie bspw. Abbildung 12.4, dort ist das Laufzeitverhalten für die Postleitzahl von Bremen gleich $O(1)$. Die Postleitzahl für Frankfurt allerdings nicht, da sich hier mehrere Elemente das Bucket teilen. Je weniger Elemente also dasselbe Bucket verwenden, desto besser für die Laufzeit. Teilen sich zu viele Elemente dasselbe Bucket, wird das Laufzeitverhalten negativ beeinträchtigt. In Abbildung 12.4 ist das zwar trivial, aber es sollte das Laufzeitthema recht gut verständlich machen. Und auch hier noch einmal: Das Laufzeitverhalten wird in starkem Maße von der Qualität der Hash-Funktion beeinflusst.

Hierzu zunächst ein einfaches Beispiel, welches die Verwendung eines `unordered`-Containers (hier `std::unordered_multimap`) in der Praxis mit eingebauten Typen verwendet:

```
00 // kapitel12/unorderedContainer.cpp
...
01 std::unordered_multimap<std::string, int> numbers;
02 numbers.emplace("Wolf", 5557354);
03 numbers.emplace("Wolf", 5557345);
04 numbers.emplace("Galileo", 555987);
05 numbers.emplace("Meier", 555883);

05 for( const auto& elem : numbers) {
06     std::cout << "Bucket   : "
                << numbers.bucket(elem.first) << std::endl;
07     std::cout << "Schlüssel: " << elem.first << std::endl;
08     std::cout << "Element  : " << elem.second << std::endl;
09 }

10 for(auto i=0; i < numbers.bucket_count(); ++i) {
11     std::cout << "Bucket " << i << " hat "
                << numbers.bucket_size(i) << " Elemente\n";
12 }
```

Da `std::unordered_multimap` mit C++11 eingeführt wurde, können wir gleich das komplette Listing C++11-konform erstellen. In Zeile 01 erstellen wir einen leeren `std::unordered_multimap`-Container, welcher `std::string` als Schlüssel und `int` als Element speichern kann. In den Zeilen 02 bis 05 fügen wir neue Elemente zum Container mit `emplace()` hinzu. In den Zeilen 06 bis 09 geben wir die Schlüssel und Elemente aus und auch, in welchem Bucket sie gespeichert wurden. Die Ausgabe kann wie folgt aussehen:

```
Bucket   : 3
Schlüssel: Meier
Element  : 555883
Bucket   : 1
Schlüssel: Galileo
Element  : 555987
Bucket   : 4
Schlüssel: Wolf
Element  : 5557354
Bucket   : 4
Schlüssel: Wolf
Element  : 5557345
```

Die Ausgabe zeigt hier sehr schön, dass für denselben Schlüssel (hier »Wolf«) dasselbe Bucket (hier 4) verwendet wird, so wie dies erwünscht ist. In den Zeilen 10 bis 12 geben wir außerdem noch alle Buckets aus und wie viele Elemente darin enthalten sind. Die Ausgabe könnte demnach wie folgt aussehen:

```
Bucket 0 hat 0 Elemente
Bucket 1 hat 1 Elemente
Bucket 2 hat 0 Elemente
Bucket 3 hat 1 Elemente
Bucket 4 hat 2 Elemente
```

Bei den eingebauten Typen und Standardklassen (wie hier mit `std::string`) ist die Verwendung der `unordered`-Container relativ einfach, weil hierbei der Standard-Hash `function_hash<>` aus der Headerdatei `<functional>` per Default verwendet wird. Etwas komplexer wird es allerdings, wenn Sie eigene Typen (wie eine Klasse) verwenden wollen. Hierzu benötigen Sie noch etwas mehr Detailwissen.

Unorderd-Versionen für eigene Typen

Zunächst müssen Sie noch wissen, dass für die Speicher- und Suchoperationen innerhalb des Buckets der `operator==()` verwendet wird, um auf Gleichheit zu prüfen.

Konkret bedeutet dies, dass Sie bei eingebauten Typen und Standardklassen, wie bspw. `std::string`, nicht eine eigene Hash-Funktion und Vergleichsfunktion für ope-

rator==() implementieren müssen. Allerdings benötigen Sie bei benutzerdefinierten Typen (sprich eigenen Klassen) für den Schlüssel eine eigene Hash-Funktion und müssen den operator==() überladen.

Das folgende Listing soll zeigen, wie Sie eine Hash-Funktion implementieren und wie der operator==() überladen werden sollte:

```
00 // kapitel12/unorderedContainer2.cpp
...
01 class DataClass {
02     int data;
03 public:
04     DataClass( int d=0 ) : data(d) {}
05     int getData() const { return data; }
06     friend bool operator==( const DataClass& d1, const DataClass& d2 ) {
07         return (d1.data == d2.data);
08     }
09 };

10 class DataClassHash {
11 public:
12     std::size_t operator() (const DataClass& d) const {
13         return d.getData() % 7;
14     }
15 };
...
16 std::unordered_multimap<DataClass,int,DataClassHash> valData;
17 valData.emplace( 101, 1234);
18 valData.emplace( 100, 3213);
19 valData.emplace( 202, 4325);
20 valData.emplace( 199, 3311);

21 for( const auto& elem : valData) {
22     std::cout << "Bucket   : " <<
        valData.bucket(elem.first.getData()) << std::endl;
23     std::cout << "Schlüssel: " << elem.first.getData() << std::endl;
24     std::cout << "Element  : " << elem.second << std::endl;
25 }
26 for(auto i=0; i < valData.bucket_count(); ++i) {
27     std::cout << "Bucket " << i << " hat "
        << valData.bucket_size(i) << " Elemente\n";
28 }
```

In Zeile 16 legen wir einen leeren `std::unordered_multimap`-Container an. Hier fällt gleich neben dem `DataClass`-Schlüssel und dem `int`-Element ein drittes Speicherobjekt `DataClassHash` auf, welches in Zeile 10 bis 15 als Klasse definiert wurde. Dabei handelt es sich um die Hash-Funktion.

Hash-Funktion

Der Zweck einer Hash-Funktion ist es, die Werte der Elemente im Container in ein bestimmtes Bucket zu stecken. Das Ziel ist es in der Regel, zwei gleiche Werte in dasselbe Bucket zu stecken und unterschiedliche Werte eben in ein unterschiedliches Bucket. Die Hash-Funktion kann als Funktion oder als Funktionsobjekt (siehe Abschnitt 12.5.5, »Funktionsobjekte«) wie in Listing *unorderedContainer2.cpp* mit `DataClassHash` implementiert werden. Als Parameter wird der Schlüssel-Wert erwartet und zurückgegeben werden muss ein Wert vom Typ `std::size_t`. Der so zurückgegebene Wert der Hash-Funktion wird jetzt vom Container in ein gültiges Bucket angepasst. Das geschieht allerdings automatisch, und darum müssen Sie sich nicht kümmern.

In den Zeilen 17 bis 20 werden die neuen Schlüssel-/Element-Paare mithilfe dieser Hash-Funktion dem Container hinzugefügt. Die Ausgabe in den Zeilen 21 bis 28 könnte wie folgt aussehen:

```
Bucket   : 1
Schlüssel: 202
Element  : 4325
Bucket   : 2
Schlüssel: 100
Element  : 3213
Bucket   : 3
Schlüssel: 199
Element  : 3311
Bucket   : 3
Schlüssel: 101
Element  : 1234
Bucket 0 hat 0 Elemente
Bucket 1 hat 1 Elemente
Bucket 2 hat 1 Elemente
Bucket 3 hat 2 Elemente
Bucket 4 hat 0 Elemente
```

Die Ausgabe zeigt sehr schön, dass unsere Hash-Funktion ziemlich schlecht gewählt wurde. Hier wurden bspw. in den Bucket mit der Nummer 3 zwei Elemente mit einem unterschiedlichen Schlüssel gelegt. In der Hash-Funktion haben wir einfach den Wert

des Schlüssels mit Modulo durch 7 geteilt. Schlimmer wird es eigentlich nur noch, wenn Sie einfach den Wert 0 oder 1 in Zeile 13 zurückgeben würden, weil hier dann alles in ein Bucket gelegt würde. Wenn Sie im Beispiel aber %4 anstatt %7 verwenden würden, dann hätte jedes Element sein eigenes Bucket. Sogar gleiche Schlüssel passen dann immer häufiger in dasselbe Bucket. Aber mit wachsendem Datenumfang und wachsenden Buckets, würden hiermit ebenfalls wieder nur die ersten vier Buckets berücksichtigt und die restlichen blieben ungefüllt. Somit hängt eine gute Hash-Funktion auch noch von der Größe der Hash-Tabelle und der Menge der Daten ab, die »gehasht« werden müssen.

An dieser Stelle will ich auch gar nicht mehr tiefer in das Thema Hash-Funktion eindringen, weil sich ein perfekter Algorithmus für das Beispiel dann wiederum nicht für andere Beispiele eignen würde. Im Beispiel wurde nur die Divisionsrestmethode verwendet. Wenn Sie etwas recherchieren, werden Sie feststellen, dass es eine Menge bekannter Hash-Algorithmen gibt (wie bspw. Brent-Hashing, Doppel-Hashing usw.).

Funktionsobjekt statt Vergleichsoperator ==

Den geforderten Vergleichsoperator `operator==()` haben wir in der Klasse auch gleich in den Zeilen 06 bis 08 implementiert. Dieser Vergleichsoperator ist nötig, um gleiche Werte im selben Bucket zu finden. Wenn ein Bucket also mehrere Werte enthält, würden diese ohne `operator==()` nicht gefunden. Bei den eingebauten Typen und einigen Standardklassen wird hierfür per Default das Funktionsobjekt (siehe Abschnitt 12.5.5, »Funktionsobjekte«) `equal_to<>` verwendet, welches die Elemente (bzw. Schlüssel) mit `operator==` vergleicht. Auch hier ist es gegebenenfalls möglich, dass Sie als dritten oder vierten (bei `multi`-Versionen) Template-Parameter auch noch ein eigenes Funktionsobjekt für Vergleiche verwenden. Hierfür müssen Sie ebenfalls wieder `operator()` implementieren, welcher zwei Elemente (oder zwei Schlüssel bei `multi`-Versionen) als Parameter erhält, die miteinander verglichen werden, und der `bool` zurückgibt, ob die Elemente gleich (`true`) oder ungleich (`false`) sind.

12.2.3 Container für Bit-Manipulationen – `bitset`

Etwas losgelöst von den sequenziellen und assoziativen Containerklassen dürfte der Container `bitset` daherkommen. Im Grunde handelt es sich um ein weiteres Array, welches auf Platzeinsparung optimiert wurde. Die Klasse dient dem Abspeichern und Manipulieren von Bitmengen (also 0 und 1) mit einer festen Größe.

bitset verwenden

Die Verwendung von `bitset` benötigt die Headerdatei `<bitset>`, und es ist relativ einfach, mit den einzelnen Bits zu spielen. Die Ein- und Ausgabe lässt sich ebenfalls ganz

bequem über `std::cin` und `std::cout` realisieren. Für die Manipulation von Bits wird die Methode `set()` verwendet. Mit `set()` ohne Argumente werden alle Bits auf 1 und mit `set(n)` das Bit `n` auf 1 gesetzt. Das Gegenstück, um alle oder ein bestimmtes Bit auf 0 zu setzen, ist `reset()` bzw. `reset(n)`. Alternativ können Sie auch den wahlfreien Zugriff mit `operator[]` verwenden, um ein bestimmtes Bit auf 1 oder 0 zu setzen. Zum Invertieren von Bits kann dann die Methode `flip()` für alle bzw. `flip(n)` für ein bestimmtes Bit verwendet werden. Und auch zum Konvertieren von `bitset` in einen `std::string` oder einen `long`-Wert stehen mit `to_string()` und `to_long()` zwei Methoden zur Verfügung.

Das folgende Listing soll die Verwendung vom Standardcontainer `std::bitset` in der Praxis demonstrieren:

```
00 // kapitel12/bitset.cpp
...
01 std::bitset<8> bits;
02 std::cout << "Bitte bitset eingeben: ";
03 if(! (std::cin >> bits) ) return -1;
04 std::cout << "Deine Eingabe : " << bits << std::endl;

05 bits.reset(); // Alle Bits auf 0
06 bits.set(4) ; // 5. Bit auf 1
07 std::cout << bits << std::endl; // 00010000
08 bits.flip(); // Alle Bits invertieren
09 std::cout << bits << std::endl; // 11101111
10 bits.set(); // Alle Bits auf 1
11 bits.flip(1); // 2. Bit invertieren
12 std::cout << bits << std::endl; // 11111101
```

12.2.4 Neue Möglichkeiten mit C++11

Mit C++11 sind ein paar neue Möglichkeiten hinzugekommen, um die Standardcontainer-Klassen noch komfortabler und einfacher benutzen zu können. In diesem Abschnitt sollen die interessantesten Neuerungen kurz erwähnt werden.

Vereinheitlichte Initialisierung und die neue Move-Semantik

Neu hinzugekommen ist, dass jetzt auch Containerklassen mit dem `{}`-Initialisierer (siehe Abschnitt 2.1.4, »Vereinheitlichte Initialisierung mit C++11«) Werte in einer von Komma getrennten Liste übergeben werden können.

Ebenso neu dabei ist die Move-Semantik (siehe Abschnitt 16.1, »Move-Semantik und Perfect Forwarding«), womit sich unnötiges und aufwendiges Kopieren von Objekten im Speicher vermeiden lässt. Stattdessen werden die bereits bestehenden Objekte verwendet und verschoben.

```

01  std::vector<int> ivec01={1,2,3,5,8,13,21};    // C++11
02  std::vector<int> ivec02 = ivec01;           // einfache Kopie
03  std::vector<int> ivec03 = std::move(ivec01); // C++11

```

In Zeile **01** sehen Sie die neue vereinheitlichte Initialisierung von C++11 im Einsatz. In Zeile **02** sehen Sie noch eine klassische Kopie des kompletten Inhalts von `ivec01` nach `ivec02`. In Zeile **03** hingegen verwenden wir die neue Move-Semantik mit `std::move()`, womit der komplette Inhalt des Behälters `ivec01` nach `ivec03` verschoben wird. Logischerweise ist der Behälter `ivec01` durch das Verschieben der Elemente nach `ivec03` nach dieser Aktion leer. Hier wurde zwar `std::vector` zur Demonstration verwendet, aber die neuen Möglichkeiten von C++11 lassen sich auch mit den anderen sequenziellen Containern einsetzen.

Neue for-Schleife und auto

Gerade zum Durchlaufen diverser Container ist die neue Range-based `for`-Schleife (siehe Abschnitt 3.3.5, »Mengenschleife – Range-based `for`«) perfekt geeignet, um alle Elemente mit den Methoden `begin()` und `end()` zu iterieren. Und den kilometerlangen Iterator-Namen kann man dann auch noch mit `auto` (siehe Abschnitt 2.5, »Typ `auto` (C++11)«) abkürzen.

Der klassische Zugriff, um einen kompletten Standardcontainer zu durchlaufen, sah bisher wie folgt aus:

```

01  std::vector<int> ivec01={1,2,3,5,8,13,21};    // C++11
...
02  for( std::vector<int>::iterator it=ivec01.begin();
                                it != ivec01.end(); ++it ) {
03      std::cout << *it << std::endl;
04      *it = *it+1;
05  }

```

Hier durchlaufen Sie in der `for`-Schleife mit einem Iterator alle Elemente von `ivec01` von `begin()` bis `end()` durch und geben mithilfe der Dereferenzierung den Inhalt aus (Zeile **03**). Auch bei der Änderung des Wertes wird eine Dereferenzierung verwendet (Zeile **04**).

In C++11 können Sie die gezeigte `for`-Schleife durch die Range-`for` austauschen, welche automatisch von `begin()` bis `end()` durchläuft. Hier also das C++11-Gegenstück:

```

01  for( auto &ival : ivec01 ) {
02      std::cout << ival << std::endl;
03      ival = ival+1;
04  }

```

Hierbei müssen Sie beachten, dass die Zählvariable (hier `ival`) kein Iterator mehr ist, sondern der direkte Elementtyp des Containers. Daher wird beim Zugriff auch keine Dereferenzierung mehr nötig (Zeile **02** und **03**). Die Referenz bei `auto` in Zeile **01** ist aber von enormer Bedeutung! Würden Sie hier keine Referenz verwenden, dann würden Sie in der Schleife nur mit einer Kopie arbeiten. Das Beispiel hierzu:

```
01 for( auto ival : ivec01) { // Ohne Referenz
02     std::cout << ival << std::endl;
03     ival = ival+1; //hat keinen Effekt auf den Wert von ivec01
04 }
```

Hier wurde das Referenz-Zeichen hinter `auto` in Zeile **01** weggelassen, wodurch in `ival` lediglich eine Kopie des Elements gespeichert wird. Die Ausgabe in Zeile **02** ist somit nur eine Kopie und das Ändern des Wertes in Zeile **03** hat keinerlei Effekt auf das Containerelement, sondern ändert nur den Wert der Kopie und ist in dieser Schleife sinnlos. Solche Kopieraktionen gehen auch auf Kosten der Laufzeit, und daher sollten Sie hier immer eine Referenz verwenden.

Wollen Sie dabei sichergehen, dass die Containerelemente in einer Schleife nicht versehentlich geändert werden, brauchen Sie lediglich ein `const` vor `auto` zu setzen, und die Containerelemente sind in der Range-based `for`-Schleife nur noch lesbar:

```
01 for( const auto &ival : ivec01) {
02     std::cout << ival << std::endl;
03     ival = ival+1; // Fehler da const !!!
04 }
```

Durch das `const` in Zeile **01** kann jetzt nur noch lesend auf die Referenz zugegriffen werden, und der Versuch in Zeile **03**, das Containerelement zu ändern, wird vom Compiler mit einer Fehlermeldung quittiert.

Neue `for`-Schleife für eigene Datentypen

Die neue `for`-Schleife kann auch für eigene Klassen verwendet werden. Voraussetzung dafür ist, dass Sie hier die Methoden `begin()` und `end()` implementieren, welche den Anfang und das Ende einer Liste oder Reihe zurückgeben. Hierbei müssen Sie allerdings auch eine `const`-Version implementieren, damit Sie `auto&` und `const auto&` verwenden können.

Const-Iteratoren

Das neue `auto` ist gerade in Verbindung mit Containern und den ewig langen Iterator-Namen eine erhebliche Vereinfachung. Aber die neue Bequemlichkeit hat auch wieder ihre Tücken. Betrachten Sie folgendes Beispiel:

```

01  std::vector<int> ivec01={1,2,3,5,8,13,21};    // C++11
02  for(const auto it=ivec01.begin()+1;it!=ivec01.end()-1;++it){
03      std::cout << *it << std::endl;
04      *it = *it+1;
05  }

```

Hier kann man die Absicht des Programmierers wohl erraten. Er will, dass hier nur lesend auf die einzelnen Containerelemente via Dereferenzierung mit `*it` zugegriffen werden kann, um eine Änderung wie in Zeile **04** zu unterbinden. Da wir hier nicht von `begin()` bis `end()`, sondern von `begin()+1` bis `end()-1` durchlaufen wollen, bringt die neue Range-based `for`-Schleife nichts mehr, um das `const auto&` zu verwenden. Der Compiler wird sich aber hier lediglich beschweren, dass der Iterator nicht inkrementiert werden kann, weil dieser `const` ist. Also nicht das, was hier wohl beabsichtigt war.

Aber auch für solche Zwecke hat sich der neue C++11-Standard gerüstet und mit den neuen Containermethoden `cbegin()` und `cend()` zwei Lese-Iteratoren implementiert.

```

...
02  for(auto it=ivec01.cbegin()+1;it!=ivec01.cend()-1;++it){
03      std::cout << *it << std::endl;
04      *it = *it+1; // Fehler !!! Read-only
05  }

```

Jetzt reicht hier auch wieder ein `auto it` aus, weil der Rückgabewert von `cbegin()` einen `const_iterator` zurückliefert. Eine Änderung wie jetzt in Zeile **04** des Containerelements über die Dereferenzierung wird mit einer Fehlermeldung quittiert, weil wir durch den Lese-Iterator jetzt nur noch lesend auf die Containerelemente zugreifen können.

Non-Member `begin()` und `end()`

Wenn wir schon bei `begin()` und `end()` sind, dann können wir auch gleich die Non-Member `std::begin(v)` und `std::end(v)` vorstellen. Beide Versionen wurden jetzt auch als freie Funktion implementiert und liefern den Iterator von `begin()` und `end()` zurück, ohne den Umweg über `v.begin()` bzw. `v.end()` zu gehen. Diese Versionen sind für alle Standardcontainer und auch Arrays verfügbar:

```

01  std::vector<int> ivec01={3,1,6,9,22,7,4};
02  int iarr[] = { 12, 3, 5, 3, 2, 9};

03  std::sort(std::begin(ivec01),std::end(ivec01));
04  std::sort(std::begin(iarr), std::end(iarr));

```

```

05 for(auto it=std::begin(ivec01); it!=std::end(ivec01); ++it){
06     std::cout << *it << std::endl;
07 }
08 for(auto it=std::begin(iarr);
        it != std::end(iarr); ++it){
09     std::cout << *it << std::endl;
10 }

```

Der Vorteil ist hier gleich ersichtlich, weil vor C++11 das Array bspw. für den Algorithmus `std::sort()` in Zeile **04** gar nicht genauso verwendet werden konnte wie mit einem Standardcontainer in Zeile **03**. Hier musste man immer etwas improvisieren:

```
sort(&iarr[0], &iarr[0]+sizeof(iarr)/sizeof(iarr[0])); //C++03
```

Das Prinzip können Sie dann auch bei selbst erstellten Containern realisieren, wenn `begin()` und `end()` als Methode oder Adapterfunktion implementiert sind.

Die **emplace**-Methoden

Sehr häufig verwendete Methoden zum Einfügen von Elementen in den Standardcontainer sind `insert()`, `push_back()` oder `push_front()`. In C++11 wurden diese Methoden um `emplace`-Versionen erweitert. So entsprechen bspw. `emplace_front()` und `emplace_back()` den Methoden `push_front()` und `push_back()`, und `emplace()` bzw. `emplace_hint()` entsprechen der Methode `insert()`.

Sie haben die neuen `emplace`-Methoden in diesem Buch schon das eine oder andere Mal verwendet, und einer der Hauptvorteile ist zunächst auf jeden Fall die einfachere Handhabung und bessere Lesbarkeit. Betrachten Sie folgenden Code:

```

01 class DataClass {
02     int data;
03 public:
04     DataClass( int d=0 ) : data(d) {}
05     int getData() const { return data; }
06 };
...
07 std::list<DataClass> ilist;
08 ilist.push_back(100);
09 ilist.push_front(200);
10 ilist.insert(std::begin(ilist), DataClass(101));

11 std::map<std::string, double> simap;
12 simap.insert(std::make_pair("Januar", 4343.234));
13 simap.insert(begin(simap), std::make_pair("Juli", 4313.636));

```

In diesem Beispiel finden Sie mehrere Versionen, wie ein Element zur Liste bzw. zum assoziativen Array hinzu- bzw. eingefügt wird. Im Beispiel von `std::list` mussten Sie hierfür recht umständlich in den Klammern ein temporäres Objekt anlegen. Ähnlich ist es mit `std::map`, wo zusätzlich noch die umständliche Verwendung von `std::pair<>` dazukommt (hier wird immerhin `std::make_pair()` zur Vereinfachung verwendet). Mit den neuen `emplace`-Gegenständen geht dies nun wesentlich einfacher:

```
01 std::list<DataClass>  ilist;
02 ilist.emplace_back(100);
03 ilist.emplace_front(200);
04 ilist.emplace(std::begin(ilist), 101);

05 std::map<std::string, double> simap;
06 simap.emplace("Januar", 4343.234);
07 simap.emplace_hint(begin(simap), "Juli", 4313.636);
```

Hier wurden die `push`- bzw. `insert`-Versionen einfach durch die `emplace`-Versionen ausgetauscht. Die `emplace`-Versionen übernehmen die Konstruktion und fügen das neue Element gleich an Ort und Stelle ein – alles, ohne ein temporäres Objekt zu erzeugen, und auch noch wesentlich besser lesbar und einfacher in der Anwendung.

Bessere Performance

Wer genau mitgelesen hat, dürfte es schon wissen. So wird bspw. mit `push_back()` oder `insert()` eine Kopie des Arguments im Container eingefügt. Es wird also ein temporäres Objekt erzeugt. Die `emplace`-Versionen erzeugen hingegen kein temporäres Objekt, sondern konstruieren das neu einzufügende Objekt aus den Argumenten durch eine Copy- oder Move-Konstruktion und fügen es direkt an der Stelle ein, an der Platz dafür geschaffen wurde.

Dass dies funktioniert, ist den neu in C++11 hinzugefügten Techniken mit den *Variadic Templates* (siehe Abschnitt 11.3, »Templates mit variabler Argumentanzahl«) und dem Perfect Forwarding (siehe Abschnitt 16.1.3, »Perfect Forwarding«) zu verdanken. Die `emplace`-Versionen wurden als Template-Funktion mit einer variablen Anzahl von Parametern implementiert.

Trotzdem gibt es Fälle, in denen `insert()` besser geeignet sein kann als `emplace()`. Beispielsweise bei Standardcontainern, die keine gleichen Elemente zulassen (wie `std::map`, `std::set`, `std::unordered_set` oder `std::unordered_map`). Würden Sie bspw. in unserem `std::map` vom Codeausschnitt zuvor ein absolut gleiches Element hinzufügen und `emplace()` verwenden, würde logischerweise zunächst ein Objekt mit

`emplace()` konstruiert. Wie sonst könnte der Container auch feststellen, ob ein Objekt bereits existiert. Ist das Element bereits vorhanden, hätten wir das Objekt umsonst konstruiert, und es müsste wieder zerstört werden. An dieser Stelle prüft `insert()` vorher nach, da ja bereits ein temporäres Objekt vorhanden ist, ob das Element bereits vorhanden ist.

Übergabe der Werte in Listenform

Endlich ist es auch möglich, Werte an die Container in der Listenform direkt wie bei einem klassischem C-Array zu übergeben:

```
01 std::list<int>  ilist { 4, 6, 3, 5 , 9};
02 std::map<std::string, double> simap {
03     {"Januar", 1234.123},
04     {"Februar", 4543.343}
05 };
```

Die neue einheitliche Initialisierungssyntax wurde ja bereits in Abschnitt 2.1.4, »Vereinheitlichte Initialisierung mit C++11«, behandelt.

Ebenfalls in Listenform können Sie seit C++11 eine ganze Liste von Werten mit der Methode `insert()` in den Container einfügen:

```
01 std::set<int> data;
02 data.insert({ 1, 3, 4, 7, 11 });

03 std::vector<double> ivec;
04 ivec.insert(std::begin(ivec), {3.33, 4.12, 5.32});
```

Dies funktioniert allerdings nur mit `insert()`. Mit `push_back()` oder `push_front()` funktioniert das nicht.

12.3 Kleine Methodenübersicht aller Containerklassen

Bevor wir mit den Iteratoren und Algorithmen weitermachen, finden Sie hier jetzt einen kleinen Überblick über die vorhandenen Methoden der verschiedenen Containerklassen, ohne hier auf die Syntax der einzelnen Methoden einzugehen. Viele Methoden liegen mehrfach überladen vor. Für eine detailliertere Syntaxbeschreibung empfehle ich Ihnen entweder die Manual-Seiten oder die Dokumentation Ihres Compilers. Im Web gibt es hier auch interessante komplette Referenzen der Standardbibliothek auf Seiten, wie z.B.: <http://en.cppreference.com>.

12.3.1 Methodenübersicht von sequenzielle Containern

Methoden, die einen Iterator zurückliefern

	array (C++11)	vector	deque	forward_ list (C++11)	list
begin	x	x	x	x	x
	Gibt einen Iterator zurück, der auf das erste Element im Container verweist.				
end	x	x	x	x	x
	Gibt einen Iterator zurück, der auf eine Position nach dem letzten Element des Containers verweist.				
rbegin	x	x	x		x
	Gibt einen reversen Iterator zurück, der auf das letzte (!) Element im Container verweist.				
rend	x	x	x		x
	Gibt einen reversen Iterator zurück, der auf das erste (!) Element im Container verweist.				
cbegin (C++11)	x	x	x	cbegin cbefore_ begin	x
	Wie begin(), nur wird ein const-Iterator zurückgegeben – also ein reiner Lese-Iterator.				
cend (C++11)	x	x	x	x	x
	Wie end(), nur wird ein const-Iterator zurückgegeben, somit ein reiner Lese-Iterator.				
crbegin (C++11)	x	x	x		x
	Wie rbegin(), nur wird ein const-Iterator zurückgegeben (also nur Lesen erlaubt).				
crend (C++11)	x	x	x		x
	Wie rend(), nur wird ein const-Iterator zurückgegeben, somit ein reiner Lese-Iterator.				

Tabelle 12.4 Methoden, die einen Iterator zurückliefern

Methoden, die sich auf die Kapazität des Containers beziehen

	array (C++11)	vector	deque	forward_ list (C++11)	list
size	x	x	x		x
	Gibt die Anzahl der Elemente zurück, die im Container enthalten sind.				
max_size	x	x	x	x	x
	Gibt die maximale Anzahl der Elemente zurück, die insgesamt im Container gespeichert werden können.				
empty	x	x	x	x	x
	Gibt true zurück, wenn der Container keine Elemente enthält und leer ist. Ansonsten wird false zurückgegeben, wenn Elemente im Container sind.				
resize		x	x	x	x
	Damit können Sie die Größe des Containers um n Elemente ändern. Wenn n kleiner ist als die aktuelle Größe (= size()), werden die Elemente hinter n gelöscht und zerstört.				
shrink_to_fit (C++11)		x	x		
	eine Anforderung, den Container mit der Größe von capacity() auf die tatsächliche Größe von size() zu reduzieren				
capacity		x			
	Gibt die aktuelle Größe des Containers zurück, wofür im Augenblick Speicher für die Elemente reserviert ist (nicht die Anzahl der Elemente).				
reserve		x			
	eine Anforderung, die Größe für capacity() anzupassen, damit mindestens n Elemente darin Platz haben				

Tabelle 12.5 Methoden, welche die Kapazität eines Containers betreffen

Methoden für einen direkten Zugriff auf die Elemente im Container

	array (C++11)	vector	deque	forward_ list (C++11)	list
front	x	x	x	x	x
	Gibt eine Referenz auf das erste Element im Container zurück.				
back	x	x	x		x
	Gibt eine Referenz auf das letzte Element im Container zurück.				
operator[]	x	x	x		
	Gibt eine Referenz auf das Element mit der Position n im Container zurück.				
at	x	x	x		
	Wie operator[], nur wird hier die Ausnahme out_of_range geworfen, wenn der Bereich n ungültig ist.				

Tabelle 12.6 Methoden für den Zugriff auf einzelne Elemente im Container

Methoden zum Hinzufügen, Löschen, Einfügen oder Tauschen von Elementen

	array (C++11)	vector	deque	forward_ list (C++11)	list
assign		x	x	x	x
	Weist dem Container neuen Inhalt hinzu. Der aktuelle Inhalt wird ersetzt.				
emplace (C++11)		x	x	emplace_ after	x
	Der Container wird erweitert, indem ein neues Element an einer bestimmten Position eingefügt wird. Das neue Element wird auch gleich an Ort und Stelle konstruiert.				

Tabelle 12.7 Methoden zum Hinzufügen, Einfügen, Löschen oder Tauschen von Elementen im Container

	array (C++11)	vector	deque	forward_ list (C++11)	list
insert		x	x	insert_ after	x
	Fügt ein neues Element an einer bestimmten Position im Container ein.				
erase		x	x	erase_ after	x
	Löscht ein einzelnes Element mit einer bestimmten Position oder einen Bereich von Elementen aus dem Container.				
emplace_ back (C++11)		x	x		x
	Wie <code>emplace</code> , nur wird das neue Element gleich hinter dem letzten Element konstruiert und hinzugefügt.				
emplace_ front (C++11)			x	x	x
	Wie <code>emplace</code> , nur wird das neue Element gleich am Anfang vor dem ersten Element konstruiert und hinzugefügt.				
push_front			x	x	x
	Fügt ein neues Element am Anfang des Containers vor dem aktuellen ersten Element hinzu.				
push_back		x	x		x
	Fügt ein neues Element am Ende des Containers hinter dem letzten Element hinzu.				
pop_front			x	x	x
	Entfernt das erste Element im Container, und reduziert die Größe <code>size()</code> um 1.				
pop_back		x	x		x
	Entfernt das letzte Element aus dem Container, und reduziert die Größe <code>size()</code> um 1.				

Tabelle 12.7 Methoden zum Hinzufügen, Einfügen, Löschen oder Tauschen von Elementen im Container (Forts.)

	array (C++11)	vector	deque	forward_ list (C++11)	list
clear		x	x	x	x
	Löscht alle Elemente in einem Container, womit die Größe von size() gleich 0 ist.				
swap	x	x	x	x	x
	Tauscht den Inhalt eines Containers mit einem anderen Container.				

Tabelle 12.7 Methoden zum Hinzufügen, Einfügen, Löschen oder Tauschen von Elementen im Container (Forts.)

Rein listenbasierte Methoden für std::list und std::forward_list

	forward_list (C++11)	list
splice	splice_after	x
	Transferiert Elemente von einem Container und fügt diese an einer bestimmten Position ein.	
remove	x	x
	Entfernt alle Elemente aus einem Container, welche mit dem übergebenen Wert übereinstimmen.	
remove_if	x	x
	Entfernt die Elemente aus dem Container, bei denen das übergebene Prädikat gleich true ist.	
unique	x	x
	Entfernt alle gleichen Elemente im Container, die aufeinanderfolgend vorliegen (abgesehen vom ersten Element). Am Ende gibt es keine gleichen aufeinanderfolgenden Elemente mehr in der Liste. Es gibt noch eine überladene Version, wo Sie ein binäres Prädikat verwenden können.	
merge	x	x
	Mischen zweier Listen. Waren die Listen sortiert, bleiben diese auch nach dem Mischen sortiert. Die übergebene Liste als Parameter ist anschließend leer.	

Tabelle 12.8 Methoden für Listen wie »std::list« und »std::forward_list«

	forward_list (C++11)	list
sort	x	x
	Sortiert die Liste. Für eigene Typen muss hierfür operator<() implementiert sein.	
reserve	x	x
	Kehrt die Reihenfolge der Elemente in der Liste um.	

Tabelle 12.8 Methoden für Listen wie »std::list« und »std::forward_list« (Forts.)

12.3.2 Methodenübersicht von assoziativen Containern

Auch bei den Methoden der assoziativen Container finden Sie viele bekannte Vertreter der sequenziellen Container vor.

Methoden, die einen Iterator zurückliefern

	set	multiset	map	multimap
begin	x	x	x	x
	Gibt einen Iterator zurück, der auf das erste Element im Container verweist.			
end	x	x	x	x
	Gibt einen Iterator zurück, der auf eine Position nach dem letzten Element des Containers verweist.			
rbegin	x	x	x	x
	Gibt einen reversen Iterator zurück, der auf das letzte (!) Element im Container verweist.			
rend	x	x	x	x
	Gibt einen reversen Iterator zurück, der auf das erste (!) Element im Container verweist.			
cbegin (C++11)	x	x	x	x
	Wie begin(), nur wird ein const-Iterator zurückgegeben – also ein reiner Lese-Iterator.			

Tabelle 12.9 Methoden, die einen Iterator zurückliefern

	set	multiset	map	multimap
cend (C++11)	x	x	x	x
	Wie end(), nur wird ein const-Iterator zurückgegeben, und somit ein reiner Lese-Iterator.			
crbegin (C++11)	x	x	x	x
	Wie rbegin(), nur wird ein const-Iterator zurückgegeben (also nur Lesen erlaubt).			
crend (C++11)	x	x	x	x
	Wie rend(), nur wird ein const-Iterator zurückgegeben, somit ein reiner Lese-Iterator.			

Tabelle 12.9 Methoden, die einen Iterator zurückliefern (Forts.)

	unordered_ set (C++11)	unordered_ multiset (C++11)	unordered_ map (C++11)	unordered_ multimap (C++11)
begin	x	x	x	x
	Gibt einen Iterator zurück, der auf das erste Element im Container verweist.			
end	x	x	x	x
	Gibt einen Iterator zurück, der auf eine Position nach dem letzten Element des Containers verweist.			
cbegin (C++11)	x	x	x	x
	Wie begin(), nur wird ein const-Iterator zurückgegeben – also ein reiner Lese-Iterator.			
cend (C++11)	x	x	x	x
	Wie end(), nur wird ein const-Iterator zurückgegeben, und somit ein reiner Lese-Iterator.			

Tabelle 12.10 Methoden, die einen Iterator zurückliefern (unordered-Versionen). Hier sind sinngemäß keine reversen Versionen wie »rbegin«, »rend«, »crbegin« oder »crend« vorhanden.

Methoden, die sich auf die Kapazität des Containers beziehen

	set und unordered_ set (C++11)	multiset und unordered_ multiset (C++11)	map und unordered_ map (C++11)	multimap und unordered_ multimap (C++11)
size	x	x	x	x
	Gibt die Anzahl der Elemente zurück, die im Container enthalten sind.			
max_size	x	x	x	x
	Gibt die maximale Anzahl der Elemente zurück, die insgesamt im Container gespeichert werden können.			
empty	x	x	x	x
	Gibt true zurück, wenn der Container keine Elemente enthält und leer ist. Ansonsten wird false zurückgegeben, wenn Elemente im Container sind.			

Tabelle 12.11 Methoden, welche die Kapazität des Containers betreffen

Die `unordered`-Versionen enthalten zusätzlich noch die Methode `reserve()`, die eine Anforderung ist, die Anzahl der Buckets des Containers (`bucket_count`) anzupassen, damit mindestens n Elemente darin Platz haben.

Methoden für den direkten Zugriff

	set und unordered_ set (C++11)	multiset und unordered_ multiset (C++11)	map und unordered_ map (C++11)	multimap und unordered_ multimap (C++11)
operator[]			x	
	Gibt eine Referenz auf den Wert mit dem Schlüssel k im Container zurück.			
at			x	
	Wie <code>operator[]</code> , nur wird hier die Ausnahme <code>out_of_range</code> geworfen, wenn der Schlüssel k ungültig ist.			

Tabelle 12.12 Methoden für einen direkten Zugriff

operator[] und at()

An dieser Stelle soll noch darauf hingewiesen werden, dass mit `operator[]`, im Gegensatz zu `at()`, gegebenenfalls auch ein Eintrag erzeugt wird, wenn nötig.

Methoden zum Hinzufügen, Löschen, Einfügen oder Tauschen von Elementen

	set und unordered_ set (C++11)	multiset und unordered_ multiset (C++11)	map und unordered_ map (C++11)	unordered_ multimap (C++11)
emplace (C++11)	x	x	x	x
	Der Container wird erweitert, indem ein neues Element hinzugefügt wird. Das neue Element wird auch gleich an Ort und Stelle konstruiert. Bei den Nicht-multi-Versionen wird das Element nur hinzugefügt, wenn es noch nicht existiert.			
emplace_hint (C++11)	x	x	x	x
	Wie <code>emplace</code> , nur kann zusätzlich noch die Position in Form eines Iterators angegeben werden, wo das neue Element eingefügt werden soll. Bei der Position handelt es sich in der Regel nur um einen Vorschlag. Der Container kann diesen Vorschlag verwenden, um das Einfügen zu optimieren.			
insert	x	x	x	x
	Fügt dem Container einzelne Elemente oder einen Bereich von Elementen hinzu und erweitert <code>size()</code> , um die Anzahl der hinzugefügten Elemente. Hierbei gibt es auch eine überladene Version, bei der die Position, wo das oder die neue(n) Elemente hinzugefügt werden, vorgeschlagen werden kann.			
erase	x	x	x	x
	Löscht einzelne Elemente oder einen ganzen Bereich von Elementen aus dem Container.			
clear	x	x	x	x
	Löscht alle Elemente im Container und setzt <code>size()</code> auf 0.			

Tabelle 12.13 Methoden, welche die Kapazität des Containers betreffen

	set und unordered_ set (C++11)	multiset und unordered_ multiset (C++11)	map und unordered_ map (C++11)	unordered_ multimap (C++11)
swap	x	x	x	x
Tauscht den Inhalt eines Containers mit einem anderen Container.				

Tabelle 12.13 Methoden, welche die Kapazität des Containers betreffen (Forts.)

Methoden für Operationen (Algorithmen)

	set	multiset	map	multimap
count	x	x	x	x
Gibt die Anzahl eines bestimmten Elements zurück, wie oft dieses im Container vorhanden ist. Bei den Nicht-multi-Versionen, wie bspw. <code>std::set</code> , kann diese Funktion nur 1, oder, wenn kein Element gefunden wurde, 0 zurückgeben.				
find	x	x	x	x
Sucht in einem Container nach einem bestimmten Element und gibt den Iterator der gefundenen Position zurück. Ansonsten wird ein Iterator auf <code>end()</code> zurückgegeben.				
lower_bound(x)	x	x	x	x
Gibt einen Iterator auf das erste Element im Container zurück, welches nicht vor dem übergebenen Wert bzw. Schlüssel <code>x</code> im Container steht. Entweder es ist gleich dem übergebenen Wert oder kommt danach.				
upper_bound(x)	x	x	x	x
Gibt einen Iterator auf das erste Element im Container zurück, welcher hinter dem übergebenen Wert bzw. Schlüssel <code>x</code> im Container gefunden wird.				
equal_range(x)	x	x	x	x
Gibt ein Paar (<code>pair</code>) von Iteratoren zurück, mit einem Bereich, wo alle Elemente im Container gleich dem Wert <code>x</code> sind.				

Tabelle 12.14 Methoden für Operationen auf assoziative Container

Bei den `unordered`-Versionen sind die beiden Methoden `lower_bound()` und `upper_bound()` hingegen nicht vorhanden.

Methoden für Buckets und Hash (C++11)

Die folgenden Methoden sind alle bei den `unordered`-Versionen `unordered_set`, `unordered_multiset`, `unordered_map` und `unordered_multimap` enthalten.

Methode	Beschreibung
<code>key_eq</code>	Gibt den Vergleich zweier Schlüssel, welche Sie als Argumente übergeben müssen, als <code>bool</code> -Wert zurück.
<code>hash_function</code>	Gibt das Hash-Funktionsobjekt zurück, welches vom Container verwendet wird.
<code>bucket</code>	Gibt die Bucket-Nummer zurück, wo das Element mit dem übergebenen Argument enthalten ist.
<code>bucket_count</code>	Gibt die Anzahl der Buckets zurück, die im Container enthalten sind.
<code>bucket_size</code>	Gibt die Anzahl der Elemente zurück, die im Bucket <code>n</code> gespeichert sind. Bucket <code>n</code> muss als Argument übergeben werden.
<code>max_bucket_count</code>	Gibt die maximal mögliche Anzahl von Buckets zurück, die der Container haben darf.
<code>rehash</code>	Setzt die Anzahl der Buckets im Container auf <code>n</code> , was als Argument übergeben werden muss. Damit wird der Container quasi neu rekonstruiert, allerdings nur, wenn <code>n</code> größer ist als <code>bucket_count</code> .
<code>load_factor</code>	Gibt einen Faktor zurück, der sich aus der Anzahl der Elemente im Container und der Anzahl der Buckets ergibt, also quasi <code>size/bucket_count</code> . Dieser Wert hat Einfluss darauf, wie wahrscheinlich es ist, dass zwei Elemente kollidieren (im selben Bucket gespeichert werden). Die Container verwenden <code>rehash</code> , um den Wert unter einer bestimmten Grenze (<code>= max_load_factor</code>) zu halten.
<code>max_load_factor</code>	Ohne Argumente wird der maximal mögliche Faktor zurückgegeben (siehe <code>load_factor</code>), ab dem ein <code>rehash</code> zur Rekonstruktion des Containers aufgerufen wird. Mit einem <code>float</code> -Wert als Argument können Sie diesen Grenzwert ändern. Standardmäßig ist dieser Wert 1.0.

Tabelle 12.15 Diese Methoden stehen nur den `unordered`-Versionen der assoziativen Container zur Verfügung.

12.4 Iteratoren

Die Iteratoren haben Sie in den Beispielen schon des Öfteren gesehen, und deren Sinn und Zweck ist schlicht und einfach das »Durchlaufen« (Iterieren) und Positionieren innerhalb eines Containers in einer bestimmten Reihenfolge. Die Iteratoren wurden so implementiert, dass dem Benutzer die wahre Implementierung verborgen bleibt. Diese muss auch gar nicht bekannt sein. Wichtig ist zunächst nur, dass an jeder Position auf das Element im Container lesend und/oder schreibend zugegriffen werden kann (mit den Operatoren `*` und `->`) und (natürlich) dass Sie von einer Position eines Elements zum nächsten Element des Containers wechseln können (mit `++` bzw. manchmal auch `--`). Des Weiteren können Sie auch nötige Vergleiche (`==` und `!=`) und eine Zuweisung (`=`) durchführen.

Das Prinzip kennen Sie ja bereits von gewöhnlichen Zeigern. Und in der Tat können Sie die Iteratoren als eine Art Abstraktion von Zeigern betrachten. Ein Iterator kann sogar wie ein Zeiger verwendet werden. Der Vorteil dabei ist ganz klar, dass Sie für unterschiedliche Container eine einheitliche Möglichkeit zum Positionieren und Durchlaufen haben. Um in einem Vektor, einer einfachen Liste oder einem komplexeren binären Baum eine Position im Container weiterzuschalten, können Sie in allen Fällen dieselbe Operation `it++` dafür verwenden, ohne sich Gedanken um die damit verbundenen Operationen oder die Implementierung machen zu müssen.

Headerdatei für Iteratoren

Da alle Container ihren eigenen Iterator-Typ definieren, benötigen Sie keine spezielle Headerdatei, um die Iteratoren zu verwenden.

Iteratoren verbinden

Wenn Sie einen Iterator erzeugen, wird dieser nicht automatisch mit einem Container verbunden. Eine solche Verbindung müssen Sie nachträglich bzw. direkt bspw. mit einer Zuweisung machen. Ein solcher Iterator ist auch nicht dereferenzierbar und vergleichbar mit einem Zeiger auf `nullptr`:

```
01 std::vector<int> ivec{ 1, 3, 4, 7, 11 };
02 std::vector<int>::iterator it;
```

Mit Zeile 02 haben Sie einen Iterator für den Container `std::vector` mit `int`-Elementen angelegt. Dieser Iterator ist mit noch keinem Container verbunden. Um allerdings einen Iterator mit einem Container zu verbinden, reicht nicht nur eine einfache Zuweisung aus, was der Compiler als Fehler ablehnen würde. Ein Iterator

will letztlich auch einen Iterator als Typ haben. Und hierfür bieten alle Standardcontainer die Methoden `begin()` und `end()` an (Abbildung 12.5):

- ▶ `begin()` – diese Methode liefert einen Iterator auf das erste Element im Container zurück.
- ▶ `end()` – diese Methode liefert das Ende in einem Container zurück. Das Ende ist hierbei eine Position hinter (!) dem letzten Element im Container.

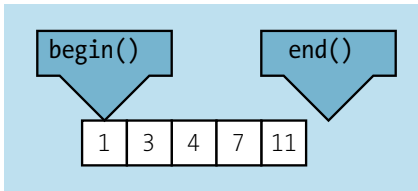


Abbildung 12.5 Jeder Container bietet die Methoden »begin()« und »end()« an, welche einen Iterator auf das erste Element und einen Iterator hinter dem letzten Element zurückliefern.

Dank der Methoden `begin()` und `end()` steht Ihnen alles Nötige zur Verfügung, um einen Container zu durchlaufen. Wenn der Container leer ist, dann sind `begin()` und `end()` gleich (`begin()==end()`).

Freie Funktion `std::begin()` und `std::end()` (C++11)

Hier können Sie selbstverständlich auch auf die freien Funktionen `std::begin(v)` und `std::end(v)` statt der Methoden `v.begin()` und `v.end()` zurückgreifen, welche neu in C++11 eingeführt wurden.

Der folgende Code demonstriert nun, wie Sie den kompletten Container mithilfe des Iterators und dank der Methoden `begin()` und `end()` durchlaufen können:

```
01 std::vector<int> data{ 1, 3, 4, 7, 11 }; // C++11
02 std::vector<int>::iterator it;
03 for(it = data.begin(); it != data.end(); ++it) {
04     std::cout << *it << std::endl;
05 }
```

Das Beispiel ist einfach. In der ersten Anweisung der `for`-Schleife bekommt der Iterator das erste Element im Container mit `begin()` zugewiesen. Die Schleife wird nun so lange ausgeführt und Element für Element auf dem Bildschirm ausgegeben, bis der Iterator auf `end()` trifft. Nach jedem Schleifendurchlauf wird die Position des Iterators mit `++it` um eine Position nach vorne gesetzt (Abbildung 12.6).

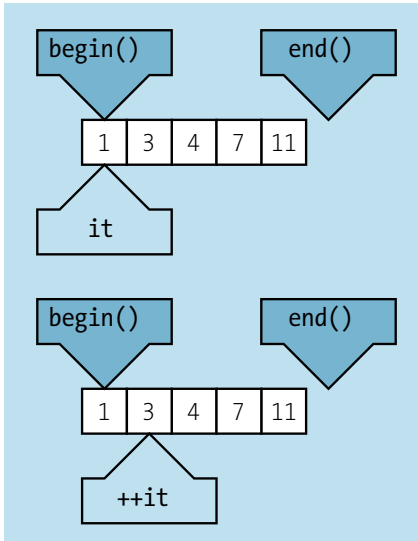


Abbildung 12.6 Der Iterator durchläuft Element für Element.

Wie in diesem Beispiel, können Sie den Iterator mit jedem Standardcontainer verwenden. würde bspw. Bei einem `std::set`-Container sähe das Beispiel ähnlich aus (Abbildung 12.7).

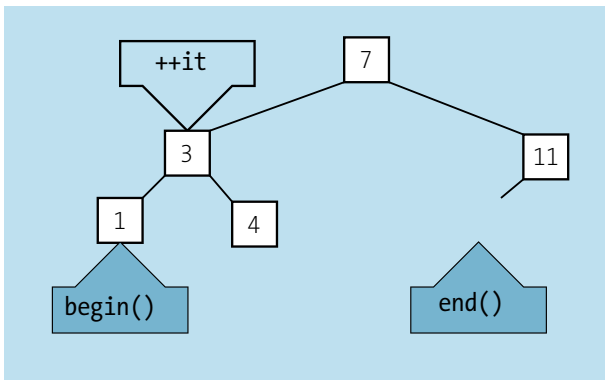


Abbildung 12.7 Durchlaufen aller Elemente in einem binären Baum wie bspw. »std::set«

Natürlich ist das Durchlaufen von binären Bäumen wesentlich komplexer als von einfachen Vektoren, aber dank der Iteratoren müssen Sie sich damit nicht befassen. Sie müssen lediglich den passenden Iterator für den Container verwenden. Bezogen auf das Beispiel in Abbildung 12.7 sieht der Code nicht wesentlich anders aus:

```
01 std::set<int> data{ 1, 3, 4, 7, 11 }; // C++11
02 std::set<int>::iterator it;
03 for(it = data.begin(); it != data.end(); ++it) {
```

```
04  std::cout << *it << std::endl;
05  }
```

Im neuen C++11-Standard brauchen Sie nicht mehr die umständliche Schreibweise `std::set<int>::iterator it=c.begin()` zu verwenden, sondern können dies mit `auto it=c.begin()` abkürzen.

++it oder it++

Die Verwendung der Prä-Inkrement-Schreibweise `++it` ist schneller als die Post-Inkrement-Schreibweise `it++`, weil intern kein temporäres Objekt angelegt werden muss, um die alte Position des Iterators zurückzugeben.

Beim Durchlaufen von assoziativen Feldern wie `std::map`, `std::multimap`, `std::unordered_map` und `std::unordered_multimap` verwenden Sie den `->`-Operator, um auf den Schlüssel mit `pos->first` und das Element mit `pos->second` zuzugreifen:

```
01  std::map<std::string, int> data{
    {"Wolf", 5551234},
    {"Meier", 555243},
    {"Galileo", 555654}
}; // C++11-like
02  for(auto it = data.begin(); it != data.end(); ++it) {
03      std::cout << it->first << std::endl; // Schlüssel
04      std::cout << it->second << std::endl; // Element
05  }
06  // oder als C++11 Range-based for-Schleife:
07  for(auto &elem : data) {
08      std::cout << elem.first << std::endl; // Schlüssel
09      std::cout << elem.second << std::endl; // Element
10  }
```

Range-based for oder Iteratoren

Wenn Sie einen kompletten Container durchlaufen und überlegen, ob Sie jetzt die neue Range-based for-Schleife dafür verwenden oder einen Iterator, so können Sie sich dafür entscheiden, was Ihnen lieber ist. Intern wird bei einer verwendeten for-Schleife wie

```
01  for(auto &elem : data) {
02      std::cout << elem << std::endl;
03  }
```

letztlich auch wieder nur Folgendes verwendet:

```

01 for(auto it = data.begin(); it != data.end(); ++it) {
02     auto &elem = *it;
03     std::cout << elem << std::endl;
04 }

```

Beide Versionen sind völlig identisch in der Laufzeit.

Konstante Iteratoren

Neben dem üblichen Iterator bieten alle Container auch eine `const`-Version mit `const_iterator` an, um einen nur lesbaren Zugriff auf einen Iterator zu gewähren:

```

01 std::deque<int> data{1, 3, 4, 7, 11};
02 std::deque<int>::const_iterator it;
03 for(it = data.begin(); it != data.end(); ++it) {
04     std::cout << *it << std::endl;
05     *it = 100; // Fehler !!! Read-only!!!
06 }

```

Mit Einführung des C++11-Standards wurde dann auch an die `auto`-Verwendung des Iterators gedacht und die Methoden `cbegin()` und `cend()` eingeführt, welche `const`-Iteratoren vom ersten Element und eine Position nach dem letzten Element im Container zurückgeben. Hier die neuen Methoden im Einsatz:

```

01 std::deque<int> data{1, 3, 4, 7, 11};
02 for(auto it = data.cbegin(); it != data.cend(); ++it) {
03     std::cout << *it << std::endl;
04     *it = 100; // Fehler !!! Read-only!!!
05 }

```

Wenn Sie hingegen die Range-based `for`-Schleife bevorzugen, reicht ein `const` vor dem Schlüsselwort `auto` aus, und der Zugriff darf auch hier nur lesend erfolgen:

```

01 std::deque<int> data{1, 3, 4, 7, 11};
02 for(const auto &elem : data) {
03     std::cout << elem << std::endl;
04     elem = 100; // Fehler !!! Read-only!!!
05 }

```

Rückwärts mit `rend()` und `rbegin()`

Container, die in beiden Richtungen durchlaufen werden können, haben außerdem noch die Iteratoren `reverse_iterator` und (die konstante Version) `const_reverse_iterator` implementiert. Bei diesen Iteratoren ist die Bedeutung von `++` und `--` vertauscht. Mit `++` wird praktisch der Container rückwärts und mit `--` vorwärts durchlaufen.

Auch die dafür angebotenen Methoden `rbegin()` und `rend()` (bzw. die konstanten Versionen `crbegin()` und `crend()`) sind hier dann verdreht. Auf das letzte Element im Container wird mit `rbegin()` zugegriffen. Auf das erste Element im Container wird hingegen mit `rend()-1` zugegriffen. Sinn und Zweck ist es, damit einen Container von hinten nach vorne zu durchlaufen.

12.4.1 Kategorien von Iteratoren

Nicht für jeden Container können dieselben Iteratoren verwendet werden. Welcher Iterator dabei verwendet wird, hängt immer von der internen Datenstruktur des Containers ab. Wenn Sie bspw. einen Container mit wahlfreiem Zugriff wie `std::vector` oder `std::deque` verwenden, dann bieten auch die Iteratoren einen wahlfreien Zugriff an. So kann bspw. bei `std::vector` ein Element direkt an der n -ten Position angesprungen werden. Container, wie bspw. Listen oder binäre Bäume (bspw. `std::list` oder `std::set`), haben intern keinen wahlfreien Zugriff und verwenden daher eine andere Kategorie von Iteratoren. Die Iteratoren werden somit, basierend auf ihren Fähigkeiten, in unterschiedliche Kategorien aufgeteilt. Bei den Containerklassen der Standardbibliothek sind dies die folgenden:

- ▶ **Forward-Iterator** (`forward_iterator`): Dieser Iterator kann nur in der Vorwärtsbewegung in einer Richtung verwendet werden. Ein klassisches Beispiel, welches diese Kategorie verwendet, ist der Container `std::forward_list`. Auch die neuen `unordered`-Versionen von C++11 können auf dieser Iterator-Kategorie implementiert sein (hier sind aber auch bidirektionale Iteratoren möglich).
- ▶ **Bidirektionaler Iterator** (`bidirectional_iterator`): Im Gegensatz zum Forward-Iterator kann dieser Iterator zusätzlich noch rückwärts bewegt werden. Die Iteratoren der Container `std::list`, `std::set`, `std::multiset`, `std::map` und `std::multimap` sind bidirektionale Iteratoren.
- ▶ **Random-Access-Iterator** (`random_access_iterator`): Dieser Iterator ist eine Erweiterung des bidirektionalen Iterators und bietet u.a. zusätzlich noch einen wahlfreien Zugriff mit `operator[]`. Die Iteratoren der Container `std::vector`, `std::deque`, `std::array` (und auch `std::string`) sind Random-Access-Iteratoren.

Zusätzlich sind hier noch zwei weitere Kategorien von Iteratoren definiert:

- ▶ **Input-Iterator** (`input_iterator`): Dieser Iterator wird nur lesend auf Objekte verwendet und kann nur vorwärts bewegt werden. Input-Stream-Operatoren verwenden bspw. einen solchen Iterator.
- ▶ **Output-Iterator** (`output_iterator`): Dieser Iterator ist das Gegenstück zum Input-Iterator und kann nur zum schreibenden Zugriff in der Vorwärtsbewegung auf Objekte verwendet werden. Auch dieser Iterator ist lediglich für einen Durchlauf geeignet.

Außerdem sollte möglichst nur ein Output-Iterator auf einem Container aktiv sein. Inserter und Output-Stream-Operatoren verwenden bspw. diesen Iterator.

In Tabelle 12.16 sehen Sie Operationen, die auf Iteratoren (eine bestimmte Kategorie) durchgeführt werden können.

Operation	Bedeutung	Erlaubt für
<code>++pos, pos++</code>	nächste Position vorwärts	alle Iteratoren
<code>--pos, pos--</code>	nächste Position rückwärts	bidirektionale und Random-Access-Iteratoren
<code>*pos</code>	Zugriff auf das Objekt	alle Iteratoren
<code>pos->...</code>	Elementzugriff	alle Iteratoren
<code>pos[n]</code>	wahlfreier Zugriff	Random-Access-Iteratoren
<code>pos1-pos2</code>	Anzahl der Objekt zwischen <code>pos1</code> und <code>pos2</code>	alle Iteratoren
<code>(pos+n)</code>	Position <code>n</code> hinter <code>pos</code>	Random-Access-Iteratoren
<code>(pos-n)</code>	Position <code>n</code> vor <code>pos</code>	Random-Access-Iteratoren
<code>==, !=</code>	Gleichheit, Ungleichheit	alle Iteratoren
<code>=</code>	Zuweisung	alle Iteratoren
<code>+=, -=</code>	zusammengesetzte Zuweisung	Random-Access-Iteratoren
<code>>, >=, <, <=</code>	kleiner/größer als ...	Random-Access-Iteratoren

Tabelle 12.16 Operationen, die auf Iteratoren durchgeführt werden können

Die Auflistung dieser Kategorien von Iteratoren soll jetzt nicht den Eindruck erwecken, dass es sich hierbei um spezielle Klassen handelt, welche Sie in der Praxis verwenden können. Hierbei handelt es sich lediglich um das Grundkonzept, auf welchem Iteratoren basieren und in welche Kategorie ein bestimmter Iterator gehört. Tabelle 12.17 bietet Ihnen nochmals einen Überblick über die verschiedenen Iterator-Kategorien und wo diese verwendet werden.

Kategorie	Verwendet in
bidirektionaler Iterator	<code>std::list</code> , <code>std::set</code> , <code>std::multiset</code> , <code>std::map</code> , <code>std::multimap</code>
Forward-Iterator	<code>std::forward_list</code> , <code>unordered-Container</code>

Tabelle 12.17 Kategorien von Iteratoren und wo diese verwendet werden

Kategorie	Verwendet in
Random-Access-Iterator	<code>std::array</code> , <code>std::vector</code> , <code>std::deque</code> , <code>std::array</code> , <code>std::string</code>
Output-Iterator	<code>ostream</code> , <code>Inserter</code>
Input-Iterator	<code>istream</code>

Tabelle 12.17 Kategorien von Iteratoren und wo diese verwendet werden (Forts.)

12.4.2 Iterator-Funktionen

Der C++-Standard liefert neben den neu in C++11 eingeführten freien Funktionen `std::begin()` und `std::end()` zusätzlich noch einige Funktionen in der Headerdatei `<iterator>` mit, welche mit Iteratoren verwendet werden können. Genau genommen handelt es sich dabei um eine Möglichkeit, allen Iteratoren die Fähigkeiten zu geben, welche normalerweise nur ein Random-Access-Iterator hat.

advance()

Mit dieser Funktionen können Sie den Iterator auf eine bestimmte Position setzen, ähnlich wie bei einem wahlfreiem Zugriff mit einem Random-Access-Iterator mit `operator[]`. Die Syntax dieser Funktion lautet:

```
void advance(InputIterator& pos, Distance n);
```

Damit können Sie den Iterator von der Position `pos` um `n` Elemente vorwärts oder rückwärts setzen. Rückwärts geht es nach wie vor nur bei bidirektionalen und Random-Access-Iteratoren.

Im folgenden Beispiel implementieren wir damit quasi einen wahlfreien Zugriff auf den Container `std::list`, was bei diesem Container so nicht angeboten wird:

```
01 #include <iterator>
...
02 std::list<int> data{1, 3, 4, 7, 11};
03 std::list<int>::const_iterator it = data.begin();

04 std::advance(it, 3);
05 std::cout << *it << std::endl; // = 7
06 std::advance(it, -1);
07 std::cout << *it << std::endl; // = 4
```

In Zeile **04** wird der Iterator mit `advance()` auf das vierte Element in der Liste gesetzt. In Zeile **06** setzen wir den Iterator wieder um eine Position zurück auf das dritte Element.

Die Verwendung ist sehr einfach, aber auch gefährlich, wenn man nicht aufpasst. Hier könnten Sie nämlich ohne Problem schreiben `std::advance(it, 100)`. Was jetzt genau passiert kann nicht vorausgesagt werden – und ist somit nicht mehr definiert. Es gibt einige Compiler, die bei einer Bereichsüberschreitung von `end()` wieder bei `begin()` fortfahren.

Wirklich wahlfreier Zugriff für alle Container?

Natürlich hängt die Verwendung von `advance()` vom verwendeten Container ab. So ist es nach wie vor nicht möglich, dass bei `std::list` ein echter wahlfreier Zugriff gemacht werden kann, weil die interne Datenstruktur dies einfach nicht hergibt. Im Grunde wird daher intern nur `++pos` n -mal (oder `--pos` bei negativen Wert) aufgerufen. Bei Random-Access-Iteratoren, wie diese bspw. mit `std::vector` verwendet werden, wird dann `pos+=n` aufgerufen, was in diesem Fall wieder ein echter wahlfreier Zugriff wäre.

distance()

Um die Anzahl der Elemente zwischen zwei Positionen desselben Containers zu ermitteln, können Sie die Funktion `distance()` verwenden. Die Syntax dieser Funktion lautet:

```
Dist distance(InputIterator pos1, InputIterator pos2);
```

Die Funktion kann bspw. dazu verwendet werden, um zu ermitteln, an welcher Position sich ein Element in einem Container befindet. Hierzu ein einfaches Beispiel:

```
01 #include <iterator>
...
02 std::list<int> data{1, 3, 4, 7, 11};
03 std::list<int>::const_iterator it1;
04 std::list<int>::const_iterator it2;

05 it1=std::find(std::begin(data), std::end(data),3);
06 it2=std::find(std::begin(data), std::end(data),11);

07 std::cout << std::distance(it1, it2) << std::endl; // = 3
```

Hier wird zunächst in der Liste nach den Elementen mit dem Wert 3 (Zeile 05) und dann 11 (Zeile 06) gesucht und die zurückgegebenen Positionen mit `distance()` in Zeile 07 verwendet, um den Abstand zwischen den beiden Elementen zu ermitteln.

Wo intern für Random-Access-Iterator ein einfaches `pos2-pos1` verwendet wird, muss bei anderen Iteratoren der komplette Container durchlaufen werden, um die Posi-

tion zu ermitteln. Unnötig wohl zu erwähnen, dass dies im Fall von großen Listen bspw. nicht unbedingt sehr effektiv ist.

next() und prev() (C++11)

Neu mit C++ wurden die Hilfsfunktionen `next()` und `prev()` eingeführt, um sich mit dem Iterator zur nächsten oder einer vorherigen Position zu bewegen. Der folgende Codeausschnitt soll kurz zeigen, wozu das gut sein kann:

```
01  std::vector<int> data{1, 3, 4, 7, 11};
02  auto it = data.begin();

03  for(; it !=data.end() && it+1 != data.end(); ++it) {
04      std::iter_swap(it, it+1); // benötigt <algorithm>
05  }
06  for( const auto& elem : data ) {
07      std::cout << elem << std::endl;
08  }
```

In diesem etwas sinnfreien Beispiel überprüfen wir im Kopf der `for`-Schleife, ob sich der Iterator `it` noch nicht am Ende des Vektors befindet UND ob sich das nächste Element (`it+1`) ebenfalls noch nicht am Ende der Liste befindet. In diesem Fall tauschen wir beide Elemente im Vektor mit der ebenfalls freien Funktion `std::iter_swap` einfach aus, indem wir auch hier die Position `it` und `it+1` als Parameter verwenden.

Da `std::vector` hier einen Random-Access-Iterator verwendet, ist ein solcher Zugriff kein Problem. Würden Sie allerdings Container verwenden (bspw. `std::list`), welche Forward- oder bidirektionale Iteratoren benutzen, funktioniert dies so nicht mehr, weil diese nicht `+` (oder gegebenenfalls `-`) als Operator unterstützen. Hier müssten Sie dann entweder einen weiteren Iterator verwenden, oder aber Sie verwenden die neue Funktion `std::next()` dafür. Bezogen auf `std::list` sieht dies dann wie folgt aus:

```
01  std::list<int> data{1, 3, 4, 7, 11};
02  auto it = data.begin();

03  for(;it !=data.end() && std::next(it) != data.end();++it) {
04      std::iter_swap(it, std::next(it));
05  }
06  for( const auto& elem : data ) {
07      std::cout << elem << std::endl;
08  }
```

Allerdings gilt auch hier, dass ein Zugriff mit `std::next()` oder `std::prev()` nicht sicherstellt, ob Sie nicht über den Bereich `end()` oder unter `begin()` hinauslaufen. Beide Funktionen bieten hier auch noch eine überladene Version mit `an`, bei der Sie

zusätzlich noch die Anzahl der Positionen angeben können, wie weit Sie den Iterator nach vorne oder hinten rücken wollen (bspw. `next(it, pos)`).

Dass Forward-Iteratoren nicht mit den freien Funktionen `prev()` funktionieren können, sollte hier auch einleuchten.

`std::iter_swap()`

Ebenfalls eine freie Hilfsfunktion für Iteratoren ist `std::iter_swap(it1, it2)`, womit Sie die Elemente von zwei Iteratoren tauschen können. Allerdings ist dies keine freie Iterator-Funktion der Headerdatei `<iterator>`, sondern gehört zu den Algorithmen und benötigt daher die Headerdatei `<algorithm>`.

12.4.3 Iterator-Adapter

Mit sogenannten Iterator-Adaptern liefert die Standardbibliothek spezielle Funktionalitäten (auch Iterator-Generatoren genannt) an, die u.a. auch die Anwendungsmöglichkeiten der Algorithmen der Standardbibliothek erweitern. Auch hierfür benötigen Sie wieder die Headerdatei `<iterator>`.

Insertter (Einfüge-Iteratoren)

Wenn Sie mit dem gewöhnlichen Iterator ein Element im Container einfügen würden, würden Sie das aktuelle Element, auf das der Iterator verweist, überschreiben. Zum Einfügen neuer Elemente in einem Container gibt es sogenannte Insert-Iteratoren. Die Standardbibliothek stellt in `<iterator>` drei verschiedene Insert-Iteratoren zur Verfügung: `front_inserter`, `back_inserter` und `inserter`.

Natürlich lassen sich diese Einfüge-Iteratoren nicht willkürlich verwenden. So ist es logischerweise nicht möglich, einen `front_inserter`-Iterator auf einen Vektor anzuwenden, weil es nicht erlaubt ist, ein Element am Container-Anfang einzufügen.

Mit `front_inserter(container)` fügen Sie das einzufügende Element am Anfang des Containers ein. Das Gegenstück `back_inserter(container)` fügt das neue Element am Ende des Containers ein. Mit dem allgemeinen `inserter(container, pos)` können Sie Elemente an einer beliebigen Position im Container einfügen.

Hierzu ein kurzer Codeausschnitt zu `back_inserter()`:

```
01 std::list<int> dat01{1, 3, 4, 7, 11};
02 std::list<int> dat02{ 2, 5, 12 };

03 std::copy(dat01.begin(), dat01.end(), back_inserter(dat02));
04 for( const auto& elem : dat02) {
05     std::cout << elem << std::endl;
06 }
```

Hier verwenden wir in Zeile **O3** den Iterator-Adapter `back_inserter()`, um mit dem Algorithmus `std::copy()` alle Elemente aus dem Container `dat01` von `begin()` bis `end()` hinter den bereits vorhandenen Elementen von `dat02` einzufügen.

Stream-Iteratoren

Die Stream-Iteratoren werden dazu verwendet, mit Ein- bzw. Ausgabeströmen zu arbeiten. Dabei wird zwischen zwei Stream-Iteratoren unterschieden: `ostream_iterator` und `istream_iterator`. Um die Stream-Iteratoren zu verwenden, müssen Sie außerdem die Header-Datei `<fstream>` einfügen.

Die beiden Stream-Iteratoren werden gewöhnlich dazu verwendet, von Strömen mit den bekannten Operatoren `operator<<` und `operator>>` sequenziell zu lesen bzw. zu schreiben. Dabei ist der `ostream_iterator` der Ausgabe-Iterator und der `istream_iterator` der Eingabe-Iterator. Ein einfaches Beispiel dazu:

```
...
01 ifstream source("Dateiname");
02 istream_iterator<string> pos(source), end;
03 while(pos != end) {
04     cout << *pos << endl;
05     ++pos;
06 }
```

Mit der Dereferenzierung von `pos` in der Schleife können Sie den gelesenen Wert ausgeben lassen. Wenn Sie diesen Codeausschnitt in der Praxis testen, werden Sie feststellen, dass der Iterator `istream_iterator` alle bekannten Whitespaces wie Leerzeichen, Tabulatorzeichen und Zeilenendezeichen vor einem Element ignoriert und zwischen zwei Elementen als Trennzeichen verwendet.

Das Gegenstück dazu ist der `ostream_iterator`, der zum Schreiben von Elementen verwendet wird. Allerdings muss man beachten, dass der `ostream_iterator` aufeinanderfolgende Elemente mit dem Operator `<<` gewöhnlich ohne Trennzeichen nacheinander in den Stream schiebt. Hierbei ist es möglich, diesem Iterator bei der Definition eine Zeichenkette vom Typ `char*` mitzugeben, die nach jedem Element als Trennzeichen fungieren soll:

```
01 ifstream quelle("testdatei.cpp");
02 ofstream ziel("testdatei.bak");
03 istream_iterator<string> iPos(quelle), ende;
04 ostream_iterator<string> oPos(ziel, "\n");
05 while(iPos != ende) {
06     *oPos++ = *iPos++;
07 }
```

Hier wurde nach jedem Trennzeichen ein Newline-Zeichen angehängt, so dass jeder neue String in eine neue Zeile der Datei geschrieben wird.

Zum Weiterlesen

Hier wurde auf das Thema Streams schon vorgegriffen. Dem Thema widmen wir uns im Buch in Kapitel 13, »Die Stream-Ein-/Ausgabeklassen von C++«.

Reverse-Iteratoren

Bei den Reverse-Iteratoren werden der Inkrement- und Dekrement-Operator so implementiert, dass diese sich genau umgekehrt verhalten. Verwenden Sie diese Iteratoren mit einem Algorithmus, werden die Elemente in genau umgekehrter Reihenfolge verarbeitet. Abgesehen von `std::forward_list` und den `unordered`-Containern, sind alle Container dazu fähig, Reverse-Iteratoren zu verwenden.

```
01 std::vector<int> data{ 1, 2, 3, 4 };
02 auto it = data.rbegin();
03 for(; it != data.rend(); ++it) {
04     std::cout << *it << std::endl; // = 4 3 2 1
05 }
```

Die Methoden `rbegin()` und `rend()` liefern jeweils einen Reverse-Iterator. Daher startet die Ausgabe hier in umgekehrter Reihenfolge, weil `rbegin()` auf das letzte Element im Container verweist und mit `++it` jeweils um ein Element von hinten nach vorne gesprungen wird. Dies wird so lange wiederholt, bis `rend()` der Position vor (!) dem ersten Element entspricht.

Natürlich können Sie einen solchen Reverse-Iterator auch ohne `auto` wie folgt definieren:

```
02 std::vector<int>::reverse_iterator it(data.rbegin());
```

Reverse-Iterator zu normalem Iterator konvertieren

Falls Sie einen Reverse-Iterator in einem normalen konvertieren wollen, können Sie dies mit `base()` machen, was eine Methode der Klasse `reverse_iterator` ist.

Move-Iteratoren (C++11)

Mit C++11 wurde ja die neue Move-Semantik (siehe Abschnitt 16.1, »Move-Semantik und Perfect Forwarding«) eingeführt. Daher werden hierfür auch Move-Iteratoren angeboten. Ein kurzer Codeausschnitt demonstriert diese neuen Move-Iteratoren:

```

01  std::vector<std::string> data{
    "string1", "string2", "string3", "string4", "string5"
    };
02  std::list<std::string> listdata(
    std::make_move_iterator(data.begin()+1),
    std::make_move_iterator(data.end()-2)
    );
03  for( const auto & elem : listdata) {
04      std::cout << elem << " "; // string2 string3
05  }
06  std::cout << std::endl;
07  for( const auto & elem : data) {
08      std::cout << elem << " "; // string1 string4 string5
09  }
10  std::cout << std::endl;

```

In Zeile 02 wurde dank des neuen Move-Iterators der Inhalt von `data.begin()+1` bis `data.end()-2` nach `listdata` verschoben statt kopiert.

12.5 Algorithmen

Für die Verarbeitung umfangreicher Datenmengen bietet die Standardbibliothek fertige Algorithmen in der Headerdatei `<algorithm>` (und dem Namensraum `std`) an. Um die Algorithmen zu verwenden, müssen Sie auch die Headerdatei angeben.

Algorithmus

Mit einem Algorithmus ist eine Funktion zum Lösen eines Problems in endlich viele oder genau beschriebene Schritte gemeint.

Hierbei werden häufig verwendete Algorithmen zum Suchen, Sortieren, Kopieren, Vergleichen, Zählen, Ersetzen usw. angeboten. Da diese Algorithmen nicht als Teil einer Klasse (bzw. Teil eines Containers) implementiert sind, sondern als globale Template-Funktion, können diese sehr flexibel für alle Arten von Containern, aber auch gewöhnliche Typen wie Arrays, Strings oder selbst definierte Container verwendet werden.

Containerklassen mit eigenen Algorithmen als Methode

Einige Container, wie bspw. `std::list`, bieten mit der Methode `sort()` einen eigenen, speziell auf den Container angepassten Algorithmus an. In diesem Fall wird empfohlen, den Algorithmus des Containers zu verwenden, weil dieser meistens besser ist als die globale Template-Funktion.

12.5.1 Bereich

Damit die Algorithmen auch funktionieren, müssen Sie einen (oder manchmal auch mehrere) Bereich(e) mit zwei Iteratoren mit `[begin, end)` festlegen. Bei Containern kann (muss nicht) dieser Bereich alle Elemente beinhalten und wird (muss auch nicht) häufig mit den Methoden `begin()` und `end()` festgelegt. Bezogen auf einen Container `x` umfasst `[x.begin(), x.end())` alle Elemente im Container.

Wichtig ist hierbei auch, dass der Iterator des letzten Elements vom Iterator des ersten Elements erreichbar ist. Daher spricht es für sich selbst, dass die Iteratoren für das erste und letzte Element vom selben Container sein müssen. An dieser Stelle müssen Sie daher als Programmierer selbst dafür sorgen, dass der Funktionsaufruf mit einem gültigen Bereich gemacht wird. Ansonsten ist das weitere Verhalten des Programms nicht vorhersehbar.

Halboffen: `[first, last)`

Bezogen auf die Syntaxangabe mit dem Bereich `[first, last)` muss noch eine Erklärung gemacht werden. Hierbei beginnt die Bearbeitung des Algorithmus an der Position `first` und wird so lange fortgesetzt, bis die Position `last` erreicht wurde. Die Position `last` gehört hierbei nicht mehr zum Bereich, wie dies bspw. bei der Methode `end()` ja auch der Fall ist, die die Position hinter dem letzten Element darstellt. Dieser Bereich wird als *halboffen* bezeichnet und in der mathematischen Notation `[first, last)` angegeben.

Der folgende Codeausschnitt soll einige Algorithmen in der Praxis zeigen:

```
00 // kapitel12/algorithm/range01.cpp
01 #include <algorithm>
...
02 std::vector<int> dat01{2, 5, 3, 4, 11};
03 auto pos = std::find(dat01.begin(), dat01.end(), 4);
04 if( pos != dat01.end() ) {
05     std::cout << "Element gefunden an Pos. "
                << std::distance(dat01.begin(), pos)
                << std::endl;
06 }
07 std::sort(dat01.begin(), dat01.end());
...
```

In Zeile 03 wird der Algorithmus `std::find()` verwendet, mit dem im Bereich `dat01.begin()` bis `dat01.end()` (also im kompletten Container) nach dem Wert 4 gesucht wird. Verweist der Iterator nicht auf `dat01.end()`, dann haben wir den Wert gefunden und lassen uns die Distanz vom ersten Element in Zeile 05 ausgeben. Ein

weiterer Algorithmus wird in Zeile **07** mit `std::sort()` verwendet, um alle Elemente im Container im Bereich `dat01.begin()` bis `dat02.end()` zu sortieren.

12.5.2 Mehrere Bereiche

Einige Algorithmen verwenden auch mehrere Bereiche. Hierbei wird in der Regel für den ersten Bereich der Anfang und das Ende benötigt, und für den zweiten Bereich wird meistens nur der Anfang verwendet. Ein einfaches Beispiel dazu ist der Algorithmus `std::copy()`:

```
std::copy( dat01.begin(), dat01.end(), // erste Bereich
          dat02.begin());             // zweiter Bereich
```

Hierbei wird der komplette Inhalt des ersten Bereichs von `dat01.begin()` bis `dat02.end()` in den zweiten Bereich kopiert, der mit `dat02.begin()` beginnt.

Auch hier ist wichtig, dass das Ende des ersten Bereichs vom Anfang des ersten Bereichs erreicht werden kann und somit vom selben Container sein muss. Zusätzlich müssen Sie noch sicherstellen, dass im zweiten Bereich mindestens so viele Elemente (oder Platz für so viele Elemente) vorhanden sind, wie im ersten Bereich (und das gilt nicht für schreibende Algorithmen). Folgender Code würde zu einem Laufzeitfehler bzw. zu undefiniertem Verhalten führen:

```
01 std::vector<int> dat01{2, 5, 3, 4, 11};
02 std::list<int> dat02;
03 std::copy(dat01.begin(), dat01.end(), dat02.begin());
```

Das Problem ist hierbei, dass mit `std::copy()` alle Elemente vom Bereich `dat01.begin()` bis `dat01.end()` an den Anfang des Bereichs `dat02.begin()` kopiert werden, aber `dat01` gar nicht genügend Speicherplatz dafür reserviert hat. Meistens ist man es gewohnt, dass der Container einem die Speicherverwaltung abnimmt. Aber in diesem Fall verwenden Sie ja globale Template-Funktionen und keine Methoden des Containers.

Eine Lösung des Problems wäre, ein Ziel zu definieren, welches genügend Elemente (bzw. Platz für die Elemente) enthält:

```
01 std::vector<int> dat01{2, 5, 3, 4, 11};
02 std::list<int> dat02;
03 dat02.resize(dat01.size());
04 std::copy(dat01.begin(), dat01.end(), dat02.begin());
```

Dadurch, dass hier in Zeile **03** der Platz für die erforderlichen Elemente reserviert wurde, gibt es jetzt auch keine Probleme mehr bei der Verwendung des Algorithmus in Zeile **04**. Eine zweite Lösung wäre die Verwendung eines Inserter-Iterators (siehe den Abschnitt »Inserter (Einfüge-Iteratoren)« in Abschnitt 12.4.3) wie bspw. mit:

```
std::copy(dat01.begin(), dat01.end(), back_inserter(dat02));
```

Aber nochmals, auch wenn hier nur `std::copy()` demonstriert wurde, gilt dies für alle anderen Algorithmen ebenso, welche zwei Bereiche verwenden. Beispielsweise muss beim Algorithmus `std::equal()`, mit dem Sie Element für Element von zwei Bereichen vergleichen können, der zweite Bereich mindestens so groß sein wie der erste.

C++11-only

Da die abgedruckten Listings häufig C++11-Techniken enthalten, müssen Sie den Code an entsprechenden Stellen anpassen, falls Ihr Compiler noch kein C++11 kann. Methoden wie `cbegin()` und `cend()` wurden erst in C++11 eingeführt. Hierfür müssen Sie dann `begin()` und `end()` verwenden, oder aber Sie verwenden `const_iterator`. Das Schlüsselwort `auto` bei Iteratoren funktioniert auch erst mit C++11, weshalb Sie hier den Iterator komplett deklarieren müssen. Auch das Range-basierte `for` kann erst mit neuen C++11-Compilern verwendet werden.

12.5.3 Algorithmen mit Prädikat

Ein Prädikat kann eine Funktion, ein Funktionsobjekt oder (seit C++11) eine Lambda-Funktion sein, welche dem Algorithmus mitgegeben wird und einen Wert vom Typ `bool` zurückgibt. Mit einem solchen Prädikat soll ermittelt werden, ob ein Element eine bestimmte Eigenschaft hat. Trifft dies zu (`= true`) dann wird der Algorithmus auf dieses Element ausgeführt. Hierbei muss noch hinzugefügt werden, dass das Prädikat das Element nicht verändern darf. Einige dieser Algorithmen mit einem Prädikat erkennen Sie anhand der Endung `_if`.

Hierzu ein einfaches Beispiel, welches eine einfache Funktion definiert und diese mit dem Algorithmus `std::count_if()` für jedes Element in einem bestimmten Bereich aufruft:

```
00 // kapitel12/algorithm/predicat01.cpp
...
01 bool odd(int val) {
02     return (val % 2);
03 }
...
04 std::vector<int> dat01{2, 5, 3, 4, 11};
05 int ctr = std::count_if(dat01.begin(), dat01.end(), odd);
06 std::cout << ctr << " ungerade Zahlen enthalten\n"; // = 3
```

In Zeile 05 wird mit dem Algorithmus `std::count_if()` für jedes Element im Bereich `dat01.begin()` bis `dat01.end()` die Funktion `odd()` mit dem Wert des aktuellen Ele-

ments aufgerufen. Die Funktion gibt `true` zurück, wenn der Wert des Elements eine ungerade Zahl ist. Für alle mit `true` zurückgegebenen Elemente wird ein interner Zähler des Algorithmus `std::count_if` um den Wert 1 inkrementiert. Am Ende erhält die Variable `ctr` als Rückgabewert des Algorithmus die Anzahl der Elemente, bei denen die Funktion `odd()` gleich `true` zurückgegeben hat – in diesem Beispiel also die Anzahl der ungeraden Zahlen im Container `dat01`.

Binäres Prädikat

Neben Algorithmen mit einfachem Prädikat gibt es auch solche mit zwei Prädikaten, womit eine Bedingung für zwei Elemente im Container formuliert werden muss. Also müssen Sie hierfür die Funktion, das Funktionsobjekt oder die Lambdafunktion so schreiben, dass zwei Argumente aufgenommen werden können. Auch hierzu ein einfaches Beispiel:

```
00 // kapitel12/algorithm/predicat02.cpp
...
01 bool mycheck(double val1, double val2) {
02     return(static_cast<int>(val1)==static_cast<int>(val2));
03 }
...
04 std::vector<double> dat01{2.1, 5.5, 3.2, 4.1};
05 std::vector<double> dat02{2.1, 5.5, 3.3, 4.1};
06 bool chk = std::equal( dat01.begin(), dat01.end(),
                        dat02.begin() );
07 if( chk == true ) {
08     std::cout << "Die Daten sind gleich!\n";
09 }
10 else {
11     std::cout << "Die Daten sind unterschiedlich!\n";
12 }
13 chk = std::equal( dat01.begin(), dat01.end(),
                    dat02.begin(), mycheck );
14 if( chk == true ) {
15     std::cout << "Die Daten sind gleich!\n";
16 }
17 else {
18     std::cout << "Die Daten sind unterschiedlich!\n";
19 }
```

In Zeile **06** vergleichen wir die beiden Container `dat01` und `dat02` Element für Element auf Gleichheit. Dass hier `std::equal()` nicht `true` ist, liegt daran, dass sich das dritte Element von `dat01` und `dat02` in der Nachkommastelle unterscheidet. Der

Algorithmus `std::equal()` bietet aber noch eine zweite Version mit einem binären Prädikat mit an. Diese Version verwenden wir in Zeile 13, indem wir hier erneut Element für Element von `dat01` und `dat02` vergleichen, nur diesmal wird die Funktion `mycheck()` (Zeilen 01 bis 03) zum Vergleich verwendet.

In der Funktion `mycheck()` vergleichen wir in Zeile 02 nur den Wert vor dem Komma, indem wir hier eine Umwandlung von `double` nach `int` durchführen. Es wird daher nur dann `true` zurückgegeben, wenn der Wert vor dem Komma gleich ist. Die Nachkommastelle interessiert uns in diesem Vergleich nicht. Hier könnte man zwar auch eine implizite Konvertierung verwenden, indem man einfach als Funktionsargument `int` statt `double` verwendet, aber hiermit soll deutlich werden, was der Programmierer mit dieser Funktion beabsichtigt und dass es kein *Versehen* war. Aus diesem Grund liefert der zweite Vergleich mit `std::equal()` auch jetzt `true` zurück, da beide Container denselben Inhalt haben.

12.5.4 Algorithmen mit einfachen unären Funktionen

Nicht immer wird gleich ein Prädikat für einen Vergleich benötigt. Es gibt auch Algorithmen, die eine Funktion, ein Funktionsobjekt oder eine Lambda-Funktion verwenden, die keinen Rückgabewert benötigen (wie bspw. der Algorithmus `std::for_each()`). Wird trotzdem ein Rückgabewert verwendet, so wird dieser ignoriert. Hierzu ein einfaches Beispiel mit dem Algorithmus `std::for_each()`:

```
00 // kapitel12/algorithm/function01.cpp
...
01 void print(double val) {
02     std::cout << val << std::endl;
03 }
...
04 std::deque<double> dat01{ 2.1, 5.5, 3.2, 4.1 };
05 for_each(dat01.begin(), dat01.end(), print);
```

In diesem Beispiel wird mit `std::for_each()` jedes Element vom Bereich `dat01.begin()` bis `dat01.end()` an die Funktion `print` übergeben. Die Funktion `print()` macht letztlich nichts anderes, als den Inhalt des Elements auf dem Bildschirm auszugeben. In dem kurzen Beispiel wird also der Inhalt des kompletten Containers ausgegeben.

12.5.5 Funktionsobjekte

Neben Funktionen können Sie auch Funktionsobjekte als Argumente an die Algorithmen übergeben. Ein Funktionsobjekt (auch *Funktor* genannt) ist im Grunde nichts anderes als eine Klasse, in der `operator()` überladen wurde. Der Vorteil von

Funktionsobjekten ist, dass diese noch weitere Eigenschaften besitzen können, und es stehen Konstruktoren und ein Destruktor zu Verfügung.

Zur Demonstration folgt hier ein Beispiel mit der Ausgabe aller Elemente auf dem Bildschirm, wie diese im Abschnitt zuvor in Listing *function01.cpp* verwendet wurden, nur eben jetzt mit einem Funktionsobjekt statt einer globalen Funktion:

```
00 // kapitel12/algorithm/functor01.cpp
...
01 class PrintObj {
02     int counter;
03 public:
04     PrintObj() {counter=0;}
05     ~PrintObj() {}
06     void operator() (const double& obj) {
07         std::cout << ++counter << " : ";
08         std::cout.width(4);
09         std::cout.setf(std::ios::left);
10         std::cout << obj << std::endl;
11     }
12 };
...
13 std::deque<double> dat01{ 2.11, 5.551, 3.2, 4.12 };
14 for_each(dat01.begin(),dat01.end(), PrintObj());
```

In der Klasse `PrintObj` finden Sie in den Zeilen **06** bis **11** die Implementierung von `operator()`. Daran können Sie hier auch gleich das Funktionsobjekt (Funktor) erkennen. Das Funktionsobjekt in unserem Beispiel kümmert sich um eine formatierte Ausgabe auf dem Bildschirm. In Zeile **14** wird dieses Funktionsobjekt als drittes Argument mit einem `double` als Argument aufgerufen. Die Ausgabe des Programms lautet:

```
1 : 2.11
2 : 5.551
3 : 3.2
4 : 4.12
```

Für solche Zwecke bietet sich die generische Programmierung an, um das Funktionsobjekt auch für andere Typen zur Verfügung zu stellen, bspw.:

```
00 // kapitel12/algorithm/functor02.cpp
...
01 template <typename T>
02 class PrintObj {
03     int counter;
04 public:
```

```

05     PrintObj() {counter=0;}
06     ~PrintObj() {}
07     void operator() (const T& obj) {
08         std::cout << ++counter << " : ";
09         std::cout.width(4);
10         std::cout.setf(std::ios::left);
11         std::cout << obj << std::endl;
12     }
13 };

```

Hier wird `operator()` jetzt mit dem formalen Parameter `T` implementiert, weshalb dieses Funktionenobjekt für jeden Typ verwendet werden kann. Bei eigenen Typen wie einer Klasse setzt dies dann allerdings auch voraus, dass `operator<<()` für die Ausgabe, wie in der (Zeile 11) verwendet, entsprechend überladen wurde. Hierzu ein paar einfache Beispiele, welche jetzt das Template-Funktionsobjekt verwenden:

```

00 // kapitel12/algorithm/functor02.cpp
...
01 class DataClass {
02     int data;
03 public:
04     DataClass( int d=0 ) : data(d) {}
05     int getData() const { return data; }
06     friend std::ostream& operator<<( std::ostream& os,
                                     const DataClass& dat );
07 };

08 std::ostream& operator<<( std::ostream& os,
                           const DataClass& dat ) {
09     os << dat.data;
10     return os;
11 }
...
12 std::deque<std::string> dat01{ "Hallo", "Welt", "!!!" };
13 for_each(dat01.begin(),dat01.end(),PrintObj<std::string>());
14 std::list<int> dat02{1, 4, 6, 7, 8, 10 };
15 for_each(dat02.begin(),dat02.end(), PrintObj<int>());
16 std::vector<DataClass> dat03;
17 dat03.push_back(DataClass(123));
18 dat03.push_back(DataClass(321));
19 for_each(dat03.begin(),dat03.end(), PrintObj<DataClass>());

```

Um eine entsprechendes Funktionsobjekt instanziiieren zu lassen, müssen Sie in spitzen Klammern, wie in den Zeilen 13, 15 und 19 zu sehen, den entsprechenden Typ

angeben. Damit sich die Klasse `DataClass` hierfür auch verwenden lässt, wurde `operator<<()` für die Ausgabe als `friend`-Funktion implementiert.

Fertige Funktionsobjekte

Die Standardbibliothek stellt ebenfalls viele fertige Funktionsobjekte zur Verfügung, die teilweise auch zum Einsatz kommen, wenn Sie kein Funktionsobjekt verwenden. Zum Beispiel:

```
std::sort( x.begin(), x.end() );
```

Da hier beim Algorithmus `std::sort` kein drittes explizites Argument verwendet wurde, das angibt, wie sortiert wird, wird hier standardmäßig in aufsteigender Reihenfolge sortiert. Intern wird hierfür das Funktionsobjekt `less<T>()` verwendet. `T` wird durch den entsprechenden Typ ersetzt.

Wollen Sie jetzt aber die Werte in absteigender Reihenfolge sortieren, können Sie entweder eine eigene Funktion oder ein eigenes Funktionsobjekt als drittes Argument erstellen und verwenden, oder aber Sie verwenden in diesem Beispiel die vorgefertigte Version der Standardbibliothek mit `greater<T>()` wie folgt:

```
std::sort( x.begin(), x.end(), greater<int>() );
```

In Tabelle 12.18 bis Tabelle 12.20 finden Sie Listen zu den Funktionsobjekten der Standardbibliothek.

Arithmetik	Entspricht
<code>plus<T></code>	<code>elem1+elem2</code>
<code>minus<T></code>	<code>elem1-elem2</code>
<code>multiplies<T></code>	<code>elem1*elem2</code>
<code>divides<T></code>	<code>elem1/elem2</code>
<code>modules<T></code>	<code>elem1%elem2</code>
<code>negate<T></code>	<code>-elem</code>

Tabelle 12.18 Arithmetische Funktionsobjekte

Vergleich	Entspricht
<code>equal_to<T></code>	<code>elem1==elem2</code>
<code>not_equal_to<T></code>	<code>elem1!=elem2</code>

Tabelle 12.19 Funktionsobjekt für den Vergleich

Vergleich	Entspricht
less<T>	elem1<elem2
less_equal<T>	elem1<=elem2
greater<T>	elem1>elem2
greater_equal<T>	elem1>=elem2

Tabelle 12.19 Funktionsobjekt für den Vergleich (Forts.)

Logik	Entspricht
logical_and<T>	elem1 && elem2
logical_or<T>	elem1 elem2
logical_not<T>	!elem

Tabelle 12.20 Logische Funktionsobjekte

Die meisten Funktionsobjekte der Standardbibliothek haben ein zweistelliges (binäres) Prädikat. Nur die Funktionsobjekte `negate<T>` und `logical_not<T>` sind ein einstelliges Prädikat.

Operatoren für eigene Typen überladen

Um allerdings die fertigen Funktionsobjekte für eigene Typen, wie bspw.

```
std::sort( dat03.begin(),dat03.end(), std::greater<DataClass>() );
```

zu verwenden, müssen Sie hierfür sinngemäß den passenden Operator überladen. Im Beispiel von `DataClass` könnten Sie hierfür `operator>()` wie folgt als friend-Funktion implementieren, damit das Funktionsobjekt `greater<T>` hier seine Arbeit auch für `T`, hier mit der Klasse `DataClass`, verrichten kann:

```
// kapitel12/algorithm/functor03.cpp
...
bool operator>(const DataClass& d1, const DataClass& d2) {
    return( d1.getData() > d2.getData());
}
```

12.5.6 Lambda-Funktionen (C++11)

Die Lambda-Funktionen wurden mit dem Hintergedanken eingeführt, kleinere Funktionen an Ort und Stelle zu schreiben und auch gleich zu verwenden. Damit können Sie eine Art inline-Funktionalität definieren, die wie ein lokales Objekt verwendet werden kann.

Der Vorteil von Lambda-Funktionen gegenüber Funktionsobjekten ist, dass Sie hiermit nicht extra eine Klasse definieren und diese auch noch instanziiieren müssen. Sie können eine Lambda-Funktion direkt verwenden.

Insgesamt kann eine Lambda-Funktion aus vier einzelnen Komponenten bestehen:

```
[ ] ( ) -> void {};
```

Hierbei handelt es sich in der Tat um einen funktionierenden Code. Schreiben Sie diese Zeile in die main-Funktion, wird diese Lambda-Funktion auch ausgeführt. In dem Fall wird allerdings gar nichts gemacht. Tabelle 12.21 verdeutlicht die Bedeutung der einzelnen Bestandteile einer Lambda-Funktion.

[]	()	-> typ	{ }
Bindung	Parameter	Rückgabewert	Funktionskörper

Tabelle 12.21 Die Notation einer Lambda-Funktion und ihre Bestandteile. Die Bestandteile Parameter und Rückgabewert sind optional und können auch weggelassen werden.

Eine leere Hallo-Lambda-Funktion könnte somit wie folgt definiert werden:

```
00 // kapitel12/algorithm/lambda01.cpp
...
01 int main() {
02     [ ] {
03         std::cout << "Hallo Lambda\n";
04     } ();
05 }
```

Diese Lambda-Funktion gibt »Hallo Lambda« auf dem Bildschirm aus. Eingeleitet wird sie mit einem leeren []. Dahinter folgt der Funktionskörper zwischen { }, in die die Anweisungen wie bei einer gewöhnlichen Funktion geschrieben werden. Am Ende sind noch die runden Klammern () nötig, damit diese Lambda-Funktion auch aufrufbar ist. »Aufrufbar« zu sein mit runden Klammern kennen Sie ja schon zur Genüge von den Funktionen, Methoden, Funktionsobjekten und jetzt noch den Lambda-Funktionen.

Rückgabewert

Aus Lambda-Funktionen können auch Werte zurückgegeben und außerhalb der Lambda-Funktion zugewiesen werden. Wenn Sie hierbei den Rückgabebetyp nicht angeben, ist der Compiler schlau genug, diesen Typ aus `return` selbst abzuleiten, bspw.:

```
00 // kapitel12/algorithm/lambda02.cpp
...
01 int main() {
02     auto val = [] {
03         return 1234.56;
04     } ();
05     std::cout << val << std::endl; // = 1234.56
06 }
```

Hier kann der Compiler aus dem Rückgabewert von `return` selbst ermitteln, um welchen Typ (hier `double`) es sich dabei handelt. Der zurückgegebene Wert wird hier an `auto val` übergeben und in Zeile **05** auf dem Bildschirm ausgegeben.

Sie können aber auch den Rückgabewert mit der neuen C++11-Syntax `-> typ` selbst festlegen. Bezogen auf unser Beispiel würde sich dies bspw. folgendermaßen verwenden lassen:

```
00 // kapitel12/algorithm/lambda03.cpp
...
01 int main() {
02     auto val = [] () -> int {
03         return 1234.56;
04     } ();
05     std::cout << val << std::endl; // = 1234
06 }
```

Hier wird jetzt ein `int` zurückgegeben. Der Wert 1234.56 wird daher implizit in ein `int` konvertieren, weshalb die Ausgabe in Zeile **05** auch 1234 ist. Die runden leeren Parameterklammern vor `-> typ` müssen Sie hier mit angeben.

Bindung []

Die eckigen Klammern `[]` leiten zum einen die Lambda-Funktion ein, und zum anderen kann auch gleich die Bindung einer Variablen im lokalen Bereich (Funktionskörper) von außerhalb der Lambda-Funktion angegeben werden. Eine leeres `[]` bedeutet keine Bindung. Mit `[=]` werden die Werte außerhalb der Lambda-Funktion als Kopie (*Call by Value*) für die Lambda-Funktion übergeben. Die Werte können daher innerhalb der Lambda-Funktion nicht direkt geändert werden. Die Verwendung von `[&]`

hingegen sorgt dafür, dass die Werte außerhalb der Lambda-Funktion als Referenz (*Call by Reference*) an die Lambda-Funktion weitergegeben werden. Den Unterschied zwischen Übergabe als Wert und Übergabe als Referenz kennen Sie ja bereits von den Funktionen und Methoden – bei den Lambdas haben Call by Value und Call by Reference dieselbe Bedeutung.

Sie können aber auch ganz individuell Werte in den eckigen Klammern angeben, welche als Kopie und/oder als Referenz an die Lambda-Funktion gehen sollen. Folgendes Beispiel soll dies kurz demonstrieren:

```
00 // kapitel12/algorithm/lambda04.cpp
...
01 int main() {
02     int val_X = 123;
03     double val_Y = 54.32;
04     [val_X, &val_Y] {
05         //val_X+=100;
06         val_Y+=100;
07         std::cout << val_X << std::endl; // 123
08         std::cout << val_Y << std::endl; // 154.32
09     }();
10     std::cout << val_Y << std::endl;      // 154.32
11 }
```

In Zeile **04** übergeben Sie `val_X` als Kopie und `val_Y` als Referenz an die Lambda-Funktion. Eine Änderung innerhalb der Lambda-Funktion einer Kopie ist daher, wie in Zeile **05** auskommentiert, so nicht möglich. Hier funktioniert nur der lesende Zugriff, wie in Zeile **07** zu sehen ist. Bei der Übergabe als Referenz, wie Sie mit `&val_Y` in Zeile **04** sehen können, kann auch schreibend auf den Wert (siehe Zeile **06**) zugegriffen werden. Wie für eine Übergabe als Referenz üblich, hat die Änderung innerhalb der Lambda-Funktion auch Auswirkungen auf den Wert außerhalb (ist ja auch eine Referenz auf dem Wert), weshalb die Änderungen innerhalb der Lambda-Funktion auch außerhalb in Zeile **10** ausgegeben werden.

In diesem Beispiel hätten Sie Zeile **04** alternativ auch wie folgt mit `[=, &val_Y]` formulieren können. Hiermit würden alle Werte als Kopie übergeben und `val_Y` als Referenz. Wenn Sie allerdings `[=, ...]` verwenden, können Sie kein weiteres Element als Kopie übergeben, und die restlichen Elemente müssen mit `&` beginnen.

Die Zugriffsdeklarationen lassen sich durchaus mischen. Ein `[&, val_Y]` würde alle Elemente als Referenz übergeben und `val_Y` als Kopie. Auch gilt allerdings, wenn Sie `[&, ...]` verwenden, dürfen keine weiteren Elemente mit einem `&` mehr folgen. Sie können nur noch Kopien dahinterschreiben.

Wenn es mehrere Werte gibt, die Sie von außen an die Lambda-Funktion weitergeben wollen, können Sie diese Bindungsliste durchaus beliebig erweitern und mischen (bspw. `[x, &y, z, v]`).

Empfehlungen für die Bindung

Auch hier gilt dasselbe wie schon für Funktionen. Bei einer Übergabe als Kopie (Call by Value) ist der Rechenaufwand größer, weil hier für jedes Element eine Kopie erstellt werden muss. Bei der Verwendung von Algorithmen mit Container und einer umfangreichen Datensammlung kann dies ein ziemlicher Rechenaufwand werden. Wenn Sie Werte innerhalb einer Lambda-Funktion ändern müssen, kommen Sie um eine Referenz nicht herum.

mutable

Wollen Sie bei der Übergabe von Werten als Kopie von außerhalb der Lambda-Funktion nach innen auf diese Kopie schreibend zugreifen, müssen Sie die Lambda-Funktion zu `mutable` (= veränderlich) erklären. Hier das Beispiel dazu:

```
00 // kapitel12/algorithm/lambda05.cpp
...
01 int main() {
02     int val_X = 123;
03     double val_Y = 54.32;
04     [val_X, &val_Y] () mutable {
05         val_X+=100;
06         val_Y+=100;
07         std::cout << val_X << std::endl; // 223
08         std::cout << val_Y << std::endl; // 154.32
09     };
10     std::cout << val_X << std::endl; // 123
11     std::cout << val_Y << std::endl; // 154.32
12 }
```

Durch das Schlüsselwort `mutable` (die runden Klammern davor sind hier auch nötig) in Zeile **04** ist jetzt ein schreibender Zugriff auf Werte möglich, die als Kopie der Lambda-Funktion übergeben wurden. Im Beispiel ist jetzt der Wert von `val_X` in Zeile **05** veränderbar, was die Ausgabe der Zeile **07** auch bestätigt. Aber eine Übergabe als Kopie bleibt innerhalb der Lambda-Funktion eine Kopie, weshalb eine Änderung innerhalb der Lambda-Funktion keine Auswirkung außerhalb hat, wie die Ausgabe von Zeile **10** demonstrieren soll.

Lambda-Funktion aufrufen

Eine Lambda-Funktion lässt sich auch via `auto` einem Bezeichner zuweisen und aufrufen, bspw.:

```
00 // kapitel12/algorithm/lambda06.cpp
...
01 int main() {
02     auto lf = [] {
03         std::cout << "Hallo Lambda\n";
04     };
...
05     lf(); // Führt Lambda-Funktion aus
06 }
```

In Zeile **02** bekommt `lf` die Lambda-Funktion zugewiesen und in Zeile **05** rufen wir diese wie eine gewöhnliche Funktion auf.

Das Prinzip funktioniert auch mit Parameter:

```
00 // kapitel12/algorithm/lambda07.cpp
...
01 int main() {
02     auto lf = [] (int x, int y) {
03         std::cout << x + y << std::endl;
04     };
...
05     lf(10, 20); // 30
06     lf(100, 123); // 223
07 }
```

Hier bekommt `lf` nochmals eine Lambda-Funktion zugewiesen, nur dieses Mal mit zwei Parametern, welche addiert werden. In den Zeilen **05** und **06** können Sie jeweils einen Aufruf davon sehen.

Lambdas und die Standardbibliothek

Lambda-Funktionen wurden nicht für Spielereien eingeführt, wie bisher gezeigt wurde. Vielmehr sollen die neuen Lambda-Funktionen als nützliche Helfer bei der Standardbibliothek für die Algorithmen bereitstehen, um bspw. nötige Funktionen oder Prädikate gleich an Ort und Stelle zu schreiben. Und hierzu auch gleich wieder der Klassiker mit der Ausgabe aller Elemente eines Containers:

```
00 // kapitel12/algorithm/lambda08.cpp
...
01 std::deque<double> dat01{ 2.1, 5.5, 3.2, 4.1 };
```

```

02  for_each(dat01.begin(), dat01.end(),
           [](double& val) {
               std::cout << val << std::endl;
           })
    ); // for_each
...

```

Die Lambda-Funktion mit dem Parameter `double&` wird vom Algorithmus `for_each()` für jedes Element aus dem Bereich `dat01.begin()` bis `dat02.end()` aufgerufen.

Der Vorteil ist hier ganz klar, Sie können alles an Ort und Stelle schreiben und anpassen. Sie müssen nicht umständlich herauf- bzw. herunterscrollen oder gar in einer anderen Datei nach einer globalen Funktion oder einem Funktionsobjekt suchen.

Aber es gibt noch mehr Vorteile. Im folgenden Beispiel soll nochmals unsere einfache Klasse `DataClass` verwendet werden, um den Algorithmus `std::sort()` zu verwenden. Hierfür war es ja nötig, den entsprechenden Operator (bspw. `operator<()`) zu überladen, damit das auch bei eigenen Typen funktioniert. Bei der Lambda-Funktion können Sie jetzt darauf verzichten und stattdessen den Vergleich gleich direkt in der Lambda-Funktion vornehmen. Hier ein Beispiel dazu:

```

00  // kapitel12/algorithm/lambda09.cpp
...
01  class DataClass {
02      int data;
03  public:
04      DataClass( int d=0 ) : data(d) {}
05      int getData() const { return data; }
06  };
...
07  std::vector<DataClass> dat;
08  dat.emplace_back(123);
09  dat.emplace_back(100);
10  dat.emplace_back(111);
11  dat.emplace_back(99);
12  std::sort(dat.begin(), dat.end(),
           [](const DataClass& d1, const DataClass& d2) {
               return(d1.getData() < d2.getData());
           })
    ); // std::sort()
13  for(const auto& elem : dat) {
14      std::cout << elem.getData() << std::endl;
15  }
...

```

Nachdem wir dem Container ein paar Elemente hinzugefügt haben, sortieren wir die einzelnen Elemente in aufsteigender Reihenfolge. Hierzu wird ja ein binäres Prädikat benötigt, welches wir in Zeile 12 gleich als Lambda-Funktion integriert haben. Hierbei werden immer jeweils zwei Elemente vom Bereich `dat.begin()` bis `dat.end()` in der Lambda-Funktion verglichen. Je nach Vergleich liefert die Lambda-Funktion gleich `true` oder `false` zurück – also ein `bool` und genau das, was der Algorithmus `std::sort()` zum Sortieren benötigt. Das war es auch schon. Um mehr brauchen wir uns hier nicht zu kümmern. Das Beispiel zeigt sehr schön, dass Sie mit Lambda-Funktionen jederzeit andere Funktionen oder Methoden aufrufen können.

Wem die Schreibweise direkt in dem Algorithmus zu unübersichtlich erscheint, der kann die Lambda-Funktion vorher definieren und per `auto` einem Bezeichner zuweisen und diesen Bezeichner dann in den Algorithmus verpacken. Das würde dann wie folgt aussehen:

```
01 auto predicat= [] ( const DataClass& d1,
                      const DataClass& d2 ){
    return(d1.getData() < d2.getData());
};
02 std::sort(dat.begin(), dat.end(), predicat);
```

Empfehlungen zu Lambda-Funktionen

An sich ist es kein Problem, noch weitaus komplexere Lambda-Funktionen zu schreiben. Aber in diesem Fall geht dies dann eigentlich an dem Sinn von Lambda-Funktionen vorbei. Lambda-Funktionen wurden eingeführt, um das Programmiererleben einfacher zu machen. Daher die Empfehlung, nur dann Lambda-Funktionen zu verwenden, wo der Code kurz und knapp aus wenigen Anweisungen besteht – am besten nur einer Anweisung. Außerdem sollte der Code der Lambda-Funktion selbsterklärend und nicht kryptisch sein.

12.5.7 Übersicht zu den Algorithmen

Nachdem Sie wissen, was Algorithmen sind und wie Sie diese verwenden können, folgt jetzt eine Übersicht zu den vorhandenen Algorithmen der Standardbibliothek. Für fast alle Algorithmen gibt es eine Version mit und eine ohne Prädikat. Der Umfang der vorhandenen Algorithmen ist beachtlich, weshalb Sie hier nur eine Übersicht dazu erhalten. Die meisten der Algorithmen sind in der Headerdatei `<algorithm>` definiert, weshalb diese mit eingebunden werden muss.

Für eine detailliertere Syntaxbeschreibung empfehle ich Ihnen entweder die Manual-Seiten oder die Dokumentation Ihres Compilers. Im Web gibt es hier auch

interessante komplette Referenzen der Standardbibliothek auf Seiten, wie z.B.: <http://en.cppreference.com>.

Nicht modifizierende Algorithmen

Bei diesen Algorithmen wird nur lesend auf die Container (bzw. Datenstrukturen) zugegriffen, und es wird nichts verändert (Tabelle 12.22). Diese Algorithmen lassen sich auf alle Arten von Containern anwenden.

Algorithmus	Beschreibung
all_of(i1, i2, pred) any_of(i1, i2, pred) none_of(i1, i2, pred)	Neu in C++11 : Überprüft, ob das unäre Prädikat <code>pred</code> gleich <code>true</code> für alle (<code>all_of</code>), einige bzw. mindestens ein (<code>any_of</code>) oder kein (<code>none_of</code>) Element im Bereich <code>i1</code> bis <code>i2</code> zurückliefert.
for_each (i1, i2, func)	Führt für jedes einzelne Element aus dem Bereich <code>i1</code> bis <code>i2</code> die Funktion <code>func</code> aus (kann auch ein Funktionsobjekt oder eine Lambda-Funktion sein).
find (i1, i2, val)	Sucht im Bereich <code>i1</code> bis <code>i2</code> nach einem Element mit dem Wert <code>val</code> .
find_if (i1, i2, pred)	Sucht im Bereich <code>i1</code> bis <code>i2</code> nach einem Element, welches der Bedingung <code>pred</code> (= <code>true</code>) entspricht.
find_if_not(i1, i2, pred)	Neu in C++11 : Sucht im Bereich <code>i1</code> bis <code>i2</code> nach einem Element, welches der Nicht -Bedingung <code>pred</code> (= <code>false</code>) entspricht.
find_end (i1,i2, j1, j2) find_end(i1,i2,j1,j2,pred)	Sucht im Bereich <code>i1</code> bis <code>i2</code> nach einer letzten passenden Teilfolge vom Bereich <code>j1</code> bis <code>j2</code> oder dass eben das Prädikat <code>pred</code> wahr ist.
find_first_of (i1,i2,j1,j2) find_first_of(i1,i2, j1,j2,pred)	Wie <code>find_end()</code> , nur dass nach dem ersten Auftreten mehrerer Elemente gesucht wird.
adjacent_find (i1,i2) adjacent_find(i1,i2,pred)	Sucht nach benachbarten Elementen zwischen <code>i1</code> und <code>i2</code> , deren Werte gleich sind bzw. wo das Prädikat <code>pred</code> gleich <code>true</code> zurückliefert.
count (i1,i2,val) count_if (i1, i2, pred)	Gibt die Anzahl der Elemente aus dem Bereich <code>i1</code> bis <code>i2</code> zurück, die den Wert <code>val</code> haben oder wo das Prädikat <code>pred</code> gleich <code>true</code> zurückgibt.

Tabelle 12.22 Algorithmen, welche den Container nicht verändern

Algorithmus	Beschreibung
<code>mismatch (i1,i2,j1)</code> <code>mismatch(i1,i2,j1,pred)</code>	Liefert die erste Position zurück, an der der Bereich <code>i1</code> bis <code>i2</code> ab <code>j1</code> bis zum Ende nicht übereinstimmt oder das Prädikat <code>pred</code> gleich <code>true</code> zurückgibt. Zurückgegeben wird die Template-Klasse <code>pair</code> , womit die erste Nicht-Übereinstimmung von <code>i1</code> bis <code>i2</code> in <code>first</code> und von <code>j1</code> in <code>second</code> enthalten ist.
<code>equal (i1,i2,j1)</code> <code>equal(i1,i2,j1,pred)</code>	Vergleicht <code>i1</code> bis <code>i2</code> ab dem Bereich <code>j1</code> auf Gleichheit, oder das Prädikat <code>pred</code> liefert <code>true</code> zurück, wenn beide Bereiche gleich sind.
<code>search (i1,i2,j1,j2)</code> <code>search(i1,i2,j1,j2,pred)</code>	Sucht im Bereich <code>i1</code> bis <code>i2</code> nach der Teilfolge <code>j1</code> bis <code>j2</code> nach einer Übereinstimmung oder wo das Prädikat <code>pred</code> gleich <code>true</code> zurückgibt.
<code>search_n (i1,i2,n,val)</code> <code>search_n(i1,i2,n,val,pred)</code>	Sucht im Bereich <code>i1</code> bis <code>i2</code> mit der Länge <code>n</code> nach einer Teilfolge mit dem Wert <code>val</code> oder wo das Prädikat <code>pred</code> gleich <code>true</code> zurückgibt.
<code>min_element (i1,i2)</code> <code>max_element(i1,i2)</code>	Liefert das kleinste bzw. größte Element aus dem Bereich <code>i1</code> bis <code>i2</code> zurück.

Tabelle 12.22 Algorithmen, welche den Container nicht verändern (Forts.)

Rückgabewert von Suchfunktionen
Die Suchfunktionen geben immer die Position des gefundenen Elements zurück, oder, wenn die Suche erfolglos war, das Ende des gesuchten Bereichs.

Modifizierende Algorithmen

Die modifizierenden Algorithmen greifen direkt auf die Elemente der Container zu, wodurch die Elemente geändert werden könnten (Tabelle 12.23). Als Zielbereich sind bei diesen Algorithmen keine assoziativen Container erlaubt.

Ausreichend Platz vorhanden?
Wenn Sie etwas in einen Zielbereich verschieben oder kopieren, müssen Sie darauf achten, dass dort auch ausreichend Speicherplatz dafür vorhanden ist. Die Algorithmen nehmen Ihnen hier nicht die Arbeit der Speicherverwaltung ab, wie Sie dies von den Methoden der Container her gewohnt sind.

Algorithmus	Beschreibung
<code>copy (i1,i2,o1)</code> <code>copy_backward (i1,i2,o1)</code>	Kopiert die Elemente vom Bereich <code>i1</code> bis <code>i2</code> in den Zielbereich, der mit <code>o1</code> beginnt. <code>copy_backward()</code> macht dasselbe, nur rückwärts und dass der Zielbereich hier mit <code>o1</code> endet(!).
<code>copy_if(i1,i2,o2,pred)</code>	Neu in C++11 : Kopiert nur die Elemente vom Bereich <code>i1</code> bis <code>i2</code> in den Zielbereich <code>o2</code> , wo das Prädikat <code>pred</code> gleich <code>true</code> ist.
<code>copy_n(i1, n, o1)</code>	Neu in C++11 : Kopiert <code>n</code> Elemente vom Bereich, der mit <code>i1</code> beginnt, in den Bereich, der mit <code>o2</code> beginnt.
<code>swap_ranges (i1,i2,o1)</code>	Damit tauschen Sie die Elemente im Bereich <code>i1</code> bis <code>i2</code> mit den Elementen, die bei <code>o1</code> beginnen.
<code>iter_swap (i1, o1)</code>	Tauscht den Inhalt, auf den <code>i1</code> verweist, mit dem Inhalt aus <code>o1</code> .
<code>transform (i1,i2,o1,pred)</code>	Damit wird für jedes Element im Bereich <code>i1</code> bis <code>i2</code> die unäre Funktion <code>pred</code> aufgerufen. Speichert den Rückgabewert im Zielbereich <code>o1</code> . Quell- und Zielbereich dürfen gleich sein!
<code>transform(i1,i2,j2,o1,pred)</code>	Ruft für jedes Element vom Bereich <code>i1</code> bis <code>i2</code> und dem Element ab <code>j2</code> die binäre Funktion <code>pred</code> auf und weist den Rückgabewert an <code>o1</code> zu.
<code>replace (i1,i2,oval,nval)</code> <code>replace_if (i1,i2,pred,nval)</code>	Damit ersetzen Sie in den Bereichen <code>i1</code> bis <code>i2</code> die Elemente mit dem Wert <code>oval</code> bzw. mit dem Wert <code>nval</code> , wenn das Prädikat <code>pred</code> gleich <code>true</code> zurückgibt.
<code>replace_copy (i1,i2,o2, oval,nval)</code> <code>replace_copy_if (i1,i2,o2, pred,nval)</code>	Wie <code>replace()</code> bzw. <code>replace_if()</code> , nur wird zusätzlich noch in den Zielbereich <code>o2</code> kopiert. Wobei sich hier Quell- und Zielbereich nicht überlappen dürfen.
<code>fill (i1, i2, val)</code> <code>fill_n (i1, n, val)</code>	Füllt alle Elemente im Bereich <code>i1</code> bis <code>i2</code> mit dem Wert <code>val</code> , oder im Fall von <code>fill_n()</code> werden ab <code>i1</code> insgesamt <code>n</code> Elemente mit dem Wert <code>val</code> gefüllt.

Tabelle 12.23 Modifizierende Algorithmen

Algorithmus	Beschreibung
<code>generate (i1,i2,func)</code> <code>generate_n (i1,n,func)</code>	Wie eben <code>fill()</code> bzw. <code>fill_n()</code> , nur wird statt eines Wertes der Bereich <code>i1</code> bis <code>i2</code> bzw. <code>n</code> Elemente ab <code>i1</code> mit dem Rückgabewert der Funktion <code>func</code> gefüllt.
<code>move(i1, i2, o1)</code> <code>move_backward(i1, i2, o1)</code>	Neu in C++11 : Verschiebt die Elemente vom Bereich <code>i1</code> bis <code>i2</code> in den Zielbereich, der mit <code>o1</code> beginnt. <code>move_backward()</code> macht dasselbe, nur rückwärts und dass der Zielbereich mit <code>o1</code> endet(!).

Tabelle 12.23 Modifizierende Algorithmen (Forts.)

Löschende Algorithmen

Die Gruppe von löschenden Algorithmen lässt sich auch zu den modifizierenden Algorithmen hinstellen (Tabelle 12.24). Es ist allerdings wichtig, zu wissen, dass hiermit zwar Elemente aus dem Container entfernt werden, aber der Container damit nicht automatisch »kleiner« wird, sondern nur die gelöschten Elemente überschrieben werden. Da es ja auch modifizierende Algorithmen sind, können diese auch nicht auf assoziative Container angewendet werden.

Echtes Löschen

Wollen Sie überschüssige Elemente nach einem löschenden Algorithmus endgültig löschen, so können Sie dies mit der Methode `erase()` des entsprechenden Containers machen.

Algorithmus	Beschreibung
<code>remove (i1,i2,val)</code> <code>remove_if (i1,i2,pred)</code>	Löscht alle Elemente im Bereich <code>i1</code> bis <code>i2</code> , die den Wert <code>val</code> haben oder wo das Prädikat <code>pred</code> gleich <code>true</code> zurückgibt.
<code>remove_copy (i1,i2,o1,val)</code> <code>remove_copy_if(i1,i2,o1,pred)</code>	Wie <code>replace_copy()</code> und <code>replace_copy_if()</code> , nur wird die <code>remove</code> -Version zum Löschen statt zum Ersetzen verwendet.
<code>unique (i1, i2)</code> <code>unique_copy (i1,i2,o1)</code>	Löscht alle Elemente, deren Vorgänger denselben Wert besitzt. <code>unique_copy()</code> hingegen kopiert die Elemente noch in den Bereich <code>o2</code> .

Tabelle 12.24 Löschende Algorithmen

Wo sind die gelöschten Elemente?

Die von den löschenden Algorithmen gelöschten Elemente werden nach hinten geschoben. Hierbei kann es hilfreich sein, dass die `remove`-Funktionen einen Iterator auf das erste gelöschte Element zurückgeben. Somit gilt, dass der Bereich des zurückgegebenen Iterators bis `end()` ungültig ist und bspw. mit dem `erase()` des entsprechenden Containers entfernt werden kann.

Mutierende Algorithmen

Bei mutierenden Algorithmen ändert sich nur die Reihenfolge der Elemente im Behälter (Tabelle 12.25). Im Grunde handelt es sich damit auch um modifizierende Algorithmen, und daher können sie auch nicht auf assoziative Container angewendet werden.

Algorithmus	Beschreibung
<code>rotate (i1,i2,i3)</code> <code>rotate_copy (i1,i2,i3,o1)</code>	Damit führen Sie eine Rotation im Bereich <code>i1</code> und <code>i3</code> durch, und <code>i2</code> ist anschließend das erste Element. Mit <code>rotate_copy()</code> wird das Ergebnis in <code>o1</code> kopiert.
<code>random_shuffle (i1,i2)</code> <code>random_shuffle (i1,i2,func)</code>	Ändert die Reihenfolge der Elemente im Bereich <code>i1</code> bis <code>i2</code> zufällig oder anhand der Funktion <code>func</code> .
<code>reverse (i1,i2)</code> <code>reverse_copy (i1,i2,o2)</code>	Damit drehen Sie die Reihenfolge vom Bereich <code>i1</code> bis <code>i2</code> um. <code>reverse_copy()</code> macht das Gleiche, aber kopiert diesen Bereich nach <code>o2</code> .
<code>partition (i1,i2,pred)</code> <code>stable_partition (i1,i2,pred)</code>	Damit zerlegen Sie den Bereich <code>i1</code> bis <code>i2</code> anhand von <code>pred</code> in zwei Bereiche: Ein Bereich liefert <code>true</code> zurück, und der zweite Bereich wird als Rückgabewert zurückgegeben, worauf Sie mit einem Iterator zugreifen können. Bei <code>stable_partition</code> bleibt die relative Reihenfolge erhalten.
<code>next_permutation (i1,i2)</code> <code>prev_permutation (i1,i2)</code>	Damit können Sie die nächste oder vorhergehende Permutation (Kombination) der Elemente im Bereich <code>i1</code> bis <code>i2</code> erzeugen.

Tabelle 12.25 Mutierende Algorithmen

Sortierende Algorithmen

Bei den sortierenden Algorithmen wird die Reihenfolge der Elemente geändert. Sortiert wird entweder mit dem Standardkriterium (`less<>`) oder einem Prädikat, wie einer benutzerdefinierten Funktion, einem Funktor (Funktionsobjekt) oder einer Lambda-Funktion (Tabelle 12.26). Auch diese Algorithmen lassen sich nicht auf assoziative Container anwenden.

Algorithmus	Beschreibung
<code>sort (i1,i2)</code> <code>sort(i1,i2,pred)</code> <code>stable_sort (i1,i2)</code> <code>stable_sort(i1,i2,pred)</code>	Damit sortieren Sie den Bereich <code>i1</code> bis <code>i2</code> standardmäßig aufsteigend (<code>less<></code>) oder gemäß dem Prädikat <code>pred</code> . Bei <code>stable_sort()</code> bleibt die relative Reihenfolge gleicher Elemente erhalten.
<code>is_sorted(i1,i2)</code> <code>is_sorted(i1,i2,comp)</code>	Neu in C++11 : Überprüft, ob der Bereich <code>i1</code> bis <code>i2</code> in aufsteigender Reihenfolge vorliegt und gibt <code>true</code> zurück, wenn dies der Fall ist, bzw. <code>false</code> , wenn nicht. Optional können Sie mit <code>comp</code> auch eine andere Vergleichsfunktion verwenden.
<code>partial_sort (i1,i2,i3)</code> <code>partial_sort(i1,i2,i3,pred)</code>	Sortiert die Elemente aufsteigend im Bereich <code>i1</code> bis <code>i3</code> , bis die Bereiche <code>i1</code> bis <code>i2</code> sortiert sind, oder anhand des Prädikats <code>pred</code> .
<code>partial_sort_copy (i1,i2 o1,o2)</code> <code>partial_sort_copy(i1,i2, o1,o2,pred)</code>	Sortiert die Elemente im Bereich <code>i1</code> bis <code>i2</code> und speichert das Ergebnis im Bereich <code>o1</code> bis <code>o2</code> . Ist der Bereich <code>o1</code> bis <code>o2</code> kleiner als <code>i1</code> bis <code>i2</code> , wird eben nur entsprechend partiell sortiert, was in <code>o1</code> bis <code>o2</code> passt. Alternativ gibt es auch noch eine Version mit dem Prädikat.
<code>nth_element (i1,inth,i2)</code> <code>nth_element(i1,inth,i2,pred)</code>	Sortiert die Elemente so, dass <code>inth</code> tatsächlich an der Position ist, wo es bei einem normalen <code>sort()</code> stehen würde. Im Bereich <code>i1</code> bis <code>inth</code> hingegen sind nur Elemente, die kleiner, und zwischen <code>inth</code> und <code>i2</code> die Elemente, die größer als <code>inth</code> sind. Mit dem Prädikat <code>pred</code> nehmen Sie die Sortierreihenfolge selbst in die Hand.

Tabelle 12.26 Sortierende Algorithmen

Algorithmen für binäre Suche (auf sortierte Bereiche)

Bei diesen Algorithmen handelt es sich um Operationen zur binären Suche nach Elementen (Tabelle 12.27). Sinngemäß sollte der Container schon in einer sortierten Reihenfolge vorliegen, damit die Suchfunktionen richtig arbeiten können. Diese Algorithmen ändern die Elemente im Container nicht.

Algorithmus	Beschreibung
<code>lower_bound (i1,i2,val)</code> <code>upper_bound (i1,i2,val)</code>	Gibt die Position im Bereich <code>i1</code> bis <code>i2</code> zurück, wo der Wert kleiner (<code>lower</code>) bzw. größer (<code>upper</code>) als <code>val</code> ist.
<code>equal_range (i1,i2,val)</code>	Gibt ein <code>pair</code> mit zwei Iteratoren zurück, welche die Position von <code>lower_bound</code> in <code>first</code> und <code>upper_bound</code> in <code>second</code> ausgehend vom Wert <code>val</code> zurückgeben.
<code>binary_search (i1,i2,val)</code>	Sucht im Bereich <code>i1</code> bis <code>i2</code> nach <code>val</code> und gibt <code>true</code> zurück, falls es gefunden wurde! Ansonsten wird <code>false</code> zurückgegeben.

Tabelle 12.27 Algorithmen zur binären Suche

Weitere Algorithmen (auf sortierte Bereiche)

Hier finden Sie weitere Algorithmen zum Mischen und Bilden neuer Bereiche aus bereits vorhandenen Bereichen, die in sortierter Reihenfolge vorliegen (Tabelle 12.28).

Algorithmus	Beschreibung
<code>merge (i1,i2,j1,j2,o1)</code>	Kopiert die sortierten Bereiche <code>i1</code> bis <code>i2</code> und <code>j1</code> bis <code>j2</code> in den Zielbereich <code>o1</code> , so dass dieser ebenfalls sortiert ist. Der Zielbereich darf sich hier nicht mit dem Quellbereich überlappen.
<code>set_union (i1,i2,j1,j2,o1)</code>	Bildet aus dem Bereich <code>i1</code> bis <code>i2</code> und <code>j1</code> bis <code>j2</code> eine Vereinigungsmenge in <code>o1</code> , wo es dann keine doppelten Elemente gibt.
<code>set_intersection (i1,i2, j1,j2,o1)</code>	Wie <code>set_union()</code> , nur wird eine Schnittmenge gebildet. Das bedeutet, es werden nur die Elemente in <code>o1</code> gesichert, welche in beiden Teilbereichen vorhanden sind.
<code>set_difference (i1,i2, j1,j2,o1)</code>	Bildet eine Differenzmenge aus den beiden Teilbereichen und speichert diese in <code>o1</code> . Diese Menge wird gebildet aus den Elementen, die im Bereich <code>i1</code> bis <code>i2</code> vorhanden sind, aber nicht im Bereich <code>j1</code> bis <code>j2</code> .
<code>set_symetric_difference(i1,i2, j1,j2,o1)</code>	Jetzt werden nur noch die Elemente in <code>o1</code> gesichert, die nicht im ersten und zweiten Teilbereich vorkommen.

Tabelle 12.28 Weitere Operationen auf sortierte Bereiche

Operationen für einen Heap

Bei einem Heap wird, vereinfacht ausgedrückt, aus einem sequenziellen Behälter ein binärer Baum (Tabelle 12.29). Natürlich nicht wirklich, aber das Verhalten des Behälters ahmt einen Baum nach. Daher ist anschließend beim sequenziellen Behälter, der ein Baum »sein soll«, das wichtigste Element gleich das erste. Das Prinzip ist dann das einer Prioritätsschlange, wie Sie dies von `priority_queue` her kennen.

Algorithmus	Beschreibung
<code>make_heap (i1,i2)</code> <code>make_heap(i1,i2,pred)</code>	Macht aus dem Bereich <code>i1</code> bis <code>i2</code> einen Heap! Ein Heap ist eine spezielle Datenstruktur, die als Baum implementiert ist.
<code>pop_heap (i1,i2)</code> <code>pop_heap(i1,i2,pred)</code> <code>push_heap (i1,i2)</code> <code>push_heap(i1,i2,pred)</code>	Mit <code>pop_heap</code> entfernen Sie das erste Element im Heap und fügen es als letztes Element wieder ein. Mit <code>push_heap</code> ist es genau umgekehrt.
<code>sort_heap (i1,i2)</code>	Sortiert die Elemente <code>i1</code> bis <code>i2</code> des Heaps und verwendet dabei den sehr effektiven Heap-Sort-Algorithmus. Anschließend ist dieser Bereich kein Heap mehr.
<code>is_heap(i1,i2)</code> <code>is_heap(i1,i2,comp)</code>	Überprüft, ob der Bereich <code>i1</code> bis <code>i2</code> ein Heap in aufsteigender Reihenfolge ist. Mit der zweiten Version können Sie mit <code>comp</code> eine eigene Vergleichsfunktion verwenden.

Tabelle 12.29 Heap-Operationen

12.6 Hilfsmittel

Im C++-Standard finden Sie viele Hilfsmittel, welche im C++-Hausgebrauch mal mehr und mal weniger benötigt werden. Hierbei handelt es sich meistens um kleinere und einfache Klassen, Typen oder Funktionen. In der Vergangenheit waren diese Hilfsmittel meistens ausschließlich in der Headerdatei `<utility>` und andere wiederum in `<algorithm>` zusammengefasst. Mittlerweile bietet der C++-Standard viele weitere solcher Hilfsmittel in verschiedenen Headerdateien an. In diesem Kapitel sollen Sie einen kleinen Überblick über die wichtigsten Hilfsmittel von C++ bekommen.

12.6.1 Template pair

Bei den assoziativen Containerklassen (siehe Abschnitt 12.2.2, »Assoziative Containerklassen«) haben Sie bereits gesehen, wie Sie das `Template pair` verwenden können. Das `Template` dient dazu, eine Kombination von zwei verschiedenen Wertpaaren zu

verwenden – auch als *heterogenes Wertpaar* bezeichnet. Die Datentypen können hierbei beliebig sein (also auch – und vor allem – Objekte von Klassen). Für das Template `pair` benötigen Sie die Headerdatei `<utility>` und gegebenenfalls den Namensraum `std` (also `std::pair`).

Die Syntax zur Definition eines `pair`-Objekts ist im Grunde immer gleich aufgebaut. Nach dem Datentyp `pair` geben Sie die beiden Typen in spitzen Klammern an, die im `pair`-Objekt abgelegt werden sollen. Dahinter folgt der Bezeichner. Folgt hinter dem Bezeichner nichts mehr, werden die beiden Typen (sofern es sich um Basisdatentypen handelt) zunächst mit 0 initialisiert.

```
pair<typ1, typ2> bezeichner;
```

pair-Basisdatentypen

Ein Standardkonstruktor bewirkt, dass die Elemente des jeweiligen Typs initialisiert werden. Hier einige Beispiele, wie Sie `pair`-Objekte mit einfachen Basisdatentypen erstellen können:

```
00 // kapitel12/utility/pair01.cpp
01 #include <utility>
...
02 std::pair<double, std::string> input_vals() {
03     double val;
04     std::string str;
05     std::cout << "Temperatur: "; std::cin >> val;
06     std::cout << "Monat      : "; std::cin >> str;
07     return std::pair<double, std::string>(val, str);
08 }
...
09 std::pair<std::string, long> number("Wolf", 554594);
10 std::pair<double, std::string> temperatur;
11 temperatur = input_vals();
...
```

In Zeile 09 werden dem Wertpaar `std::string/long` die Werte direkt übergeben. In Zeile 10 legen wir nur ein neues Wertpaar an, welches in Zeile 11 die Werte von der Funktion `input_vals()` (Zeilen 02 bis 08) erhält. Hier können Sie auch gleich sehr schön erkennen, dass Sie mit `pair` zwei Werte aus einer Funktion zurückgeben können, ohne extra Strukturen bzw. Klassen dafür zu verwenden.

std::make_pair

Die Standardbibliothek bietet mit `std::make_pair(val1, val2)` auch eine Komfortfunktion an, um aus `val1` und `val2` ein Paar zu konstruieren. Das erspart bspw. die

umständlichere Verwendung mit den spitzen Klammern. Bezogen auf die Funktion `input_vals()` aus Listing *pair01.cpp* könnten Sie Zeile 07 mit `std::make_pair()` wie folgt ändern:

```
00 // kapitel12/utility/pair01.cpp
...
07     return std::make_pair(val, str);
...
```

Zugriff auf die einzelnen Elemente von pair

Das Template `pair` ist als Struktur (`struct`) in der Headerdatei `<utility>` definiert, wo alle Mitglieder öffentlich sind. Hier ein Codeausschnitt zu dieser Struktur:

```
template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    // Konstruktor
    pair(){};
    // initialisiere first mit x und second mit y:
    pair(const T1& x, const T2& y);
    // copy constructor:
    template<class U, class V> pair(const pair<U, V> &p);
};
```

Auf die einzelnen Werte dieser Struktur wird über den entsprechenden Bezeichner zugegriffen, gefolgt vom gewünschten Element. Das erste `pair`-Element ist hierbei über den Bezeichner `first` und das zweite Element über `second` erreichbar. Bezogen auf unser Listing *pair01.cpp* lässt sich die Ausgabe der beiden `pair`-Objekte wie folgt realisieren:

```
00 // kapitel12/utility/pair01.cpp
...
01     std::cout << number.first << " : "
                << number.second << std::endl;
02     std::cout << temperatur.first << " / "
                << temperatur.second << std::endl;
```

Neu in C++11

Mit C++11 können Sie auf das erste Element alternativ auch mit `std::get<0>(number)` statt `number.first` und auf das zweite Element mit `std::get<1>(number)` statt `number.second` zugreifen.

Eigene Klassen mit pair verwenden

Natürlich können Sie in einem `pair`-Objekt auch Objekte verschiedener Klassen abspeichern. Allerdings müssen hier folgende Voraussetzungen erfüllt sein:

- Die Klasse muss einen Kopierkonstruktor besitzen.
- Die Klasse muss *zuweisbar* sein – somit muss der Zuweisungsoperator entsprechend überladen sein.
- Die Klassen müssen vergleichbar sein. Dazu müssen mindestens die Operatoren `<` und `==` durch `friend`-Funktionen überladen werden. Die restlichen Vergleichsoperatoren werden automatisch aus diesen beiden überladenen Operatoren gebildet. Das ist so, weil die Standardbibliothek im Namensraum `std::rel_ops` Vergleichsoperatoren zur Verfügung stellt, die es ermöglichen, dass in einer Klasse nur die Operatoren `==` und `<` definiert sein müssen und dennoch der komplette Satz an Vergleichsoperatoren verwendet werden kann.

Hier ein einfachstes Beispiel dazu, wo unsere Klasse `DataClass` alle diese Voraussetzungen erfüllt, um Teil eines `pair`-Templates sein zu dürfen:

```
00 // kapitel12/utility/pair01.cpp
...
01 class DataClass {
02     int data;
03 public:
04     DataClass( int d=0 ) : data(d) {}
05     DataClass( const DataClass& d) {data=d.data;}
06     DataClass& operator=(const DataClass& d) {
07         data=d.data;
08         return *this;
09     }
10     friend bool operator<( const DataClass& d1,
                           const DataClass& d2) {
11         return (d1.data < d2.data);
12     }
13     friend bool operator==(const DataClass& d1,
                           const DataClass& d2) {
14         return (d1.data == d2.data);
15     }
16     int getData() const { return data; }
17 };
...
18 std::pair<std::string, DataClass> mydata01(
    "Datensatz 01", 1234);
19 std::pair<std::string, DataClass> mydata02(
    "Datensatz 02", 1234);
```

```

20  std::cout << mydata01.first << ":"
      << mydata01.second.getData() << std::endl;
21  if( mydata01 != mydata02 ) {
22      std::cout << "Das Paar ist unterschiedlich\n";
23  }

```

Die Klasse `DataClass` hat jetzt mit dem Kopierkonstruktor (Zeile 05), dem Zuweisungsoperator (Zeilen 06 bis 08) und den Vergleichsoperatoren `<` und `==` alle erforderlichen Komponenten, um auch als Element von `pair` verwendet werden zu können, wie dies im Beispiel mit den Zeilen 18 bis 23 demonstriert wird.

pair und (unordered-)map- bzw. multimap-Container

Auch wenn sich das `pair`-Template recht vielseitig verwenden lässt, so ist das Hauptanwendungsgebiet vorwiegend bei den Containern `(unordered) map` und `multimap`, wo `pair` ideal dazu geeignet ist, die Schlüssel-/Wert-Paare zu verwalten.

Zum Abschluss noch eine Übersicht zu den Befehlen für das Template `pair` in Tabelle 12.30.

Befehl	Bedeutung
<code>pair<T1,T2> paar;</code>	Der Standardkonstruktor erzeugt ein Paar mit den Typen <code>T1</code> und <code>T2</code> .
<code>pair<T1,T2> paar(v1,v2)</code>	Erzeugt ein Paar mit den Typen <code>T1</code> und <code>T2</code> und initialisiert diese gleich mit den Werten <code>v1</code> und <code>v2</code> .
<code>pair<T1,T2> paar(paar2);</code>	Kopierkonstruktor erzeugt eine Kopie von <code>paar2</code> .
<code>paar1 = paar2;</code>	Zuweisungsoperator weist <code>paar1</code> den Inhalt von <code>paar2</code> zu.
<code>paar.first get<0>(first) (C++11)</code>	direkter Zugriff auf das erste Element
<code>paar.second get<1>(paar) (C++11)</code>	direkter Zugriff auf das zweite Element
<code>paar1==paar2; paar1!=paar2; paar<paar2; paar1<paar2; paar>paar2; paar1>=paar2;</code>	Vergleiche zwischen zwei <code>std::pair</code> -Objekten

Tabelle 12.30 Übersicht zu den wichtigsten Befehlen für das Template »`std::pair`«

Befehl	Bedeutung
<code>paar1.swap(paar2);</code> <code>swap(paar1, paar2);</code>	Die Daten von paar1 und paar2 austauschen. Beide sind erst seit C++11 dabei.
<code>make_pair(v1, v2);</code>	Erzeugt und gibt ein neues Paar aus den Typen v1 und v2 zurück.

Tabelle 12.30 Übersicht zu den wichtigsten Befehlen für das Template »std::pair« (Forts.)

12.6.2 Tupel, das bessere pair (C++11)

Das Template `std::tuple` wurde neu mit C++11 eingeführt und kann ähnlich wie das Template `std::pair` mehrere unterschiedliche Typen als Elemente speichern. Im Gegensatz zu `pair` sind Sie nicht mehr nur auf ein Paar beschränkt, sondern in einem Tupel lassen sich beliebig viele Elemente sichern. Für `std::tuple` benötigen Sie die Headerdatei `<tuple>`, und es findet sich alles wieder im Namenraum `std`.

Nutzen eines Tupels

Ein Tupel kann sehr nützlich sein, wenn Sie eine heterogene Liste mit verschiedenen Elementen benötigen, ohne extra dafür eine Struktur oder Klasse schreiben zu müssen, um dort die Daten zu speichern.

Ein `std::tuple` lässt sich fast genauso wie ein `std::pair` erzeugen. Hierzu einige Beispiele:

```
std::tuple<int, double, int> tripleVal;
std::tuple<double, std::string> twoVal(3.14, "Pi");
std::tuple<int, int, int, double, std::string> fiveVal;
std::tuple<int> singleVal;
```

Das folgende Beispiel demonstriert Listing *pair01.cpp* aus Abschnitt 12.6.1, wo einfach `std::pair` gegen `std::tuple` ersetzt wurde:

```
00 // kapitel12/utility/tuple01.cpp
01 #include <tuple>
...
02 std::tuple<double, std::string> input_vals() {
03     double val;
04     std::string str;
05     std::cout << "Temperatur: "; std::cin >> val;
06     std::cout << "Monat      : "; std::cin >> str;
07     return std::tuple<double, std::string>(val, str);
08 }
```

```
...
09  std::tuple<std::string, long> number("Wolf", 554594);
10  std::tuple<double, std::string> temperatur;
11  temperatur = input_vals();
...
```

std::make_tuple, tuple und pair konvertieren

Auch hier bietet die Standardbibliothek mit `std::make_tuple(val1, val2, ..., val_N)` eine Komfortfunktion an, um aus `val1`, `val2` bis `val_N` ein Tupel zu konstruieren. Das erspart bspw. die umständlichere Verwendung mit den spitzen Klammern. Bezogen auf die Funktion `input_vals()` aus Listing *tuple01.cpp* könnten Sie auch hier Zeile 07 mit `std::make_tuple()` wie folgt ändern:

```
00  // kapitel12/utility/tuple01.cpp
...
07  return std::make_tuple(val, str);
...
```

Noch C++11-mäßiger geht es auch hier mit dem Schlüsselwort `auto`:

```
auto tvals = std::make_tuple("Data", 123.12, 100);
```

Das Tupel ist hier sogar kompatibel zum Template `std::pair` (ist ja auch wieder ein Template), wenn zwei (passende) Typen verwendet werden. Im Listing *tuple01.cpp* könnten Sie quasi auch alles bei `std::pair` wie folgt belassen:

```
...
02  std::pair<double, std::string> input_vals() {
03      double val;
04      std::string str;
05      std::cout << "Temperatur: "; std::cin >> val;
06      std::cout << "Monat      : "; std::cin >> str;
07      return std::make_pair(val, str);
08  }
...
09  std::tuple<double, std::string> temperatur;
10  temperatur = input_vals(); // auch erlaubt
```

Zugriff auf die Elemente von std::tuple

Der Zugriff auf die einzelnen Elemente eines Tupels kann bequem mit dem Funktions-Template `get<n>()` gemacht werden. `n` steht für die Position des Elements. Die Zählung beginnt hier bei 0. Das folgende Listing soll demonstrieren, wie Sie an die einzelnen Elemente in einem Tupel mit `get<>` herankommen:

```

00 // kapitel12/utility/tuple02.cpp
...
01 std::tuple<std::string, long> number("Wolf", 554594);
02 auto tvals = std::make_tuple("Data", 123.12, 100);

03 std::cout << std::get<0>(number) << ":"
    << std::get<1>(number) << std::endl;
04 std::cout << std::get<0>(tvals) << ":"
    << std::get<1>(tvals) << ":"
    << std::get<2>(tvals) << std::endl;
05 std::get<0>(number) = "Wolff";
06 std::cout << std::get<0>(number) << ":"
    << std::get<1>(number) << std::endl;
07 std::cout << "Bitte double-Wert eingeben: ";
08 std::cin >> std::get<1>(tvals);
09 std::cout << std::get<0>(tvals) << ":"
    << std::get<1>(tvals) << ":"
    << std::get<2>(tvals) << std::endl;

```

In den Zeilen **03** und **04** werden die einzelnen Elemente der Tupel `number` und `tvals` mit `std::get<n>` ausgegeben. Dass der Zugriff auch schreibend möglich ist, sollen die Zeilen **05** mit der Zuweisung und **08** über die Standardeingabe zeigen.

Der Zugriff eines Tupels muss bereits zur Übersetzungszeit feststehen. In der Praxis bedeutet dies daher auch, dass Sie hier nicht in einer `for`-Schleife durch die einzelnen Elemente iterieren können. Wie für ein Template-Argument üblich, benötigen Sie für `get<>` zwischen den spitzen Klammern einen konstanten Ausdruck. Die einzige Möglichkeit, hierfür die Elemente zu iterieren, führt über die *Variadic Templates* (siehe Abschnitt 11.3, »Templates mit variabler Argumentanzahl (C++11)«). Allerdings ist die Implementierung hier auch wieder alles andere als einfach und muss rekursiv erfolgen.

Hilfsfunktion `std::tie`

Um einzelne Argumente aus einem Tupel zu extrahieren, bietet sich die Komfortfunktion `std::tie()` an. Diese Funktion gibt die Argumente eines Tupels als Referenz zurück. Anstatt mit mehreren `get<>` die einzelnen Elemente aus einem Tupel zu holen, können Sie alles in einem Rutsch mit `std::tie()` machen. Sind Sie an bestimmte Elemente in dem Tupel nicht interessiert, schreiben Sie einfach `std::ignore` an diese Stelle.

Der folgende Codeausschnitt soll die Verwendung mit `std::tie()` relativ vereinfacht in der Praxis demonstrieren:

```

00 // kapitel12/utility/tuple03.cpp
...
01 std::tuple<std::string, long> number("Wolf", 554594);
02 auto tvals = std::make_tuple("Data", 123.12, 100);
...
03 std::string str;
04 double dval;
05 int ival;
06 std::tie(str, dval, ival) = tvals;
07 std::cout << str << ":" << dval << ":" << ival << std::endl;

08 std::tie(str, std::ignore) = number;
09 std::cout << str << std::endl;

```

In Zeile **06** extrahieren wir alle Elemente aus dem Tupel `tvals` und übergeben die einzelnen Werte mittels `std::tie` an `str`, `dval` und `ival`. Die Ausgabe in Zeile **07** bestätigt dann auch, dass die Zuweisung funktioniert hat. In Zeile **08** verwenden wir erneut `std::tie()`. Nur sind wir diesmal nur am ersten Element in dem Tupel `number` interessiert, also an `std::string`. Der `long`-Wert von `number` interessiert uns hier nicht, weshalb wir dafür innerhalb von `std::tie()` einfach `std::ignore` schreiben.

Eigene Klassen mit `std::tuple` verwenden

Für Tupel mit eingebauten Typen und (vielen) Standardklassen (wie bspw. `std::string`) als Elementen können Sie bspw. die Vergleichsoperatoren (`==`, `!=`, `<=`, `<`, `>` und `>=`) wie gehabt ohne Einschränkungen verwenden.

```

00 // kapitel12/utility/tuple04.cpp
...
01 std::tuple<double, int, std::string> mixed01, mixed02;
02 mixed01 = std::make_tuple(3.1415, 101, "text01");
03 mixed02 = std::make_tuple(3.1415, 101, "text02");

04 if( mixed01 == mixed02 ) { // ohne Einschränkung möglich
05     std::cout << "Beide Tupel haben denselben Inhalt\n";
06 }
07 else {
08     std::cout << "Die Tupel sind unterschiedlich\n";
09 }

```

Bei eigenen Klassen hingegen müssen Sie dieselben Vorkehrungen treffen wie schon bei `std::pair`:

- Die Klasse benötigt einen Kopierkonstruktor, und der Zuweisungsoperator muss überladen sein.
- Die Klassen müssen vergleichbar sein. Dazu müssen mindestens die Operatoren `<` und `==` durch `friend`-Funktionen überladen werden. Die restlichen Vergleichsoperatoren werden automatisch aus diesen beiden überladenen Operatoren gebildet.

Diese Klasse haben Sie ja bereits in Listing *pair01.cpp* im Abschnitt »Eigene Klassen mit `pair` verwenden« in Abschnitt 12.6.1 gesehen, weshalb wir diese hier nicht nochmals extra abzudrucken brauchen (den kompletten Quellcode finden Sie auf der Buch-CD). Ist die Klasse `DataClass` entsprechend implementiert, können Sie auch Tupel damit wie folgt verwenden:

```
00 // kapitel12/utility/tuple05.cpp
...
01 std::tuple<double, int, DataClass> mixed01, mixed02;
02 mixed01 = std::make_tuple(3.1415, 101, 404);
03 mixed02 = std::make_tuple(3.1415, 101, 405);

04 if( mixed01 == mixed02 ) {
05     std::cout << "Beide Tupel haben denselben Inhalt\n";
06 }
07 else {
08     std::cout << "Die Tupel sind unterschiedlich\n";
09 }
```

Weitere nützliche Funktionen für `std::tuple`

Es gibt noch einige spezielle Hilfsfunktionen, die im Zusammenhang mit Tupeln recht nützlich sind. Die Anzahl der Elemente in einem Tupel erhalten Sie mit `std::tuple_size<typ>::value` und mit `std::tuple_element<n,typ>::type` können Sie einen Alias für den Typ `in get<n>(tup)` erzeugen.

Und dann wäre da noch `std::tuple_cat()`, womit Sie aus mehreren Daten wie anderen Tupeln, Basisdatentypen und Klassen oder `std::pair` ein Tupel erzeugen können.

Das folgende Listing soll diese Hilfsfunktionen im Einsatz zeigen:

```
00 // kapitel12/utility/tuple06.cpp
...
01 std::tuple<std::string, long> number("Wolf", 554594);
02 double dval = 3.334;
03 std::pair<long, std::string> apair(123456, "Text");
04 auto bigtup = std::tuple_cat(number, std::tie(dval), apair);
...
```

```

05  std::cout << "Elemente in bigtup: " <<
      std::tuple_size<decltype(bigtup)>::value <<std::endl;
06  std::tuple_element<0,decltype(bigtup)>::type first =
      std::get<0>(bigtup);
07  std::tuple_element<2,decltype(bigtup)>::type third =
      std::get<2>(bigtup);
08  std::cout << "Wert im Element 1: " << first << std::endl;
09  std::cout << "Wert im Element 3: " << third << std::endl;

```

In Tabelle 12.31 finden Sie nochmals eine Übersicht der wichtigsten Befehle für Tupel.

Befehl	Beschreibung
<code>tuple<T1,T2,...,Tn> tup;</code>	Erzeugt ein Tupel mit n Elementen mit den Typen, die im formalen Parameter angegeben werden. Klassen werden dabei mit dem Standardkonstruktor und fundamentalen Typen mit 0 initialisiert.
<code>tuple<T1,T2,...,Tn> tup(v1,v2,...,vn);</code>	Erzeugt ein Tupel mit n Elementen und initialisiert die einzelnen Elemente mit den übergebenen Werten v_1, v_2 bis v_n .
<code>tup1=tup2;</code>	Weist den Tupeln <code>tup1</code> ein Tupel <code>tup2</code> zu.
<code>tup=paar;</code>	Weist einem Tupel mit zwei Elementen ein <code>std::pair</code> zu. Die Typen von <code>std::pair</code> müssen übereinstimmen.
<code>tup1==tup2; tup1!=tup2; tup1<tup2; tup1<=tup2; tup1>tup2; tup1>=tup2;</code>	Vergleichsoperatoren für Vergleiche mit Tupeln
<code>tup1.swap(tup2); swap(tup1, tup2);</code>	Die Daten von <code>tup1</code> werden mit <code>tup2</code> getauscht.
<code>make_tuple(v1,v2,...,vn);</code>	Erzeugt ein Tupel aus und mit den Typen der übergebenen Werte.
<code>tie(ref1, ref2,...,refn);</code>	Erzeugt ein Tupel von Referenzen, was gewöhnlich dafür verwendet wird, um die einzelnen Werte aus einem Tupel zu extrahieren.

Tabelle 12.31 Übersicht der gängigen Befehle für »std::tuple«

Tupel in einen Container packen

Natürlich können Sie Tupel auch in Container packen. Folgendes Beispiel demonstriert diesen Vorgang:

```

00 // kapitel12/utility/tuple07.cpp
...
01 using Telefon = std::tuple<std::string, std::string, long>;
02 Telefon num01("Mustermann", "Max", 555870);
03 Telefon num02("Musterfrau", "Maxi", 555643);

04 std::deque<Telefon> numbers;
05 numbers.push_back(num01);
06 numbers.push_back(num02);
07 numbers.push_back(
    std::make_tuple("Mustersohn", "Max", 555132));
...
08 std::string name, forname;
09 long mobile = 0;
10 for( const auto& elem : numbers) {
11     std::tie(name, forname, mobile) = elem;
12     std::cout << "Name      : " << name << std::endl;
13     std::cout << "Vorname  : " << forname << std::endl;
14     std::cout << "Nummer   : " << mobile << std::endl;
15 }

```

Damit Sie nicht so viel tippen müssen, verwenden wir gleich in Zeile **01** einen `using`-Alias für die Tupel. In den Zeilen **02** und **03** erzeugen wir zwei Tupel mit diesem Alias. Jetzt wollen wir unsere Tupel in einen Container verpacken und verwenden `std::deque` in Zeile **04** dafür. Entsprechend sieht auch die Deklaration aus. Dank unseres `using` in Zeile **01** ersparen wir uns eine grausame Zeile wie:

```
std::deque<std::tuple<std::string, std::string, long>> numbers;
```

In den Zeilen **05** bis **07** fügen wir neue Tupel-Daten am Ende der `deque` hinzu. Natürlich können Sie dies auch wieder an Ort und Stelle mit `std::make_tuple()` machen, wie es in Zeile **07** verwendet wurde.

Die Ausgabe aller Daten machen wir in den Zeilen **08** bis **15**. Dank der Hilfsfunktion `std::tie()` ist es zum Glück kein Problem, an die einzelnen Daten der Tupel innerhalb von `std::deque` zu kommen.

12.6.3 Vergleichsoperatoren für eigene Typen

Sie haben es bei `std::pair` und `std::tuple` bereits gelesen und verwendet und sich sicherlich gewundert, warum es bei Objekten von selbst definierten Typen ausreicht, nur die Operatoren `==` und `<` zu implementieren. Dies verdanken Sie dem Namensraum `std::rel_ops` aus der Headerdatei `<utility>`. In diesem Namenraum sind

Funktions-Templates enthalten, welche aus dem `operator==` das Verhalten von `!=` und aus dem `operator<` das Verhalten von `>`, `<=` und `>=` ableiten.

Folgendes Minimalbeispiel soll dies demonstrieren:

```
00 // kapitel12/utility/rel_ops.cpp
...
01 #include <utility>
02 using namespace std::rel_ops;

03 class DataClass {
04     int data;
05 public:
06     DataClass( int d=0 ) : data(d) {}
07     friend bool operator<( const DataClass& d1,
                           const DataClass& d2) {
08         return (d1.data < d2.data);
09     }
10     friend bool operator==( const DataClass& d1,
                           const DataClass& d2) {
11         return (d1.data == d2.data);
12     }
13 };
...
13 DataClass d1(100);
14 DataClass d2(200);
15 if( d1 != d2 ) {
16     std::cout << "Die Elemente sind nicht gleich\n";
17 }
```

Dass bei diesem Beispiel der Vergleich der beiden `DataClass`-Objekte in Zeile 15 mit dem `!=`-Operator funktioniert, verdanken Sie dem Funktions-Template im Namensraum `std::rel_ops` in der Headerdatei `<utility>`, weil damit aus den implementierten `operator==` und `operator<` die restlichen Operatoren abgeleitet werden.

12.6.4 `std::bind` (C++11)

`std::bind` wurde mit C++11 als eine allgemeine Version für die veralteten `std::bind1st()` und `std::bind2nd()` eingeführt. Das Template `std::bind()` bindet Parameter oder Werte an eine Funktion und gibt ein Funktionsobjekt zurück, welches Sie dann entweder wie eine normale Funktion aufrufen oder eben auch sehr komfortabel mit den Algorithmen verwenden können, um bspw. aus einem binären Funktionsobjekt ein unäres Funktionsobjekt zu machen. Ein einfaches Beispiel hierzu:

```

00 // kapitel12/utility/bind01.cpp
...
01 #include <functional>
02 using namespace std::placeholders;
...
03 std::vector<int> ivec{12, 32, 11, 33, 5, 29};
04 int cnt = std::count_if(ivec.begin(), ivec.end(),
    std::bind(std::greater<int>(),_1, 20));
05 std::cout << cnt << std::endl; // = 3

```

In diesem Beispiel werden in Zeile **04** mit `std::count_if` alle Elemente gezählt, die größer als (`greater<int>()`) 20 sind. Das Bemerkenswerte hierbei ist, dass `std::greater<int>()` ein binäres Funktionsobjekt ist und unser Algorithmus eigentlich ein unäres Funktionsobjekt erwartet. Durch `std::bind()` wird das binäre Funktionsobjekt mitsamt dem Argument zu einem unären Funktionsobjekt gebunden.

Das jeweils aktuelle Argument, mit dem im Augenblick verglichen wird, wird mit Platzhalter `_1` (genauer `std::placeholders::_1`) verwendet. Der Wert im Platzhalter wird dann mit dem binären Objekt `std::greater<int>()` mit dem Wert 20 verglichen, um zu sehen, ob dieser größer ist. Sie können auch ein solches Funktionsobjekt binden und weiterverwenden, um zu überprüfen, ob Werte größer als 20 sind. Folgendes Beispiel demonstriert dies und macht die Hilfsfunktion `std::bind()` auch etwas einfacher:

```

00 // kapitel12/algorithm/bind02.cpp
...
01 #include <functional>
02 using namespace std::placeholders;
03 std::vector<int> ivec{12, 32, 11, 33, 5, 29};

04 auto f = std::bind(std::greater<int>(),_1, 20);

05 int cnt = std::count_if(ivec.begin(), ivec.end(), f);
06 std::cout << cnt << std::endl; // = 3

07 if( f(cnt) ) {
08     std::cout << "Mehr als 20 Elemente sind >20\n";
09 }
10 else {
11     std::cout << "Weniger als 20 Elemente sind >20\n";
12 }

```

In Zeile **04** erstellen Sie ein unäres Funktionsobjekt, mit dem Sie `int`-Werte für `_1` als Argument übergeben können, um zu prüfen, ob diese größer als 20 sind. Hierbei erweist sich `auto` wieder mal als besonders nützlich. Einmal so definiert, können Sie

es nach wie vor in einem Algorithmus wie in Zeile 05 oder komplett alleinstehend wie in Zeile 07 verwenden.

Natürlich können Sie noch weitere Platzhalter mit `_1`, `_2`, `_3` usw. verwenden. Diese Technik wird als *Currying* bezeichnet. Ein einfaches Beispiel auch hierzu:

```
00 // kapitel12/utility/bind03.cpp
...
01 #include <functional>
02 using namespace std::placeholders; // _1, _2, _3, ...

03 class DivZero {}; // Leere Fehlerklasse

04 double safe_division(double x, double y) {
05     if(y==0) {
06         throw DivZero();
07     }
08     return(x/y);
09 }
...
10 auto f = std::bind(safe_division, _1, _2);
11 try {
12     auto dval1 = f(100,20);
13     auto dval2 = f(100, 0); // Fehler!!!
14 }
15 catch(DivZero& e) {
16     std::cout << "Division durch 0 abgefangen\n";
17 }
```

Hier haben wir in Zeile 10 ein Funktionsobjekt für eine sichere Division erstellt, wo Sie zwei Argumente für die Platzhalter `_1` (= Divisor) und `_2` (= Dividend) verwenden können, wie es in Zeile 12 und 13 dann auch hingeschrieben und verwendet wurde.

`bind1st()`, `bind2nd`, `ptr_fun()`, `mem_fun()` und `mem_fun_ref()`

Mit Einführung von `std::bind()` sind die oben genannten Binder jetzt veraltet und werden mit C++11 zu »deprecated« erklärt.

`std::bind()` vs. Lambdas

`std::bind()` ist wirklich ein mächtiges Feature und seit der Standarderweiterung mit TR1 im Einsatz. Da allerdings mit Lambda-Funktionen ab C++11 ein ähnliches und gleichwertiges Feature vorhanden ist, dürfte diese Hilfsfunktion etwas an Bedeutung verloren haben.

Nützlich dürfte `std::bind()` aber trotzdem sein, wenn der Compiler neuere C++11-Features wie Lambdas nicht unterstützt. Dann kann der Programmierer immer noch auf `std::bind()` zurückgreifen.

12.6.5 Verschieben mit `move()` (C++11)

Eines der Top-Features im neuen C++11-Standard ist die Möglichkeit, die Daten mit der neuen Move-Semantik zu verschieben, anstatt sie zu kopieren. Dass dies nun möglich ist, verdanken wir den *RValue*-Referenzen, welche ebenfalls neu mit dem C++11-Standard eingeführt wurden. Gerade bei einer sehr umfangreichen Datenmenge sorgt ein Verschieben mit dem neuen Funktions-Template `std::move()` für einen gewaltigen Performance-Schub gegenüber dem Kopieren. Das Tolle daran ist auch, dass alle Standardcontainer diese Funktionalität bereits von Haus aus unterstützen:

```
01 std::vector<int> ivec01{23, 32, 43, 12, 34, 43};
02 std::vector<int> ivec02 = ivec01; // Kopie
03 std::vector<int> ivec03 = std::move(ivec02); // Verschieben
```

Durch das Kopieren von `ivec01` nach `ivec02` in Zeile **02** haben beide Vektoren denselben Inhalt. Nach dem Verschieben von `ivec02` nach `ivec03` in Zeile **03** ist der Inhalt in `ivec03`, und `ivec02` ist somit leer.

Bei eigenen Typen (wie Klassen) müssen Sie die Funktionalität hingegen noch implementieren. Kurz wurde darauf ja bereits im Abschnitt »Verschiebekonstruktor (C++11)« in Abschnitt 7.4.2 und in Abschnitt 7.10, »Klassen für das Verschieben schreiben (C++11)«, eingegangen. Aber auch hier wurde das Thema nur angerissen, und die *RValue*-Referenzen wurden noch nicht besprochen. Das Thema lässt sich hier auch nicht mal eben schnell abhandeln, weshalb Sie eine umfassendere Einführung zu *RValue*-Referenzen, Move-Semantik und dem Perfect Forwarding in Abschnitt 16.1, »Move-Semantik und Perfect Forwarding«, finden.

Weitere Vorteile von `std::move()`

Neben dem Vorteil der besseren Performance hat das neue Funktions-Template `std::move()` auch den Vorteil, dass Sie Daten, die nicht kopiert werden können, zumindest verschieben können. Hierzu gehören bspw. Threads, Mutexe oder Locks (siehe Kapitel 15, »Multithreading (C++11)«).

12.6.6 Smart Pointer (C++11)

Zwar gab es mit dem `auto_ptr` bereits einen etwas intelligenteren Zeiger in C++, aber dieser hatte so seine Schwächen, und seine Verwendung wird jetzt mit dem C++11-

Standard nicht mehr empfohlen (er ist als »deprecated« markiert). Als bessere Alternative wurden in C++11 mit `shared_ptr`, `weak_ptr` und `unique_ptr` gleich drei neue kluge Zeiger eingeführt, die alle in der Headerdatei `<memory>` und im Namensraum `std` enthalten sind. Diese Zeiger bieten eine automatische Speicherverwaltung, bei der die Ressourcen automatisch freigegeben werden, wenn sie den Gültigkeitsbereich verlassen. Speicherlecks gehören hiermit der Vergangenheit an.

Zum Weiterlesen

Eine umfassende Einführung in die Smart Pointer finden Sie in Abschnitt 16.3, »Smart Pointer (C++11)«.

12.6.7 Zeitbibliothek – `<chrono>` (C++11)

Endlich liefert C++11 auch eine eigene Zeitbibliothek mit der Headerdatei `<chrono>` mit, und man ist nicht mehr so von den Zeitfunktionen der alten C-API oder Drittanbieter-Bibliotheken abhängig.

Zeitdauer (`duration`)

Hierbei bietet die Bibliothek die wichtigsten Zeitdauern bereits vordefiniert als Synonyme an, Sie finden sie in Tabelle 12.32.

Zeiteinheit	Synonym für
<code>nanoseconds</code>	Nanosekunden
<code>microseconds</code>	Mikrosekunden
<code>milliseconds</code>	Millisekunden
<code>seconds</code>	Sekunden
<code>minutes</code>	Minuten
<code>hours</code>	Stunden

Tabelle 12.32 Vordefinierte Zeiteinheiten

Die Anwendung dieser vordefinierten Zeiteinheiten wird zum Kinderspiel:

```
00 // kapitel12/utility/chrono01.cpp
...
01 #include <chrono>
02 using namespace std::chrono; // minutes, seconds, ...
...
```



```

03 minutes min01 = hours(2) + minutes(11);
04 seconds sec01 = minutes(min01);
05 nanoseconds nano01 = minutes(min01) - hours(1);
06 std::cout << min01.count() << " min" << std::endl;
07 std::cout << sec01.count() << " sec" << std::endl;
08 std::cout << nano01.count() << " nansosec" << std::endl;

```

Um die Zeiten verwenden zu können, benötigen Sie den Header `<chrono>` (Zeile 01) und den Namensbereich `std::chrono` (Zeile 02). Jetzt können Sie ganz komfortabel mit den Zeiteinheiten rechnen, wie Sie in den Zeilen 03 bis 05 sehen können. Die Ausgabe der entsprechenden Zeiteinheit können Sie dann ebenfalls ganz bequem mit dem Objekt, gefolgt von der Elementfunktion `count()`, durchführen, wie in den Zeilen 06 bis 08 demonstriert wird.

std::ratio

In der neuen Bibliothek `std::ratio` sind eine Menge vordefinierter international standardisierter Namen, welche Ihnen bei der Umrechnung von Einheiten helfen können. Mit einem Blick in die Headerdatei `<ratio>` können Sie sich hierbei selbst einen Überblick verschaffen. Ein Ausschnitt dazu sieht wie folgt aus:

```

...
typedef ratio<1, 1000000> micro
typedef ratio<1, 1000> milli
typedef ratio<1, 100> centi
typedef ratio<1, 10> deci
typedef ratio<10, 1> deca
typedef ratio<100, 1> hecto
typedef ratio<1000, 1> kilo
typedef ratio<1000000, 1> mega
...

```

Viel interessanter ist allerdings die Möglichkeit, sich auf diese Weise seine eigenen Einheiten bauen zu können. Aufbauend auf `ratio<Zähler,Nenner>` und das Klassen-Template `duration` ist dies einfacher, als man annehmen würde:

```

00 // kapitel12/utility/chrono02.cpp
...
01 #include <chrono>
02 using namespace std::chrono;
...
03 using Meilen = duration <double, std::ratio<1609, 1000>>;
04 using Yard = duration <double, std::ratio<10, 10936>>;
05 using Kilometer = duration <double, std::ratio<1>>;

```

```

06 Kilometer km(22.7);
07 Meilen mile01 = km;
08 Yard yard01 = km;
09 std::cout << mile01.count() << std::endl;
10 std::cout << yard01.count() << std::endl;

```

In Zeile 03 erstellen wir die Einheiten Meilen vom Typ `double` mit dem Faktor 1609 zu 1000. Eine Zeile später erstellen wir noch eine Einheit `Yard` mit dem Faktor 10 zu 10936, und in Zeile 05 erstellen wir `Kilometer` mit dem Faktor 1 zu 1, was somit in diesem Fall unsere Basiseinheit darstellen soll. Anschließend in Zeile 06 definieren wir einen Typ `Kilometer` mit 22.7 km. Diesen Wert weisen wir in den Zeilen 07 und 08 jeweils einem Typ `Meilen` und `Yard` zu. In den Zeilen 09 und 10 lassen wir nun mithilfe von `count()` und den von uns definierten Faktoren die entsprechenden `Kilometer` in `Meilen` und in `Yard` umrechnen und ausgeben.

Vorhandene Zeitgeber

Endlich können Sie mit der neuen C++11-Zeitbibliothek auch Zeitmessungen durchführen. Hierfür stehen Ihnen drei verschiedene Zeitgeber zur Verfügung:

- ▶ `system_clock`
Die Systemzeit, welche auch mit der externen Uhr synchronisiert werden kann. Diese Zeit kann durch eine Zeitumstellung oder andere Systemeinstellungen verändert werden. Hierbei handelt es sich gewöhnlich um den Beginn der Zeitrechnung ab dem 1.1.1970.
- ▶ `steady_clock`
Diese Zeit läuft mit einer immer konstanten Geschwindigkeit und kann nicht explizit verändert werden. Diese Zeitrechnung beginnt in der Regel ab dem Bootzeitpunkt des Rechners.
- ▶ `high_resolution_clock`
Die Uhrzeit mit der höchsten auf dem System verfügbaren Genauigkeit. Auf manchen Systemen ist diese Zeit lediglich ein Synonym für `system_clock` bzw. `steady_clock`.

Den aktuellen Zeitpunkt dieser drei Uhren können Sie jederzeit mit `now()` zurückgeben. Ein einfaches Beispiel soll dies demonstrieren:

```

00 // kapitel12/algorithm/chrono03.cpp
01 #include <chrono>
02 using namespace std::chrono;
03 ...
04 void funktion() {
05     for(int i=0; i<123456789; ++i);
06 }

```

```

...
06 const auto start = steady_clock::now();
07 funktion();
08 const auto stop = steady_clock::now();

09 nanoseconds dauer = stop - start;
10 std::cout << "Dauer: " << dauer.count() << "ns = ";
11 milliseconds millisec = duration_cast<milliseconds>(dauer);
12 std::cout << "Dauer: " << millisec.count() << "ms\n";

```

In Zeile **06** holen wir uns den aktuellen Zeitpunkt und speichern diesen in `start`. Der Rückgabewert von `steady_clock::now()` ist `steady_clock::time_point`, aber wir machen uns das Leben mit `auto` einfacher. In Zeile **07** verschwenden wir ein wenig Rechenzeit, indem wir eine nutzlose Funktion aufrufen. In Zeile **08** holen wir dann erneut den aktuellen Zeitpunkt und speichern diesen in `stop`. In Zeile **09** berechnen wir die Differenzen der beiden Zeiten und speichern diese in `dauer` mit Nanosekunden als Einheit und geben diese Zeit in Zeile **10** auf dem Bildschirm aus. Damit haben Sie quasi die Laufzeit der Funktion `funktion()` in Nanosekunden gemessen. Eine Zeile später in Zeile **11** wollen wir den Wert von Nanosekunden in Millisekunden »reduzieren«. Da bei dieser Umrechnung an Genauigkeit verloren geht, müssen Sie diesen Wert explizit mit `duration_cast<>` konvertieren.

Kapitel 13

Die Stream-Ein-/Ausgabeklassen von C++

Ohne eine Ein- und Ausgabe von Daten wäre eine Programmiersprache wohl sinnlos. Im Standard von C++ wurden die verschiedenen Ein-/Ausgabe-Streams in einer Hierarchie von Klassen-Templates gesteckt – kurz der I/O-Stream-Bibliothek.

Der Umgang mit Dateien und mit der Ein- und Ausgabe wird in C++ über ein spezielles und erweiterbares Streamkonzept realisiert. In diesem Kapitel wird dieses grundlegende Streamkonzept von C++ etwas umfassender erläutert.

13.1 Das Ein-/Ausgabe-Stream-Konzept von C++

Auf den folgenden Seiten werden Sie sehr häufig den Begriff *Stream* (engl. für Strom) oder Stream-Objekt lesen. `cout` oder `cin` sind bspw. solche (vordefinierten globalen) Stream-Objekte. Ein Stream ist ein Strom von Daten, die in eine bestimmte Richtung fließen, oder kurz der Datenfluss. Jedes Stream-Objekt hat dabei eigene Eigenschaften, welche von der entsprechenden Stream-Klasse festgelegt werden.

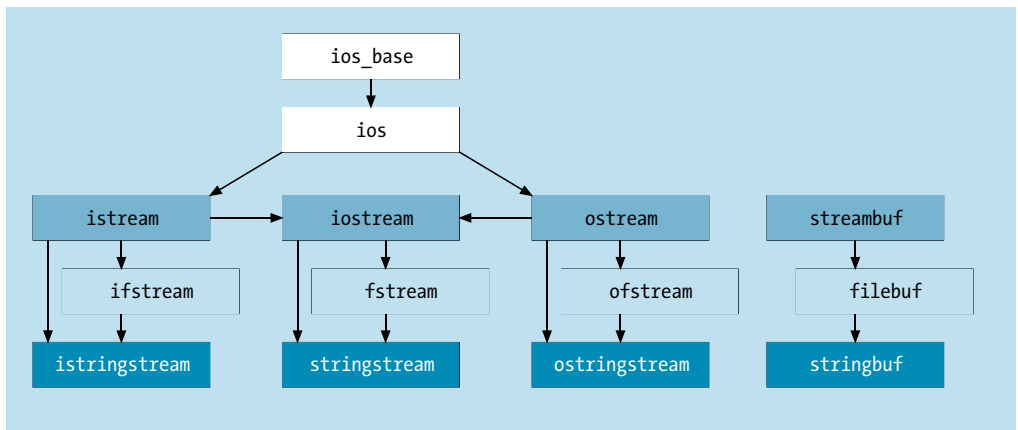


Abbildung 13.1 Vereinfachte Darstellung der Ein-/Ausgabe-Klassenhierarchie von C++

Das komplette Prinzip basiert wieder auf einer Hierarchie von Klassen-Templates, in der die Klasse `ios_base` an der Spitze der Klassenhierarchie steht und von der alle anderen Stream-Klassen abgeleitet werden. In Abbildung 13.1 können Sie eine vereinfachte Darstellung der Ein-/Ausgabe-Klassenhierarchie sehen.

Die I/O-Stream-Bibliothek mit allen Facetten ist extrem umfangreich, weshalb hier nicht auf alle Details der Bibliothek eingegangen werden kann.

Spezialisierungen

Bei der Darstellung in Abbildung 13.1 wurden bereits die Spezialisierungen für einen Basistyp dargestellt. Abgesehen von `ios_base` würde hier nämlich vor jedem Klassen-Template noch das Präfix `basic_` stehen. Eine Spezialisierung sieht demnach wie folgt aus:

```
...
typedef basic_ostream<char> ostream;
...
```

Hierbei handelt es sich um eine Spezialisierung für den Datentyp `char` aus dem Klassen-Template `basic_ostream`. Solche Spezialisierungen lassen sich natürlich auch für erweiterte Zeichensätze, wie bspw. `wchar_t` verwenden.

13.2 Globale, vordefinierte Standard-Streams

Die I/O-Stream-Bibliothek bietet von Haus aus vordefinierte und globale Stream-Objekte an, welche Sie praktisch schon ab dem ersten C++-Programm verwendet haben, ohne zu wissen, worum es sich hierbei handelt. Die Rede ist hier natürlich von den Standard-Stream-Objekten `cin`, `cout`, `cerr` und `clog` aus dem Namensbereich `std`.

Für die Ausgabe sind in der Headerdatei `<ostream>` die Stream-Objekte `cout` (Standardausgabe), `cerr` (Standardfehlerausgaben, ungepuffert) und `clog` (Standardfehlerausgabe, gepuffert) vorhanden. Für die Eingabe findet sich in der Headerdatei `<istream>` das Stream-Objekt `cin`, welches für die gewöhnlichen Anwendereingaben (bspw. von der Tastatur) verwendet werden kann.

Stream-Operatoren << und >>

Im Grunde ist ein Datenstrom zunächst nichts anderes als eine unleserliche Folge von Bytes, welche erst durch eine Spezialisierung von `operator<<()` für die Ausgabe und `operator>>()` für die Eingabe zu einem für uns sinnvollen Zusammenhang gefasst (genauer formatiert) wird.

In der Headerdatei `<ostream>` finden Sie daher den Operator `<<()` mehrfach überladen für alle in C++ eingebauten Typen definiert:

```
ostream& operator<< (bool val);
ostream& operator<< (short val);
ostream& operator<< (unsigned short val);
ostream& operator<< (int val);
...
```

Dasselbe gilt auch für den Operator `>>()` in der Headerdatei `<istream>`:

```
istream& operator>> (bool& val);
istream& operator>> (short& val);
istream& operator>> (unsigned short& val);
istream& operator>> (int& val);
...
```

Dadurch, dass bei den Operatoren `operator<<()` und `operator>>()` als Rückgabebetyp eine Referenz verwendet wird, wird die Verwendung der Operatoren mehrmals in der Reihe erst möglich. Folgendes ist damit gemeint:

```
// kapitel13/iostream/stream01.cpp
...
01 int val1, val2;
02 std::cout << "Bitte 2 int-Werte: ";
03 std::cin >> val1 >> val2;
04 std::cout << val1 << " : " << val2 << std::endl;
...
```

Eigene Typen verwenden

Viele Standardklassen wie `std::string` haben ebenfalls die Operatoren `operator<<()` bzw. `operator>>()` implementiert. Bei eigenen Klassen müssen Sie daher die beiden Operatoren selbst implementieren, wenn Sie `<<` bzw. `>>` auch damit verwenden wollen. In Abschnitt 8.3.14, »Ein- und Ausgabeoperatoren – `operator>>` und `operator<<`«, haben Sie bereits gesehen, wie dies in der Praxis realisiert wird.

Die Standard-Ein-/Ausgabe-Operatoren unterstützen fast alle fundamentalen Typen in C++. Ausgenommen davon sind `void` und `nullptr_t`. Natürlich erfolgt die Ein- bzw. Ausgabe nicht einfach willkürlich. Es gibt auch ein paar Dinge, die Sie hierbei beachten müssen:

► Numerische Typen

Beim Einlesen eines numerischen Wertes muss die Eingabe mit mindestens einer Ziffer beginnen. Geben Sie das Vorzeichen mit + oder - an, so darf dies nur vor der ersten Ziffer stehen. Zwischen dem Vorzeichen und der Ziffer darf allerdings kein Leerraum mehr folgen. Führende Leerraum-Zeichen werden übersprungen und ignoriert. Bei einer fehlerhaften Eingabe wird der numerische Wert auf 0 und `std::failbit` gesetzt.

► `char*`

Die Verwendung von `char*` bei der Eingabe war schon in C sehr kritisch. Eingelesen wird bis zum ersten Leerzeichen oder bis *End of File* (dazu später mehr). Führende Leerzeichen werden übersprungen und ignoriert, was allerdings mit dem Manipulator `skipws` geändert werden kann. Das Problem beim Einlesen von `char*` ist, dass hiermit über den Pufferbereich eingelesen werden kann (Pufferüberlauf, engl. *Buffer Overflow*), bspw.:

```
01 char cval[20];
02 std::cin >> cval; // !!! Gefährlich !!!
03 std::cout << cval << std::endl;
```

Wenn Sie hier in Zeile 02 mehr als 20 Zeichen einlesen, so macht dies `std::cin` brav (!) mit und schreibt munter in einen undefinierten Speicherbereich. Daher müssen Sie bei alten C-Strings (wenn Sie diese unbedingt verwenden müssen) immer darauf achten, dass die maximale Länge nicht überschritten wird. Hierfür bietet sich der Manipulator `std::setw()` aus der Headerdatei `<iomanip>` an:

```
std::cin >> std::setw(20) >> cval;
```

► `bool`

Die Ausgabe von `bool` wird bei `false` nach 0 und bei `true` nach 1 konvertiert. Bei der Eingabe hingegen gilt zunächst dasselbe wie bei numerischen Werten. Bei der Eingabe von gültigen numerischen Werten ungleich 0 wird nach 1 (= `true`) konvertiert. Bei der Eingabe von 0 wird auch 0 (für `false`) verwendet. Bei einer fehlerhaften Eingabe wird der Wert ebenfalls auf 0 und `std::failbit` gesetzt.

13.3 Methoden für die Aus- und Eingabe von Streams

Neben den üblichen Operatoren `operator<<()` und `operator>>()` für Streams können Sie auch Methoden verwenden, welche die I/O-Stream-Bibliothek anbietet. Allerdings arbeiten diese Methoden im Gegensatz zu den Operatoren `<<` oder `>>` mit rohen »Bytes« und somit unformatiert. Die Methoden für die Standard-Ein-/Ausgabe verwenden den Typ `streamsize` aus der Headerdatei `<ios>`, was gewöhnlich ein integraler signed-Typ (`size_t`) ist. Das bedeutet z.B. bei der Eingabe auch, dass führende Leerzeichen nicht überlesen werden.

13.3.1 Methoden für die unformatierte Ausgabe

Für die rohe und unformatierte Ausgabe stellt die I/O-Stream-Bibliothek die Methoden `put()` und `write()` aus der Headerdatei `<ostream>` zur Verfügung. Die Methode `put()` gibt einzelne Zeichen und `write()` eine bestimmte Anzahl von Zeichen aus. Folgender Codeausschnitt soll diese Methoden kurz demonstrieren:

```
00 // kapitel13/iostream/stream02.cpp
...
01 std::string sdata("rohe Daten");
02 for(int it=0; it<sdata.size(); ++it) {
03     std::cout.put(sdata.at(it));
04 }
05 // oder C++11-like:
06 for(auto c : sdata) std::cout.put(c);
...
07 std::cout << 10 ; // = 10;
08 std::cout.put(10); // = Newline '\n'
...
09 char cdata[] = {"rohe Daten"};
10 std::cout.write(cdata, 4); // = rohe
11 std::cout.put(10); // Newline
12 std::cout.flush();
```

In den Zeilen **02** bis **04** bzw. im C++11-Stil in Zeile **06** geben Sie die einzelnen Zeichen des Strings `sdata` unformatiert auf dem Bildschirm aus. In den Zeilen **07** und **08** können Sie sehr schön den Vergleich zwischen formatierter und unformatierter Ausgabe sehen. Während in Zeile **07** mit `operator<<()` der formatierte Wert 10 ausgegeben wird, gibt `put()` in Zeile **08** den unformatierten (meistens ASCII-)Wert auf dem Bildschirm aus. Mit einem Blick auf eine ASCII-Tabelle steht der Wert 10 dort für das Newline-Zeichen. In Zeile **10** wird außerdem noch die unformatierte Ausgabe mit `write()` demonstriert. Hier wird einfach eine bestimmte Anzahl von Zeichen (hier 4) von `cdata` unformatiert auf dem Bildschirm ausgegeben. Hier besteht natürlich wieder eine Gefahr, weil hier auch mehr Zeichen angegeben werden können, als überhaupt vorhanden sind. In Zeile **12** erzwingen wir außerdem noch mit `flush()`, dass alle Daten, die sich noch im Puffer des Streams befinden, sofort ausgegeben werden müssen.

Stream positionieren

Zur Positionierung von Streams können Sie die Methoden `tellp()` und `seekp()` verwenden, welche ebenfalls in `<ostream>` enthalten sind. Diese Methoden sind erst dann richtig interessant, wenn Sie mit Dateien arbeiten. Mehr dazu erfahren Sie in Abschnitt 13.6, »Streams für die Datei-Ein-/Ausgabe«.

Methode	Beschreibung
<code>ostream& put(char c);</code>	Schreibt das Argument <code>c</code> in den Stream.
<code>ostream& write(char* s, streamsize n);</code>	Schreibt <code>n</code> Zeichen von <code>s</code> in den Stream. Der Aufrufer muss sicherstellen, dass der Wert von <code>n</code> nicht größer ist, als Zeichen in <code>s</code> vorhanden sind. Auch das Terminierungszeichen beendet die Ausgabe von <code>write()</code> nicht und wird ebenfalls geschrieben.
<code>ostream& flush();</code>	Entleert den Stream-Puffer unverzüglich. Alle noch nicht geschriebenen Zeichen im Puffer müssen jetzt ausgegeben werden.

Tabelle 13.1 Methoden zur unformatierten Ausgabe in `<ostream>`

Alle Methoden geben den Stream `ostream` zurück. Ob die Ausgabe erfolgreich war, können/sollten Sie mit den Funktionen aus der Headerdatei `<ios>` überprüfen (siehe Abschnitt 13.4, »Fehlerbehandlung bzw. Zustand von Streams«).

13.3.2 Methoden für die (unformatierte) Eingabe

Beim Einlesen von rohen Daten mit Methoden wird bspw. im Gegensatz zum formatierten Einlesen mit `>>` ein beginnendes Leerzeichen (auch Whitespace-Zeichen genannt) nicht einfach übersprungen und ignoriert. Für die unformatierte Eingabe bietet die Headerdatei `<istream>` mehrere Methoden an, die auch teilweise mehrfach überladen sind. Daher bietet Tabelle 13.2 gleich eine Übersicht zu den Methoden.

Methode	Beschreibung
<code>istream& get(char& c);</code>	Übergibt ein einzelnes Zeichen aus dem Stream an <code>c</code> .
<code>istream& get(char* s, streamsize n);</code>	Hiermit werden <code>n-1</code> Zeichen aus dem Stream eingelesen und als C-String in <code>s</code> gespeichert. Werden weniger als <code>n-1</code> Zeichen verwendet, wird bis zum ersten Newline-Zeichen (<code>'\n'</code>) eingelesen. Das Newline-Zeichen wird allerdings nicht in <code>s</code> gespeichert. Das Terminierungszeichen (<code>'\0'</code>) wird automatisch am Ende hinzugefügt.

Tabelle 13.2 Methoden zur unformatierten Eingabe in `<istream>`

Methoden	Beschreibung
<code>istream& get(char* s, streamsize n, char del);</code>	Hiermit werden $n-1$ Zeichen aus dem Stream eingelesen und als C-String in <code>s</code> gespeichert. Werden weniger als $n-1$ Zeichen verwendet, wird bis zum ersten Begrenzungszeichen gelesen, welches in <code>del</code> enthalten ist. Das <code>del</code> -Zeichen wird allerdings nicht mit in <code>s</code> gespeichert. Das Terminierungs-Zeichen (<code>'\0'</code>) wird automatisch am Ende hinzugefügt.
<code>istream& getline(char* s, streamsize n, char del);</code>	Der bedeutendere Unterschied zu den entsprechenden <code>get()</code> -Versionen ist eigentlich nur, dass hier das Fehler-Flag <code>failbit</code> gesetzt wird, wenn mehr als $n-1$ Zeichen eingelesen wurden.
<code>istream& read(char* s, streamsize n);</code>	Hiermit werden n (nicht $n-1$) Zeichen aus dem Stream extrahiert und in den C-String <code>s</code> gespeichert. Im Gegensatz zu den <code>get</code> - und <code>getline</code> -Versionen wird die eingelesene Sequenz nicht mit dem Nullzeichen (<code>'\0'</code>) terminiert. Eine klassische Funktion zum Kopieren ganzer Blöcke von Daten.

Tabelle 13.2 Methoden zur unformatierten Eingabe in `<istream>` (Forts.)

Bei allen Versionen, bei denen Sie $n-1$ Zeichen vom Stream in die Adresse von `char* s` extrahieren, müssen Sie sicherstellen, dass in `s` auch genügend Speicherplatz vorhanden ist. Alle Methoden geben den Stream `istream` zurück. Ob die Eingabe erfolgreich war, können/sollten Sie mit dafür vorhandenen Mechanismen aus der Headerdatei `<ios>` überprüfen (siehe Abschnitt 13.4, »Fehlerbehandlung bzw. Zustand von Streams«). Das Extrahieren von Zeichen aus einem Stream wird bei allen Methoden zur Eingabe beendet, wenn *EOF* (*End of File*) gelesen wurde.

EOF (End of File)

EOF wird häufig mit dem Wert `-1` definiert und dient dazu, das Ende eines Datenstroms oder einer Datei zu kennzeichnen. In einer interaktiven Shell können Sie EOF auch mit einer Tastenkombination erzeugen. In klassischen Unix-Shells können Sie diese mit `[Strg]+[D]` reproduzieren. Bei der Eingabeaufforderung von Windows wird EOF mittels `[Strg]+[Z]` erzeugt.

Hierzu ein einfaches Listing, welches einige der Methoden für die unformatierte Eingabe demonstrieren soll:

```

00 // kapitel13/iostream/stream03.cpp
...
01 const unsigned int MAX = 10;
02 char buffer[MAX] = {0};

03 std::cout << "Eingabe getline : ";
04 std::cin.getline(buffer, MAX);
05 std::cout << std::cin.gcount()
    << " Zeichen wurden eingelesen\n";
06 for(auto c : buffer) {
07     if(c && c!='\0') std::cout.put(c);
08 }

09 std::cin.ignore(MAX, '\n');

09 std::cout << "\nEingabe machen (mit . beenden) : ";
10 char ch=0;
11 while( std::cin.get(ch)) {
12     if(ch == '.') break;
13     std::cout.put(ch);
14 }
15 std::cout << "Eingabe beendet" << std::endl;

```

In Zeile **04** extrahieren wir maximal `MAX` Zeichen aus dem Stream `cin` und speichern dies in `buffer` ab. Wie viele Zeichen tatsächlich aus dem Stream extrahiert wurden, können Sie sich mit der Methode `gcount()` vom Typ `streamsize` (gewöhnlich `size_t`) ausgeben lassen, wie dies im Beispiel in Zeile **05** zu sehen ist. Die Methode `gcount()` kann für alle anderen Methoden zur unformatierten Eingabe auch verwendet werden.

In den Zeilen **06** bis **08** geben wir die einzelnen Zeichen unformatiert mit der neuen C++11-Range-`for` und `put()` auf dem Bildschirm aus.

In Zeile **09** wurde die Methode `ignore()` verwendet. Damit werden in diesem Beispiel `MAX` Zeichen aus dem Eingabestrom bzw. bis zum Zeichen `'\n'` einfach überlesen und `'\n'` aus dem Puffer entfernt. Statt `'\n'` kann hier auch ein anderes Zeichen verwendet werden. Im Beispiel wurde diese Zeile verwendet, um das Newline-Zeichen aus dem Puffer zu »ziehen«, damit es nicht beim nächsten Einlesen in Zeile **11** verwendet wird und dadurch quasi einfach »übersprungen« werden kann. Der »Trick« funktioniert allerdings nicht überall. Auf anderen Systemen (bspw. beim Mac) können Sie statt Zeile **11** auch folgende Zeilen einfügen:

```

std::cin.clear();
std::cin.ignore(std::cin.rdbuf() -> in_avail());
std::cin.get();

```

In Zeile 11 bis 14 sehen das klassische Unix-Framework, womit einfach alle Zeichen aus dem Stream `cin` unformatiert eingelesen und auf dem Stream `cout` ausgegeben werden. Eingelesen wird so lange, bis ein Punkt (Zeile 12) eingegeben oder EOF produziert wurde.

Das Programm bei der Ausführung:

```
Eingabe getline : Hallo getline
9 Zeichen wurden eingelesen
Hallo get
Eingabe machen (mit . beenden) : Hallo get
Hallo get
Noch eine Zeile
Noch eine Zeile
Wir beenden mit einem Punkt.
Wir beenden mit einem Punkt
Eingabe beendet
```

std::string und getline()
In der Headerdatei `<string>` finden Sie ebenfalls eine Methode `getline()`, welche eine Zeile von einem Stream in einen `std::string` einliest.

13.4 Fehlerbehandlung bzw. Zustand von Streams

Bei den Streams reicht es oft nicht aus, einfach nur zu prüfen, ob etwas funktioniert (`true`) oder nicht funktioniert (`false`). Da hierbei verschiedene Arten von Fehlern auftreten können, bietet die I/O-Stream-Bibliothek hierfür Mechanismen an, um den Zustand des Streams zu prüfen. Hierfür finden Sie mit dem `iostate` aus `ios_state` ein paar Konstanten (siehe Tabelle 13.3).

Konstante	Bedeutung
<code>goodbit</code>	Keine Probleme mit dem Stream. Alle anderen Bits sind nicht gesetzt.
<code>eofbit</code>	End of File (EOF) ist gesetzt. Das Ende eines Datenstroms ist erreicht.
<code>failbit</code>	Fehler bei einer Ein-/Ausgabefunktion. Eine Ein-/Ausgabeoperation konnte nicht erfolgreich ausgeführt werden.
<code>badbit</code>	Schwerer Fehler. Der Stream befindet sich in einem undefinierten Zustand.

Tabelle 13.3 Konstanten in »iostate«

Die Konstanten sind `ios_base` als `public`-Mitglieder definiert und können entweder direkt über `ios_base` (bspw. `ios_base::goodbit`) oder über die von `ios_base` abgeleiteten Klassen bzw. erzeugten Objekte erreicht werden (bspw. `ios::goodbit` oder `cout.goodbit`).

Methoden für die Behandlung von Fehlern bei Streams

Um die einzelnen Zustände abzufragen, bietet Ihnen C++ verschiedene Methoden an. In Tabelle 13.4 finden Sie eine Übersicht dazu.

Methode	Bedeutung
<code>bool good();</code>	Gibt <code>true</code> zurück, wenn alles okay ist (<code>goodbit</code> ist gesetzt).
<code>bool eof();</code>	Gibt <code>true</code> zurück, wenn der Dateistrom zu Ende ist (<code>eofbit</code> ist gesetzt).
<code>bool fail();</code>	Gibt <code>true</code> zurück, wenn ein Fehler aufgetreten ist (<code>failbit</code> oder <code>badbit</code> sind gesetzt).
<code>bool bad();</code>	Gibt <code>true</code> zurück, wenn ein schwerer Fehler aufgetreten ist (<code>badbit</code> ist gesetzt).
<code>void clear();</code>	Setzt den Zustand von <code>iostate</code> zurück. Status wird auf <code>goodbit</code> gesetzt.
<code>void clear(iostate f);</code>	Setzt den Zustand von <code>iostate</code> zurück und den Status auf <code>f</code> .
<code>void setstate(iostate f);</code>	Ändert den aktuellen Zustand, in dem der vorhandene Status mit einer ODER-Verknüpfung mit <code>f</code> kombiniert wird.
<code>iostate rdstate();</code>	Gibt den aktuell gesetzten Status zurück.

Tabelle 13.4 Methoden zur Überprüfung des Stream-Zustandes

eofbit und failbit gesetzt

In der Regel ist `failbit` auch gesetzt, wenn `eofbit` gesetzt wurde. Dies kommt daher, weil der Zustand `eofbit` erst gesetzt wird, nachdem versucht wird, eine weitere Aktion hinter End of File durchzuführen. Das bedeutet dann allerdings auch, dass die nächste Operation fehlschlägt, und daher wird auch `failbit` gesetzt.

Hierzu nun ein einfaches Beispiel, welches verschiedene Streams auf deren Zustände prüft:

```

00 // kapitel13/iostream/stream04.cpp
01 #include <fstream>
...
02 void checkIOstate(std::ios_base& state) {
03     if( state.good() ) {
04         std::cout << "Alles in Ordnung\n";
05     }
06     else if( state.bad() ) {
07         std::cout << "Fataler Fehler\n";
08     }
09     else if( state.fail() ) {
10         std::cout << "Fehler bei der Ein-/Ausgabe\n";
11         if( state.eof() ) {
12             std::cout << "Ende des Datenstroms erreicht\n";
13         }
14     }
15     state.clear();
16 }
...
17 int val=0;
18 std::cout << "Wert eingeben: ";
19 std::cin >> val;
20 checkIOstate( std::cin );

21 std::ifstream file;
22 file.open("nichtvorhanden.text");
23 checkIOstate(file);

24 std::fstream fstr;
25 fstr.open("neueDatei.txt",
           std::ofstream::out|std::ofstream::in|
           std::ofstream::binary|std::ofstream::trunc);
26 fstr << "Text in der Datei\n";
27 fstr.seekp(std::ios_base::beg);

28 char ch;
29 while( fstr.good() ) {
30     fstr.get(ch);
31     if(fstr.good()) std::cout.put(ch);

```

```

32 }
33 checkIOstate(fstr);
...
```

In den Zeilen 02 bis 16 wurde eine globale Funktion `checkIOstate()` geschrieben, der Sie einen beliebigen Stream als Argument zur Überprüfung des Zustandes übergeben können. Daraufhin werden die einzelnen Methoden zur Klärung des Zustandes geprüft. Am Ende setzen wir in Zeile 15 den Zustand mit `clear()` wieder zurück auf `goodbit`.

In Zeile 19 werden Sie aufgefordert, einen Wert einzugeben. Korrekterweise muss hier ein Integer eingegeben werden. In Zeile 23 überprüfen wir den Stream `cin` daraufhin. Bei korrekter Eingabe ist `goodbit` gesetzt und `good()` (Zeile 03) gibt `true` zurück. Bei einer falschen Eingabe (bspw. wenn Sie ein Zeichen statt eines Integers eingegeben haben) gibt `fail()` (Zeile 09) `true` zurück.

In Zeile 23 überprüfen wir einen Datei-Stream `std::ifstream`, wo wir eine Datei zum Lesen öffnen wollten. Im Beispiel wird davon ausgegangen, dass die Datei nicht existiert, weshalb auch hier `fail()` gleich `true` ist.

Und zum Schluss in den Zeilen 28 bis 33 will ich Ihnen noch End of File demonstrieren. In den Zeilen 24 bis 27 erstellen wir eine Datei, schreiben etwas rein und setzen den Datenstrom wieder auf den Anfang der Datei. In der Schleife in den Zeilen 29 bis 32 lesen wir so lange Zeichen für Zeichen aus dem Stream `fstr`, solange der Zustand von `fstr` gleich `good()` (`goodbit` gesetzt) ist. Wenn der Zustand nicht mehr `good()` ist, wurde entweder das Ende des Datenstroms erreicht (`eofbit` ist gesetzt), oder es ist ein Fehler aufgetreten (`failbit` ist gesetzt). Wie bereits erwähnt, ist gewöhnlich `failbit` ebenfalls gesetzt, wenn `eofbit` gesetzt ist.

Ausnahme `ios_base::failure`

Bei Fehlern des Ein-/Ausgabesystems der I/O-Stream-Bibliothek kann es auch sein, dass die Ausnahme `ios_base::failure` geworfen wird (wenn `badbit` gesetzt wird). Dies hängt allerdings von der Implementierung des Compilerherstellers ab. Trotzdem ist es möglich, diese Ausnahme selbst mit `throw` zu werfen und abzufangen.

Streams mit `bool` überprüfen

Der Zustand des Streams kann auch mit `bool` und dem `!`-Operator getestet werden. Beide wurde als Methoden in `ios_base` implementiert (`operator bool()` und `operator !()`). Der Operator `bool` wurde mit C++11-Standard eingeführt und war vorher nur als `operator void(*)` implementiert. Beide Methoden geben zurück, ob ein Fehler aufgetreten ist (`failbit` oder `badbit` gesetzt).

Folgendermaßen wird bspw. der Operator `bool` verwendet:

```
00 // kapitel13/iostream/stream05.cpp
...
01 unsigned int val;
02 std::cout << "Wert eingeben: ";
03 std::cin >> val;
04 if( std::cin ) { // operator bool() (C++11)
05     // Eingabe korrekt
06 }
07 else {
08     // Fehler bei der Eingabe
09     std::cerr << "Fehler bei std::cin\n";
10 }
...
```

In Zeile **04** wird der Stream, weil dieser in einem `if` verwendet wird, automatisch nach `bool` konvertiert. Genau für solche Zwecke wurde in der Klasse der operator `bool()` definiert. Genauso einfach lässt sich der `!`-Operator auf Streams verwenden.

13.5 Streams manipulieren und formatieren

Wie die Zeichen in einem Stream ein- bzw. ausgegeben werden, wird durch sogenannte Flags (dt. Schalter) gesteuert, welche in der Basisklasse `ios_base` definiert sind. Flags können Sie sich wie ein Array von Bits vorstellen, wo die Werte eben 1 oder 0 sein können. Einzelne Bits machen natürlich noch keinen Wert, daher werden die Bits in einer Variablen gespeichert. Abhängig von den gesetzten Bits ändert sich auch der Wert der Variablen, und jeder Wert hat eine spezielle Bedeutung. Alle Flags haben eine Standardeinstellung, bei der eingestellt ist, dass bspw. alle integralen Werte ohne weitere Angaben in dezimaler (dec) Schreibweise ausgegeben werden.

Die Format-Flags können Sie direkt ändern, indem Sie Methoden wie `setf()`, `unsetf()` oder `flags()` aus der Klasse `ios_base` verwenden. Allerdings wurde bereits vor längerer Zeit (extra) eine bessere Alternative dafür mit den Manipulatoren zur Verfügung gestellt, womit Sie alle Flags direkt als »Funktion« in einen Ein- bzw. Ausgabe-Stream einfügen und aufrufen können. In der Praxis ist somit meistens der Einsatz von Manipulatoren dem direkten Zugriff auf die Format-Flags mit Methoden vorzuziehen. Daher soll zunächst auf die Manipulatoren eingegangen werden. Auf die Methoden, mit denen Sie die Format-Flags direkt verändern können, wird nur kurz in Abschnitt 13.5.4, »Format-Flags direkt ändern«, eingegangen.

13.5.1 Manipulatoren

Manipulatoren für Streams sind Objekte, welche Sie bei den Standard-Ein-/Ausgabe-Operatoren << bzw. >> einfügen können. Den Manipulator endl bspw. kennen Sie bereits. Dieser fügt dem Puffer ein Newline-Zeichen hinzu, um eine neue Zeile auszugeben. Die grundlegenden Manipulatoren finden Sie in <istream> bzw. <ostream> in Tabelle 13.5 aufgelistet.

Manipulator	Bedeutung
endl	Gibt eine neue Zeile aus und leert den Ausgabepuffer.
ends	Fügt dem Puffer das Terminator-Zeichen ('\0') hinzu.
flush	Entleert den Ausgabepuffer des Streams.
ws	Ignoriert führendes Leerzeichen beim Einlesen.

Tabelle 13.5 Grundlegende Manipulatoren in <istream> bzw. <ostream>

Unabhängige Manipulatoren in <ios>

Eine weitere beträchtliche Anzahl von Manipulatoren finden Sie in der Headerdatei <ios>. Hierbei handelt es sich um Manipulatoren, um bestimmte Zustände ein- bzw. auszuschalten oder um die Daten bei der Ausgabe zu formatieren. In Tabelle 13.6 bis Tabelle 13.9 finden Sie einen Überblick über diese Manipulatoren.

Manipulator	Bedeutung
boolalpha noboolalpha	Damit (de)aktivieren Sie, dass die booleschen Variablen wörtlich (true/false) oder numerisch (1/0) ein- bzw. ausgegeben werden können. Standardmäßig ist boolalpha nicht gesetzt.
showbase nowshowbase	Damit schalten Sie die Basis der Zahlen ein bzw. aus. Bei hexadezimaler Basis wäre dies 0x und bei einer oktalen Basis 0. Standardmäßig ist showbase nicht gesetzt.
showpoint nowshowpoint	Wenn eine Nachkommastelle mit Nullen endet, können Sie diese hiermit ein- bzw. ausblenden. Standardmäßig ist showpoint nicht gesetzt.
showpos nowshowpos	Bei positiven Zahlen kann hiermit das + davor ein- bzw. ausgeblendet werden. Standardmäßig ist showpos nicht gesetzt.
skipws noskipws	Damit können Sie führende Leerzeichen beachten oder ignorieren. Standardmäßig ist skipws gesetzt – also führende Leerzeichen werden ignoriert.

Tabelle 13.6 Unabhängige Manipulatoren zum Ein- bzw. Abschalten bestimmter Zustände aus der Headerdatei <ios>

Manipulator	Bedeutung
unitbuf nounitbuf	Leert nach jeder Operation den Puffer oder puffert eben. Standardmäßig ist unitbuf nicht gesetzt, und es wird gepuffert.
uppercase nouppercase	Groß- oder Kleinschreibung von e, x, a, b, c, d, e und f. Hat nichts zu tun mit Groß- oder Kleinschreibung von gewöhnlichen Zeichen. Standardmäßig ist uppercase nicht gesetzt.

Tabelle 13.6 Unabhängige Manipulatoren zum Ein- bzw. Abschalten bestimmter Zustände aus der Headerdatei `<iostream>` (Forts.)

Die Verwendung, diese Zustände ein- bzw. abzuschalten, ist recht einfach. Einmal in einem bestimmten Stream (de)aktiviert, bleibt der Zustand so lange bestehen, bis Sie wieder den gegensätzlichen Manipulator aufrufen. Ein einfaches Beispiel:

```
00 // kapitel13/iostream/stream06.cpp
...
01 void f() {
02     bool b=true;
03     std::cout << b << std::endl; // = true
04 }
...
05 bool b=true;
06 std::cout << std::boolalpha << b << std::endl; // = true
07 b=false;
08 std::cout << b << std::endl; // = false
09 f();
10 std::cout << std::noboolalpha << b << std::endl; // = 0
11 b=true;
12 std::cout << b << std::endl; // = 1
```

In Zeile **06** schieben wir `std::boolalpha` in den Stream `std::cout`, um `true` bzw. `false` von booleschen Variablen wörtlich auszugeben, was die Ausgabe in derselben Zeile auch gleich beweist. In Zeile **08** wird wieder der Wert der booleschen Variablen wörtlich ausgegeben. Daran können Sie schön erkennen, dass die Einstellung, einmal aktiviert, auch weiterhin aktiv bleibt.

Die Ausgabe in der Funktion `f()` in Zeile **03** demonstriert, dass die Einstellung auch global gilt. In Zeile **10** deaktivieren wir diese Einstellung wieder, indem hier `std::noboolalpha` in den Stream geschickt wird. Die Ausgabe der jetzt wieder numerischen Werte von 0 bzw. 1 statt `false` bzw. `true` von `bool` in den Zeilen **10** und **12** bestätigen diesen Vorgang.

Auch wenn es hier nicht verwendet wurde, können Sie dies durchaus bei der Eingabe mit dem Stream `std::cin` verwenden, bspw.:

```
01 bool b;
02 std::cin >> std::boolalpha >> b;
```

Hier können Sie jetzt tatsächlich `true` oder `false` eingeben, und es wird als boolescher Wert erkannt.

Manipulatoren für Ganzzahlen

Weitere Manipulatoren sind vorhanden, womit Sie das Format der Zahlenbasis zwischen dezimal, hexadezimal und oktal festlegen können. In Tabelle 13.7 finden Sie eine Übersicht zu den `basefield`-Flags aus der Headerdatei `<ios>`.

Manipulator	Bedeutung
dec	Das Lesen/Schreiben von integralen Werten erfolgt im dezimalen Zahlenformat. Dieser Zustand ist die Standardeinstellung.
hex	Das Lesen/Schreiben von integralen Werten erfolgt im hexadezimalen Zahlenformat.
oct	Das Lesen/Schreiben von integralen Werten erfolgt im oktalen Zahlenformat.

Tabelle 13.7 Manipulatoren in `<ios>` für das Zahlenformat von integralen Typen einstellen können

Hierzu die drei Manipulatoren für die Änderung des integralen Zahlenformats in der Praxis:

```
00 // kapitel13/iostream/stream07.cpp
...
01 void f() {
02     int val = 100;
03     std::cout << val << std::endl; // = 0x64
04 }
...
05 int val = 255;
06 std::cout << std::showbase;
07 std::cout << std::dec << val << std::endl; // =255
08 std::cout << std::hex << val << std::endl; // =0xff
09 f();
10 std::cout << std::oct << val << std::endl; // =0377
11 std::cout << val << std::endl;           // =0377
```

In Zeile **06** wird der Manipulator `std::showbase` verwendet, damit anschließend auch die Zahlenbasis `0x` für hexadezimale bzw. `0` für oktale Werte ausgegeben wird. Die Verwendung des Manipulators `dec` in Zeile **07** hätten Sie sich auch sparen können, weil es die Standardeinstellung ist und der Wert ohnehin in dezimaler Form ausgegeben wird. In Zeile **08** wird der Wert, dank des Manipulators `hex`, in hexadezimaler Schreibweise ausgegeben. Die Einstellung bleibt dann für den Stream global erhalten, wie die Ausgabe der Funktion `f()` in Zeile **03** bestätigt. In Zeile **10** wird der Wert mithilfe des Manipulators `oct` in oktaler Darstellung ausgegeben. Die Ausgabe in Zeile **11** zeigt nochmals, dass die Einstellung für den Stream so bleibt. Wollen Sie wieder die dezimale Schreibweise haben, müssen Sie den Manipulator `dec` verwenden.

Manipulatoren für Gleitpunktzahlen

Auch für Gleitpunktzahlen gibt es einige Manipulatoren (`floatfield`-Flags), die ebenfalls in `<ios>` enthalten sind und in Tabelle 13.8 aufgelistet werden.

Manipulator	Bedeutung
<code>fixed</code>	Ausgabe von Gleitpunktzahlen in der Gleitkommaschreibweise (bspw. 3.14159)
<code>scientific</code>	Ausgabe von Gleitpunktzahlen in der Exponential-Schreibweise (bspw. 3.14159e+000)
<code>hexfloat</code>	Schreibt den Wert von Gleitpunktzahlen im hexadezimalen Format (bspw. 0x1.921f901b866ep+1). Neu hinzugefügt in C++11.
<code>defaultfloat</code>	Schreibt den Wert in der eingestellten Standard-Gleitkommaschreibweise. Das ist die Standardeinstellung, bevor diese mit einem anderen Wert geändert wird. Neu dabei in C++11.

Tabelle 13.8 Manipulatoren für Gleitkommazahlen in der Headerdatei `<ios>`

Manipulatoren zum Ausrichten

Zu guter Letzt gibt es noch diverse Manipulatoren, mit denen sich die Ausgabe des Streams ausrichten lässt. Eine Übersicht zu diesen `adjustfield`-Flags, welche ebenfalls in `<ios>` definiert ist, finden Sie in Tabelle 13.9.

Manipulator	Bedeutung
<code>internal</code>	Füllt den Raum zwischen Vorzeichen und Wert auf.
<code>left</code>	Richtet die Ausgabe linksbündig aus.
<code>right</code>	Richtet die Ausgabe rechtsbündig aus.

Tabelle 13.9 Manipulatoren für das Ausrichten der Ausgabe in `<ios>`

Das Verwenden dieser Manipulatoren zum Ausrichten der Ausgabe ist natürlich nur dann sinnvoll, wenn die Ausgabe wenigstens mit `setw()` aus der Headerdatei `<iomanip>` auf eine bestimmte Füllweite eingerichtet wurde. Zum besseren Verständnis ein Beispiel mit der entsprechenden Ausgabe auf dem Bildschirm dazu:

```
00 // kapitel13/iostream/stream08.cpp
...
01 int val = -1000;
02 std::cout << std::setw(10) << std::internal
    << val << std::endl;
03 std::cout << std::setw(10) << std::left << val << std::endl;
04 std::cout << std::setw(10) << std::right
    << val << std::endl;
```

Das Programm bei der Ausführung spricht für sich selbst:

```
-      1000
-1000
      -1000
```

Manipulatoren mit Argumenten

Es gibt einige Manipulatoren mit Argumenten, welche in der Headerdatei `<iomanip>` definiert sind. Einen Überblick darüber finden Sie in Tabelle 13.10 aufgelistet.

Manipulator	Bedeutung
<code>setiosflags(ios::fmtflags m)</code>	Damit können Sie mehrere Flags mit einem bitweisen verknüpfen und auf <code>m</code> setzen.
<code>resetiosflags(ios::fmtflags m)</code>	Löscht alle mit <code>m</code> angegebenen Flags wieder.
<code>setbase(int base)</code>	Damit können Sie als Basis der integralen Werte auch 8 (= oktal), 10 (= dezimal) und 16 (= hexadezimal) setzen.
<code>setfill(char ch)</code>	Wird eine Füllweite verwendet, wird der leere Raum mit dem Zeichen <code>ch</code> gefüllt.
<code>setprecision(int n)</code>	Damit können Sie die Nachkommastellen für Gleitkommazahlen festlegen.
<code>setw(int w)</code>	Damit können Sie eine Füllweite zwischen zwei Werten in einem Stream festlegen.

Tabelle 13.10 Manipulatoren mit Parameter aus der Headerdatei `<iomanip>`

Hierzu wieder einige Beispiele, welche diese Manipulatoren mit Argument in der Praxis demonstrieren sollen:

```
00 // kapitel13/iostream/stream09.cpp
01 #include <iomanip>
...
02 double dval = 3.14159;
03 std::ios_base::fmtflags ff(
    std::ios::scientific|std::ios::uppercase);
04 std::cout << std::setiosflags(ff);
05 std::cout << dval << std::endl;
06 std::cout << std::resetiosflags(ff) << dval << std::endl;
07 std::cout << std::setprecision(3) << dval << std::endl;

08 std::cout << std::setw(10);
09 std::cout << std::setfill('*') << 1246 << std::endl;
```

In Zeile **03** speichern wir gleich ein paar Format-Flags in der Bitmaske vom Typ `ios_base::fmtflags`. Mehrere Flags können hier mit dem `|`-Operator verknüpft werden. In Zeile **04** setzen wir die Flags so, dass eine Gleitkommazahl im Exponenten-Format und das `e` hierbei großgeschrieben wird. Sie müssen nicht unbedingt `ios_base::fmtflags` anlegen und können die Flags auch direkt in `setiosflags` reinschreiben. In Zeile **06** setzen wir diese Vorgaben aus Zeile **04** wieder zurück. In Zeile **07** wird `setprecision()` demonstriert, womit die Gleitkommazahl auf drei Ziffern abgeschnitten wird. Das betrifft aber nur die Ausgabe. Der Wert `dval` wird hierbei nicht verändert. In Zeile **09** wird noch `setfill()` demonstriert, indem wir in der Zeile zuvor die Füllweite mit `setw()` auf zehn Zeichen gesetzt haben. Der Leerraum der Füllweite wird hierbei jetzt mit dem Zeichen `*` aufgefüllt.

Das Programm bei der Ausführung:

```
3.141590E+00
3.14159
3.14
*****1246
```

setw() ist anders

Beachten Sie bitte, dass der Manipulator `setw()` die Einstellung der Füllweite nicht permanent macht, wie dies bei allen anderen Manipulatoren der Fall ist. Die Füllweite wird nach jeder Verwendung zurückgesetzt und muss daher vor jeder Verwendung erneut gesetzt werden.

Neu in C++11

In C++11 sind mit `get_money()`, `put_money()`, `get_time()` und `put_time()` vier weitere Manipulatoren hinzugekommen, womit Sie landesspezifische Dinge wie die Ein-/Ausgabe der Währung von Geld bzw. Datum und Uhrzeit verwenden können.

13.5.2 Eigene Manipulatoren ohne Argumente erstellen

Sie können natürlich auch eigene Manipulatoren erstellen. In der Praxis müssen Sie hierbei lediglich den Stream als Argument und einen Stream als Referenz auf diesen Stream wieder zurückgeben. Dazwischen können Sie den Stream manipulieren (daher auch der Name Manipulator). Eigene Manipulatoren ohne Argumente lassen sich mit ein paar Zeilen Code implementieren:

```
00 // kapitel13/iostream/stream10.cpp
...
01 std::ostream& tabl(std::ostream& os) {
02     os << '\t';
03     return os;
04 }

05 std::istream& firstNum(std::istream& is) {
06     char ch;
07     is.get(ch);
08     if( (ch >= '0') && (ch <= '9') ) {
09         std::cin.putback(ch);
10     }
11     return is;
12 }
...
13 int val=0;
14 std::cout << "Text1" << tabl << "Text2" << std::endl;
15 std::cout << "Eingabe machen: ";
16 std::cin >> firstNum >> val;
17 std::cout << val << std::endl;
```

In den Zeilen **01** bis **04** manipulieren wird den `ostream`. Diese Funktion können Sie praktisch durch einen `ostream`, oder genauer durch den `operator<<()` durchlaufen lassen, weil ein `ostream` als Argument und der `ostream` als Referenz auch wieder zurückgegeben wird. Innerhalb der Funktion können Sie jetzt den Stream beliebig ändern. Sie können hierbei bspw. auch einen der vielen Manipulatoren verwenden,

welche die Standardbibliothek in `<ostream>` anbietet. Im Beispiel haben wir einen einfachen Manipulator `tabl` geschrieben, welcher einen Tabulator-Vorschub erzeugt, wie in Zeile 14 auch demonstriert wird.

Andersherum geht es natürlich auch wieder, wie die Zeilen 05 bis 12 demonstrieren sollen, wo Sie einen `istream` als Argument und diesen `istream` als Referenz zurückgeben. Innerhalb der Funktion können Sie jetzt den Eingabe-Stream manipulieren. Es ist nicht einfach, ein sinnvolles Beispiel dazu zu schreiben. In diesem Beispiel machen wir etwas Ähnliches wie `std::cin` beim Einlesen von einem Leerzeichen am Anfang macht, welches ignoriert wird. Wir prüfen in diesem Beispiel beim ersten Zeichen von `istream`, ob es einem der Zeichen 0 bis 9 entspricht. Ist dies der Fall, schieben wir es zurück in den Eingabe-Stream (Zeile 09). Handelt es sich nicht um eines der Zeichen 0 bis 9, wurde ein falsches erstes Zeichen eingegeben, und wir lassen es einfach »verschluckt«. Der Manipulator `firstnum` wird in Zeile 16 demonstriert.

Das Programm bei der Ausführung:

```
Text1          Text2
Eingabe machen: x12345
12345
```

Dass Sie einen selbst geschriebenen Manipulator durch die Ein-/Ausgabe-Operatoren schieben können, liegt an der (komplexen) Implementierung der Operatoren:

```
01 ostream& ostream::ostream<<(ostream& (*fp) (ostream& ) ) {
02     *fp(*this);
03     return *this;
04 }
```

Dank des Funktionszeigers `*fp` in Zeile 02 werden unsere selbst geschriebenen Manipulatoren im Stream aufgerufen. Beim Eingabe-Stream ist dies dann genauso.

13.5.3 Eigene Manipulatoren mit Argumenten erstellen

Auch Manipulatoren mit Argumenten können Sie ohne allzu großen Aufwand mit einem *Funktor* (alias Funktionsobjekt) realisieren.

Funktionszeiger -> Funktionsobjekt

Ein Funktor ist letztendlich nichts anderes als ein Funktionsobjekt. Und Funktionsobjekte wiederum sind eine Erweiterung der Funktionszeiger aus alten Zeiten. In C++ wird statt eines Funktionszeigers eben ein Funktionsobjekt mit dem `operator()` definiert.

Hierzu ein einfaches Beispiel in der Praxis, wie Sie Manipulatoren mit Parameter schreiben können:

```
00 // kapitel13/iostream/stream11.cpp
...
01 class nendl {
02     int lines;
03 public:
04     nendl(int n=1) : lines(n){}
05     std::ostream& operator()(std::ostream& os) const {
06         for(int i=0; i<lines; ++i) {
07             os << '\n';
08         }
09         return os;
10     }
11 };

12 std::ostream& operator<<( std::ostream& os,
                           const nendl& line){
13     return line(os);
14 }
...
15 std::cout << "Text1" << nendl(4);
16 std::cout << "Text2" << nendl(2);
17 std::cout << "Text3" << nendl();
...
```

In Zeile **01** bis **11** sehen Sie die typische Klassendefinition, welche für die Definition des Funktionsobjekts in den Zeilen **05** bis **10** benötigt wird. Das Funktionsobjekt selbst macht nichts anderes, als n-Mal eine neue Zeile (*Newline*) auszulösen – ein `std::endl` für mehrere Zeilen, wenn Sie es so wollen. Wenn Sie die Anwendung in Zeile **15** sehen, werden Sie feststellen, dass eine Verwendung mit dem Funktionsobjekt alleine noch nicht möglich ist. Der Aufruf von

```
std::cout << nendl(4);
```

sieht als Funktion geschrieben wie folgt aus:

```
operator<<(std::cout, nendl(4));
```

Und genau so eine *Funktion* wurde in den Zeilen **12** bis **14** geschrieben. Auch hier muss der Stream als Referenz übergeben und wieder zurückgegeben werden. Und zusätzlich muss hier noch mit dem Funktionsobjekt gearbeitet werden. Der eigentliche Funktionsaufruf erfolgt daher in Zeile **13**, wo aus `line(os)` das Funktionsobjekt

mit `line.operator(os)` aufgerufen und die Manipulation auf dem Stream durchgeführt wird.

Das Programm bei der Ausführung:

Text1

Text2

Text3

Die Implementierung eines Manipulators mit Argumenten für den Eingabe-Stream lässt sich ähnlich implementieren, wie eben gezeigt. Allerdings wurde hier darauf verzichtet, damit der Code nicht unnötig lang wird.

Referenz als Rückgabewert

Auch hier empfiehlt es sich immer, eine Referenz als Rückgabewert zu verwenden, weil Sie damit sicherstellen können, dass weitere Operationen mit dem Operator verkettet werden können.

13.5.4 Format-Flags direkt ändern

Neben der Formatierung von Streams mithilfe von Manipulatoren können Sie auch noch eine direkte Veränderung der Flags mit Methoden durchführen. Zwar ist die Verwendung von Manipulatoren in der Regel gängiger und erheblich komfortabler, aber trotzdem soll hier kurz auf die vorhandenen Methoden eingegangen werden. Der Unterschied ist auch recht schnell erklärt. Betrachten Sie folgenden Codeausschnitt, in dem wir Manipulatoren verwenden, um die Ausgabe zu formatieren:

```
00 // kapitel13/iostream/stream12.cpp
...
01 int val = 255;
02 std::cout << std::showbase << std::hex
    << val << std::endl;
03 std::cout << std::noshowbase << std::dec
    << val << std::endl;
```

In Zeile **02** wird durch die beiden Manipulatoren der hexadezimale Wert (`ff`) mitsamt der Zahlenbasis (`0x`) ausgegeben. In Zeile **03** wurde die Ausgabe der Zahlenbasis wieder deaktiviert und als dezimaler Wert ausgegeben. Die Ausgabe des Programms:

```
0xff
255
```

Die hier verwendeten Manipulatoren sind gleichwertig zu den jetzt folgenden Anweisungen, wo die Flags direkt mit Methoden geändert wurden:

```
00 // kapitel13/iostream/stream13.cpp
...
01 int val = 255;
02 std::cout.setf(std::ios_base::hex,
    std::ios_base::basefield);
03 std::cout.setf(std::ios_base::showbase);
04 std::cout << val << std::endl;

05 std::cout.unsetf(std::ios_base::showbase);
06 std::cout.setf(std::ios_base::dec,
    std::ios_base::basefield);
07 std::cout << val << std::endl;
```

Hier geschieht dasselbe wie im Beispiel von Listing *stream12.cpp*. Nur wird hier in Zeile **02** zunächst das Bit für die hexadezimale Zahlendarstellung mit `hex` in der Gruppe `basefield` gesetzt. Damit auch die Zahlenbasis (0x) mit ausgegeben wird, setzen wir dieses Flag in Zeile **03** für den Stream `cout`. Dasselbe machen wir in der umgekehrten Reihenfolge in Zeile **05**, wo wir das Bit `showbase` wieder löschen. In Zeile **06** setzen wir noch das Flag in der Gruppe `basefield` zurück auf die dezimale Ausgabe (`dec`) von Ganzzahlen.

Ich denke, das Beispiel zeigt sehr schön die übersichtlichere und komfortable Verwendung von Manipulatoren im Gegensatz zu den hier verwendeten Methoden `setf()` und `unsetf()`. Bei Manipulatoren müssen Sie sich nicht mit sperrigen Dingen wie Flags, Bits und Bitmasken herumschlagen. Und letztendlich ruft ein Manipulator wie `std::showbase` ohnehin intern die Methode `std::cout.setf(std::ios::showbase)` auf.

Erschwerend kommt noch hinzu, dass einige Format-Flags auch in Gruppen verwendet werden können, wie Sie im Beispiel von `dec` mit `basefield` gesehen haben. Genommen sind es drei Gruppen mit `adjustfield` (und den Flags `left`, `right`, `internal`), `basefield` (mit den Flags `dec`, `oct`, `hex`) und `floatfield` (mit den Flags `scientific` und `fixed`).

Unabhängige Flags, wie bspw. `showbase` (siehe auch Tabelle 13.6), können alleine verwendet werden (allerdings mit `ios_base::` bzw. `ios::`). In Tabelle 13.11 finden Sie einen Überblick über die Methoden in der Klasse `ios_base`, womit Sie die Format-Flags direkt ändern können.

Methode	Bedeutung
setf(flags)	Setzt die Format-Flags auf flags.
setf(flags, group)	Setzt die Format-Flags der Gruppe group auf flags.
unsetf(flags)	Löscht alle flags.
flags()	Gibt die aktuell gesetzten Format-Flags zurück.
flags(flags)	Setzt die Format-Flags auf flags.

Tabelle 13.11 Methoden für den Zugriff auf Format-Flags

Hierzu nochmals dasselbe Beispiel wie oben, nur soll alles jetzt allein mit der Methode `flags()` realisiert werden:

```
00 // kapitel13/iostream/stream14.cpp
...
01 int val = 255;
02 std::ios::fmtflags ff = std::cout.flags();
03 std::cout.flags(std::ios::hex | std::ios::showbase);
04 std::cout << val << std::endl;
05 std::cout.flags(ff);
06 std::cout << val << std::endl;
```

In Zeile **02** holen wir den aktuellen Zustand der Format-Flags und speichern diesen in `ff`. In Zeile **03** setzen wir die Flags `hex` und `showbase` für den Stream `cout`, was sich in der Ausgabe in Zeile **04** dann auch auswirkt. In Zeile **05** verwenden wir den in Zeile **02** gesicherten Zustand der Format-Flags und stellen diesen wieder her, was die Ausgabe in Zeile **06** auch wieder bestätigt.

Weitere Methoden zur Formatierung

Auch zu einigen Manipulatoren mit Parametern, um vorwiegend die Feldausgabe (bzw. Füllweite) zu formatieren, finden Sie passende Methoden in der Klasse `ios_base` (Tabelle 13.12).

Methode	Beschreibung
<code>int fill() const;</code>	Gibt das aktuell gesetzte Füllzeichen zurück.
<code>int fill(int ch);</code>	Setzt das Füllzeichen auf <code>ch</code> . Entspricht dem Manipulator <code>setfill(ch)</code> .

Tabelle 13.12 Weitere Methoden der Klasse »ios_base« zur Formatierung

Methode	Beschreibung
<code>int width() const;</code>	Gibt die Füllweite zurück.
<code>int width(int n);</code>	Setzt die Füllweite auf <code>n</code> . Entspricht dem Manipulator <code>setw(n)</code> .
<code>int precision() const;</code>	Gibt den aktuellen Wert der Genauigkeit von Gleitpunktzahlen zurück.
<code>int precision(int n);</code>	Setzt die Genauigkeit von Gleitpunktzahlen auf <code>n</code> . Entspricht dem Manipulator <code>setprecision(n)</code> .

Tabelle 13.12 Weitere Methoden der Klasse »ios_base« zur Formatierung (Forts.)

width() ist anders

Hier gilt dasselbe wie für den Manipulator `setw()`, wo die Einstellung der Füllweite nicht permanent eingestellt bleibt, wie dies bei allen anderen Flags der Fall ist. Die Füllweite wird nach jeder Verwendung zurückgesetzt und muss daher vor jeder Verwendung erneut gesetzt werden.

13.6 Streams für die Datei-Ein-/Ausgabe

Ihre bisherige Interaktion mit dem Rechner spielte sich bisher vorwiegend im Arbeitsspeicher des Systems ab. Dort haben Sie Variablen abgelegt und bei Bedarf wieder darauf zugegriffen. Sobald das Programm allerdings beendet ist, sind die Daten weg. In diesem Abschnitt werden Sie jetzt erfahren, wie Sie die Daten dauerhaft auf einem externen Speicher sichern und wieder darauf zugreifen können.

Bevor Sie Daten in Dateien speichern und wieder laden können, will ich Ihnen noch kurz den allgemeinen Zugriff auf Dateien erklären. Um auf Dateien zuzugreifen, werden natürlich entsprechende Funktionalitäten benötigt. Diese werden Ihnen von der C++-Standardbibliothek zur Verfügung gestellt. Gelesen und geschrieben wird von bzw. in Dateien natürlich nicht Byte für Byte, sondern (aus Effizienzgründen) in ganzen Blöcken. Für eine solche Datenübertragung wird ein Puffer als Zwischenspeicher im Arbeitsspeicher verwendet. Die Datei oder den Inhalt einer Datei können Sie sich im Grunde auch wieder als ein Array von Bytes mit einer bestimmten Länge vorstellen.

So wie Sie die Daten in eine Datei schreiben, müssen Sie diese auch wieder auslesen. Das heißt, um die Formatierung und Strukturierung müssen Sie sich selbst kümmern. Damit das Lesen bzw. Schreiben von Dateien auch sauber der Reihe nach funktioniert, gibt es eine Dateiposition (angefangen bei 0). Mithilfe dieser Dateiposition

ist sichergestellt, dass eine Datei sequenziell abgearbeitet werden kann. Wenn Sie bspw. 10 Bytes aus einer Datei gelesen haben, befindet sich auch die Leseposition auf dem zehnten Byte. Selbiges gilt natürlich auch bei einer Schreiboperation.

13.6.1 Streams für die Datei-Ein-/Ausgabe

Die Streams für die Datei-Ein-/Ausgabe wurden nahtlos in das Stream-Konzept der Standardbibliothek eingebaut. Wenn Sie einen Blick auf den Anfang des Kapitels und auf Abbildung 13.1 mit der vereinfachten Klassenhierarchie werfen, können Sie sehr schön erkennen, dass die Datei-Streams `ifstream`, `ofstream` und `fstream` alle bekannten Klassen als Basisklasse haben. Dies bedeutet, dass Sie alle bekannten Operationen zur Ein-/Ausgabe (wie bspw. die Operatoren `<<` und `>>`) auch bei den Dateien verwenden können. Im Grunde ändert sich nur das Stream-Objekt. Statt `cout` oder `cin` stehen hierfür jetzt eben andere Stream-Objekte, welche mit einer Datei verbunden sind. Hierzu eine kurze Übersicht zu den drei vorhandenen Datei-Stream-Klassen:

- ▶ `ifstream` ist von `istream` abgeleitet und wird für das Lesen von Dateien verwendet.
- ▶ `ofstream` ist von `ostream` abgeleitet und dient dem Schreiben in Dateien.
- ▶ `fstream` wurde von `iostream` abgeleitet und kann zum Lesen und Schreiben in Dateien verwendet werden.

Für alle Datei-Streams müssen Sie die Headerdatei `<fstream>` im Programm angeben.

13.6.2 Verbindung zu einer Datei herstellen

Um überhaupt etwas mit einer Datei anfangen zu können, müssen Sie zunächst ein entsprechendes Stream-Objekt vom Typ `ifstream`, `ofstream` oder `fstream` instanziiieren. Des Weiteren müssen Sie die Datei öffnen. Das Öffnen der Datei können Sie entweder direkt bei der Instanziierung mit dem entsprechenden Konstruktor machen oder mit der Methode `open()`, indem Sie dort den Dateinamen (optional mit Pfad) angeben. Der Dateiname kann als C-String und seit C++11 auch `std::string` angegeben werden. Ebenfalls optional können Sie einen Modus angeben, wie Sie die Datei öffnen wollen.

Keine Pfadangabe

Verwenden Sie keine Pfadangabe, wird davon ausgegangen, dass sich die Datei im selben Verzeichnis befindet, in dem auch das Programm ausgeführt wird.

Hierzu einige Beispiele, in denen ein paar Datei-Streams angelegt und Dateien geöffnet werden:

```

00 // kapitel13/fstream/fstream01.cpp
01 #include <fstream>
...
02 std::string name;
03 std::cout << "Datei zum Lesen öffnen: ";
04 std::cin >> name;

05 std::ifstream file01;
06 file01.open(name.c_str());
07 if( file01.fail() ) {
08     std::cerr << "Konnte " << name << " nicht öffnen\n";
09 }
10 std::ofstream file02("data.dat");
11 if( file02.good() ) {
12     std::cerr << "data.dat geöffnet bzw. erzeugt\n";
13 }
14 std::fstream file03;
15 file03.open("database.db");
16 if( !file03 ) {
17     std::cerr << "Konnte database.db nicht öffnen\n";
18 }

```

In Zeile **05** wird mit dem Standardkonstruktor ein `ifstream`-Objekt zum Lesen angelegt. Das so erzeugte Stream-Objekt `file01` ist mit noch keiner Datei verbunden. Dies erledigen wir in Zeile **06** mit der Methode `open()` und dem Dateinamen, welchen Sie zuvor in den Zeilen **02** bis **04** eingegeben haben. Die Umwandlung von `std::string` in einen C-String mit der Methode `c_str()` können Sie sich sparen, wenn Ihr Compiler den neuen C++11-Standard kann. Ob die Datei geöffnet werden konnte, überprüfen wir in Zeile **07** mit der Methode `fail()`.

In Zeile **10** legen Sie ein neues Stream-Objekt zum Schreiben an. Den Dateinamen geben wir hier gleich im Konstruktor mit an. Existiert diese Datei nicht, wird sie angelegt. Existiert diese Datei hingegen, wird sie in der Standardeinstellung auf die Länge 0 gesetzt und der alte Inhalt überschrieben. In Zeile **11** prüfen wir, ob die Datei erfolgreich geöffnet bzw. angelegt werden konnte.

Im letzten Beispiel in Zeile **14** öffnen wir eine Datei `database.db` zum Lesen und Schreiben. Wenn die Datei nicht existiert, wird die Überprüfung in Zeile **16** `true` zurückgeben und eine entsprechende Fehlermeldung auf dem Bildschirm ausgegeben.

Fehlerüberprüfung

Zur Fehlerüberprüfung beim Öffnen einer Datei können Sie somit sowohl `if(!file)` als auch `if(file.fail())` verwenden. Beide Male wird `true` zurückgegeben, wenn das Öffnen einer Datei fehlgeschlagen ist und somit das Flag `failbit` gesetzt wird (siehe

dazu auch Abschnitt 13.4, »Fehlerbehandlung bzw. Zustand von Streams«). Seit C++11 werden bei erfolgreicher Öffnung einer Datei alle Fehler-Flags gelöscht und das Flag `goodbit` gesetzt. Daher können Sie jetzt auch mit der Methode `good()` testen, ob die Datei erfolgreich geöffnet werden konnte (`true`) oder nicht (`false`).

Flags zum Öffnen einer Datei

Beim Öffnen eines Datei-Streams können Sie es jetzt entweder bei den Standardwerten der Klassen `fstream`, `ofstream` oder `ifstream` belassen, oder aber Sie setzen optionale Flags, womit Sie festlegen wie die Datei geöffnet wird. In der Headerdatei `<ios>` finden Sie hierzu den Typ (genauer die Bitmaske) `ios::openmode` definiert. Die einzelnen Flags (siehe Tabelle 13.13) können Sie, sofern sinnvoll, mit dem bitweisen ODER-Operator (`|`) auch miteinander verknüpfen.

Flag	Kurz für	Beschreibung
<code>ios::in</code>	input	Öffnen zum Lesen (Standard von <code>ifstream</code>)
<code>ios::out</code>	output	Öffnen zum Schreiben (Standard von <code>ofstream</code>)
<code>ios::app</code>	append	Immer am Ende etwas hinzufügen, wenn geschrieben wird
<code>ios::trunc</code>	truncate	Die Dateilänge wird auf 0 gesetzt, und vorhandener Inhalt wird gelöscht.
<code>ios::ate</code>	at end	Positioniert die Lese- bzw. Schreibposition gleich nach dem Öffnen am Ende der Datei. Ohne dieses Flag befindet sich die Lese- und/oder Schreibposition nach dem Öffnen gewöhnlich am Anfang der Datei.
<code>ios::binary</code>	binary	Öffnet die Datei im rohen Binärmodus. Umwandlungen vom Zeilenende werden dann bspw. nicht durchgeführt, und es werden rohe Bytes übertragen. Ohne diesen Modus wird eine Datei in einem formatierten »Textmodus« geöffnet.

Tabelle 13.13 Verschiedene Flags zum Öffnen einer Datei

Standardwerte von `ifstream`, `ofstream` und `fstream`

Der Standardwert von `ifstream` ist sinngemäß `ios::in`, der von `ofstream` ist `ios::out|ios::trunc` und von `fstream` gleich `ios::in|ios::out`.

Die Flags können Sie entweder direkt im Konstruktor hinter dem Dateinamen oder in der Methode hinter dem Dateinamen schreiben. Hierzu zwei einfache Beispiele:


```
00 // kapitel13/fstream/fstream02.cpp
01 #include <fstream>
...
02 std::string name;
03 std::cout << "Datei zum Schreiben eingeben: ";
04 std::cin >> name;
05 std::ofstream file01(name, std::ios::out|std::ios::app);
06 if( file01.fail() ) {
07     std::cerr << "Konnte " << name << " nicht öffnen\n";
08 }

09 std::fstream file02;
10 file02.open("database.db", std::ios::out|std::ios::trunc);
11 if( !file02 ) {
12     std::cerr << "Konnte database.db nicht öffnen\n";
13 }
```

In Zeile 05 öffnen wir eine Datei zum Schreiben (`ios::out`) und hängen den neuen Inhalt am Ende der Datei an (`ios::out`). Wenn Ihr Compiler den C++11-Standard noch nicht kann, müssen Sie den Dateinamen hier noch mit `name.c_str()` in einen C-String umwandeln. In Zeile 10 öffnen wir ebenfalls eine Datei zum Schreiben (`ios::out`), wobei hier der Inhalt der Datei gelöscht wird (`ios::trunc`). Existiert diese Datei noch nicht, wird sie angelegt.

Verwirrung entsteht häufig, wann welche Flags wie kombiniert werden sollten. Definitiv lässt sich hierzu sagen, dass Sie immer mindestens die Flags `ios::in` bzw. `ios::out` verwenden sollten, wie es auch die Standardeinstellung der Klassen macht. Ebenso muss für `ios::in` die Datei bereits existieren. Wenn Sie kein `ios::in` verwenden, wird die Datei neu angelegt, wenn diese nicht vorhanden ist. In Tabelle 13.14 finden Sie einen kurzen Überblick über verschiedene Kombinationen von Flags und was diese bewirken.

Flags	Bedeutung
<code>ios::in</code>	Lesen; Datei muss existieren.
<code>ios::out</code>	Leeren und Schreiben; Datei wird erzeugt, falls nicht vorhanden.
<code>ios::out ios::trunc</code>	Leeren und Schreiben; Datei wird erzeugt, falls nicht vorhanden.
<code>ios::out ios::app</code>	Daten am Ende hinzufügen; Datei wird erzeugt, falls nicht vorhanden.

Tabelle 13.14 Bedeutungen von verschiedenen Kombinationen zum Öffnen einer Datei

Flags	Bedeutung
<code>ios::app</code>	Daten am Ende hinzufügen; Datei wird erzeugt, falls nicht vorhanden.
<code>ios::in ios::out</code>	Lesen und Schreiben; Datei muss existieren; Dateiposition am Anfang
<code>ios::in ios::out ios::trunc</code>	Leeren, Lesen und Schreiben; Datei wird erzeugt, falls nicht vorhanden.
<code>ios::in ios::app</code>	Lesen und Schreiben; Daten am Ende hinzufügen; Datei wird erzeugt, falls diese nicht existiert.
<code>ios::in ios::out ios::app</code>	Lesen und Schreiben; Daten am Ende hinzufügen; Datei wird erzeugt, falls diese nicht existiert.

Tabelle 13.14 Bedeutungen von verschiedenen Kombinationen zum Öffnen einer Datei

Datei schließen

Wenn Sie mit der Datei fertig sind, sollten Sie die Ressourcen dem System wieder zurückgeben und die Datei schließen. Zum einen stehen Ihnen nicht unendlich viele Datei-Streams zur Verfügung, und zum anderen stellen Sie so sicher, dass Daten, die sich noch im Puffer befinden, endgültig abgearbeitet werden. Zum Schließen einer Datei wird die Methode `close()` verwendet. Folgender Codeausschnitt soll `close()` kurz demonstrieren:

```

00 // kapitel13/fstream/fstream03.cpp
01 #include <fstream>
...
02 std::ofstream file01("data.db");
03 if( file01.fail() ) {
04     std::cerr << "Konnte data.db nicht öffnen\n";
05 }
06 file01 << "Text für die Datei\n";
07 if( file01.is_open() ) {
08     file01.close();
09 }
```

In Zeile **08** schließen wir die Verbindung mit der Datei. Damit wir nicht versehentlich eine Datei schließen, welche gar nicht (mehr) geöffnet ist, verwenden wir die Methode `is_open()` in Zeile **07**, welche überprüft, ob das Stream-Objekt `file01` überhaupt mit einer Datei verbunden ist. Die Methode gibt `true` zurück, wenn dies der Fall ist.

Auch geschlossen wird eine Datei, wenn das laufende Programm beendet oder der Gültigkeitsbereich der Stream-Variablen verlassen wird, z.B.:

```
...
01  std::ofstream file01("data001.db");
02  {
03      // ...
04      std::ofstream file02("data002.db");
05  } // ab hier wird file02 geschlossen
...
```

Hierzu jetzt noch eine kurze Übersicht über die grundlegenden Methoden zum Öffnen und Schließen von Dateien in Tabelle 13.15.

Methode	Beschreibung
open(dateiname)	Öffnet die Datei für ein Stream-Objekt im Standardmodus. Der Standardmodus hängt von der verwendeten Klasse ab (ifstream, ofstream oder fstream).
open(dateiname, flags)	Öffnet die Datei für ein Stream-Objekt und verwendet flags für den Modus.
close()	Schließt die Datei.
is_open()	Gibt true zurück, wenn ein Stream-Objekt mit einer Datei geöffnet ist, ansonsten false.

Tabelle 13.15 Methoden zum Öffnen und Schließen von Dateien

Neu in C++11

In C++11 wurde neben der Möglichkeit, jetzt `std::string` für den Dateinamen verwenden zu können, auch noch ein Move-Konstruktor hinzugefügt und der Kopierkonstruktor mit `deleted` implementiert, damit dieser nicht mehr verwendet werden kann.

13.6.3 Lesen und Schreiben

Nachdem Sie eine Datei geöffnet haben, können Sie, wie Sie das bereits vom Standard-Stream-Objekt wie `cin` oder `cout` her gewohnt sind, dieses Stream-Objekt zum Lesen und Schreiben verwenden.

Die einfachste Möglichkeit, etwas in eine Datei zu schreiben oder etwas daraus zu lesen, dürften wieder die Operatoren `<<` zum Schreiben und `>>` zum Lesen sein. Hier-

mit können Sie auch alle bekannten Manipulatoren und Methoden für die Formatierung verwenden (siehe Abschnitt 13.5, »Streams manipulieren und formatieren«). Folgendes Beispiel soll dies demonstrieren:

```
00 // kapitel13/fstream/fstream04.cpp
01 #include <fstream>
...
02 std::ofstream file("data.dat");
03 if( !file ) {
04     std::cerr << "Konnte data.dat nicht öffnen\n";
05     return 1;
06 }

07 int ival=0;
08 std::cout << "Bitte eine Ganzzahl eingeben: ";
09 std::cin >> ival;
10 file << std::setw(10) << std::setfill('*')
    << ival << std::endl;
11 file.close();
```

In Zeile **02** öffnen wir hier eine Datei *data.dat* zum Schreiben. Ist die Datei bereits vorhanden, wird der alte Inhalt gelöscht. Existiert diese Datei noch nicht, wird sie angelegt. In Zeile **09** werden Sie aufgefordert, einen integralen Wert einzugeben. In Zeile **10** schieben wir diesen eingegebenen Wert mit einer Feldbreite von zehn Zeichen, wobei die restlichen Zeichen mit einem Sternchen aufgefüllt werden, in das Stream-Objekt *file*, wo alles in die Datei geschrieben wird. Haben Sie bspw. den Wert »1234« eingegeben, sieht der Inhalt der Datei *data.dat* jetzt wie folgt aus:

```
*****1234
```

In der Praxis wird allerdings das formatierte Schreiben und Lesen mit den Operatoren << bzw. >> eher selten eingesetzt. Es ist wesentlich effizienter, das Lesen und Schreiben von und in Dateien in ganzen Blöcken und im Binärmodus durchzuführen. Auf die einzelnen Möglichkeiten (zeichenweise, zeilenweise und blockweise) soll jetzt in den nächsten Abschnitten etwas genauer eingegangen werden.

Byteweises lesen und schreiben

Wenn Sie eine Datei Byte für Byte lesen oder schreiben wollen/müssen, können Sie auch hier auf die Methoden `get()` und `put()` zurückgreifen (siehe Abschnitt 13.3, »Methoden für die Aus- und Eingabe von Streams«). Die Verwendung ist dieselbe wie schon mit den Stream-Objekten `cin` bzw. `cout`, nur dass Sie eben das entsprechende Stream-Objekt verwenden, womit Sie die Datei geöffnet haben.

Das folgende Listing liest eine Datei Byte für Byte aus und gibt jedes Byte auf dem Bildschirm aus:

```

00 // kapitel13/fstream/fstream05.cpp
01 #include <fstream>
...
02 std::string name;
03 std::cout << "Datei zum Lesen öffnen: ";
04 std::cin >> name;

05 std::ifstream file(name.c_str());
06 if( !file ) {
07     std::cerr << "Fehler beim Öffnen\n";
08     return 1;
09 }
10 char ch;
11 while(file.get(ch) ) {
12     std::cout.put(ch);
13 }
14 if( file.eof() ) {
15     file.clear();
16 }
17 file.close();

```

Nach der Abfrage eines Dateinamens versuchen wir, diese Datei in Zeile 05 zu öffnen. Wenn Ihr Compiler den C++11-Standard bereits kann, können Sie auf die Methode `c_str()` verzichten. In der `while`-Schleife in Zeile 11 starten wir mit dem byteweisen Lesen aus dem Datei-Stream `file`. Jedes eingelesene Zeichen mit `get()` wird daraufhin in Zeile 12 auch gleich mit `put()` auf den Bildschirm ausgegeben. Die `while`-Schleife wird so lange ausgeführt, bis das Ende der Datei (EOF, End of File) erreicht wurde.

Es sollte jetzt kein Problem sein, statt einer Ausgabe auf dem Bildschirm eine Datei Byte für Byte zu kopieren. Hierzu müssen Sie einfach statt der Ausgabe auf `std::cout` einen Datei-Stream zum Schreiben anlegen, öffnen und die gelesenen Bytes eben dort (byteweise) reinschreiben. Das Beispiel hierzu will ich Ihnen nicht vorenthalten:

```

00 // kapitel13/iostream/fstream06.cpp
01 #include <fstream>
...
02 std::ifstream file(name.c_str());
03 if( !file ) { /* Fehler */ }
04 std::ofstream filecopy("backup.txt");
05 if( !filecopy ) { /* Fehler */ }

```

```

06 char ch;
07 while(file.get(ch) ) {
08     filecopy.put(ch);
09 }
...

```

Genauso können Sie auch von der Standardeingabe über das Stream-Objekt `std::cin` Daten von der Tastatur einlesen, anstatt den Daten-Stream `file` zu verwenden und so die eingetippten Daten in den Datei-Stream `filecopy` zu schreiben. Hierbei müssen Sie allerdings EOF (End of File) selbst mit der Tastenkombination `[Strg]+[D]` bzw. `[Strg]+[Z]` auslösen.

Langsames Kopieren

Es sollte einleuchtend sein, dass byteweises Kopieren nicht sehr effizient ist.

Zeilenweises lesen und schreiben

Das zeilenweise Abarbeiten einer Datei setzt natürlich auch Dateien voraus, in denen Zeilenumbrüche vorhanden sind. Daher ist das zeilenweise Lesen bzw. Schreiben eher für formatierte Dateien geeignet. Es macht wenig Sinn, eine binäre Datei zeilenweise abzarbeiten. Für das zeilenweise Einlesen eines Streams bietet sich die Methode `getline()` an (siehe Abschnitt 13.3, »Methoden für die Aus- und Eingabe von Streams«) und für das formatierte Schreiben in den Stream der operator<<().

```

00 // kapitel13/fstream/fstream07.cpp
01 #include <fstream>
...
02 std::string name;
03 std::cout << "Datei zum Lesen öffnen: ";
04 std::cin >> name;

05 std::ifstream file(name.c_str());
06 if( !file ) { /* Fehler */ }
07 std::ofstream filecopy("backup.txt");
08 if( !filecopy ) { /* Fehler */ }
09 std::string puffer;
10 while( getline(file, puffer) ) {
11     filecopy << puffer << std::endl;
12     std::cout << puffer << std::endl;
13 }
14 if( file.eof() ) {
15     file.clear();

```

```

16 }
17 file.close();
18 filecopy.close();

```

Das Listing entspricht im Großen und Ganzen dem Beispiel zuvor, in dem byteweise eingelesen wurde. Nur finden Sie jetzt in Zeile 09 statt eines Zeichens einen String als Puffer vor, in den wir in der `while`-Schleife in Zeile 10 mit `getline()` vom Datei-Stream `file` eine Zeile einlesen. In Zeile 11 kopieren wir diese eingelesene Zeile, welche sich jetzt in `puffer` befindet, in den Datei-Stream `filecopy`. Das Newline wird bei `getline()` entfernt, und dieses müssen Sie daher hier mit in den Stream geben. Zusätzlich geben wir die Zeile auch noch in Zeile 12 auf dem `std::cout` (gewöhnlich dem Bildschirm) aus. Auch die `while`-Schleife liest jetzt so lange Zeile für Zeile ein, bis das Dateiende mit EOF erreicht wurde.

Blockweises Lesen und Schreiben

Zum Lesen und Schreiben ganzer roher Datenblöcke eignen sich die Methoden `read()` und `write()` am besten (siehe Abschnitt 13.3, »Methoden für die Aus- und Eingabe von Streams«).

Hierzu ein Listing, wie Sie eine Datei in einem Rutsch kopieren können. Hierbei wurde auch gleich ein wahlfreier Zugriff verwendet, der gleich im nächsten Abschnitt 13.6.4 erläutert werden soll.

```

00 // kapitel13/fstream/fstream08.cpp
01 #include <fstream>
...
02 std::string name;
03 std::cout << "Datei zum Lesen öffnen: ";
04 std::cin >> name;

05 std::ifstream file(name.c_str(), std::ios::binary);
06 if( !file ) {
07     std::cerr << "Fehler beim Öffnen\n";
08     return 1;
09 }
10 std::ofstream filecopy("backup.dat", std::ios::binary);
11 if( !filecopy ) {
12     std::cerr << "Fehler beim Öffnen\n";
13     return 1;
14 }

15 file.seekg(0, std::ios::end);
16 std::streampos size = file.tellg();

```

```

17 std::cout << "Dateigröße : " << size << " Byte\n";
18 file.seekg(0, std::ios::beg); // Wichtig !!!

19 char* puffer = new char[size];

20 file.read(puffer, size);
21 if( !file ) { std::cerr << "Fehler bei read...\n"; }
22 std::cout << "Gelesen   : " << file.gcount() << " Byte\n";
23 filecopy.write( puffer, size );
24 if( !filecopy ) { std::cerr << "Fehler bei write...\n"; }

25 delete[] puffer;
26 file.close();
27 filecopy.close();

```

Nachdem wir in den Zeilen 02 bis 14 eine Datei zum Lesen und eine weitere Datei zum Schreiben für das Kopieren geöffnet haben (beide im binären Modus mit `ios::binary`), setzen wir in Zeile 15 die Position der zu kopierenden Datei mit der Methode `seekg()` auf das Ende (`ios::beg`). Mithilfe der Position am Ende der Datei bekommen wir in Zeile 16 mit der Methode `tellg()` das Byte-Offset vom Dateianfang zurück und speichern hierbei die Größe der Datei, welche wir für das Kopieren benötigen, in der Variablen `size`.

Ganz wichtig ist es jetzt auch, dass Sie die Leseposition wieder auf den Anfang setzen, wie dies mit Zeile 18 wiederum mit einem wahlfreien Zugriff gemacht wurde. In Zeile 19 reservieren wir Speicher, welchen wir für die komplette Datei zum Einlesen benötigen, und lesen den Inhalt der kompletten Datei mit der Größe von `size` Byte jetzt mit `read()` in den neu reservierten Speicher (Zeile 20) ein. Der komplette Inhalt der Datei befindet sich jetzt in `puffer`. In Zeile 23 wiederum schreiben wir jetzt den kompletten Inhalt mit der Größe von `size` Byte mit der Methode `write()` von `puffer` in den Datei-Stream `filecopy`.

Speichern von Vektoren, Strukturen oder Klassen

Zwar wurde im Listing nur demonstriert, wie Sie blockweise eine einfache Datei kopieren können. In der Praxis ist die Möglichkeit mit `read()` und `write()` auch bestens dafür geeignet, komplexere Datenstrukturen wie Vektoren, Strukturen oder die Eigenschaften von Klassen zu speichern. Dabei muss natürlich immer darauf geachtet werden, dass der Typ in ein `char*` umgewandelt werden muss.

Da Sie früher oder später die Eigenschaften von Klassen werden speichern wollen und es durchaus etwas komplexer erscheint, soll hier noch ein einfaches Rezept geschrieben werden, welches demonstriert, wie Sie dies in der Praxis realisieren können. Als Grundgerüst soll folgende Klasse verwendet werden:


```

00 // kapitel13/fstream/fstream09.cpp
01 #include <fstream>
...
02 class DataClass {
03     std::string text;
04     int data;
05 public:
06     DataClass(std::string t="",int d=0.0) : text(t),data(d) {}
07     std::ostream& write(std::ostream& os);
08     std::istream& read(std::istream& is);
09     void print_data();
10 };
...

```

Das Hauptaugenmerk liegt hierbei auf der Implementierung der beiden Methoden `write()` und `read()`, welche wie folgt definiert wurden:

```

00 // kapitel13/fstream/fstream09.cpp
...
01 std::ostream& DataClass::write(std::ostream& os) {
02     os << text << std::ends;
03     os.write(reinterpret_cast<char*>(&data), sizeof(data));
04     return os;
05 }

06 std::istream& DataClass::read(std::istream& is) {
07     std::getline(is, text, '\0');
08     is.read(reinterpret_cast<char*>(&data), sizeof(data));
09     return is;
10 }

11 void DataClass::print_data() {
12     std::cout << text << " : " << data << std::endl;
13 }
...

```

Anhand des Methodenkopfes in Zeile **01** können Sie sehr schön erkennen, dass Sie hier die Eigenschaften von `DataClass` in den Stream `os` schreiben und diesen Typ auch wieder zurückgeben. Bei Zeile **02** mit der Eigenschaft `std::string` hängen wir den Manipulator `std::ends` (`='\0'`) als Endemarkierung für diese Eigenschaft dran, welche wir zum Auslesen der Daten wieder benötigen. Bedenken Sie hierbei, dass `std::string` wiederum ein Teilobjekt der Klasse `DataClass` ist, deren Länge immer unterschiedlich sein kann. In Zeile **03** müssen wir den Typ in ein `char*` umwandeln,

weil `write()` dies als erstes Argument erwartet. Am Ende geben wir in Zeile 04 den übergebenen Stream wieder zurück.

Ähnlich läuft dies auch beim Gegenstück von `write()` in den Zeilen 06 bis 10 ab. Nur eben jetzt in die andere Richtung. Die Methode erhält den einen Eingabe-Stream als Parameter und gibt diesen auch wieder zurück. Zum Einlesen der ersten Eigenschaft `text` verwenden wir die `std::getline()`-Funktion (Zeile 07). Hier lesen wir bis zum Zeichen `'\0'` ein, welches wir bei `write()` zuvor extra dafür angehängt haben. Auch beim Einlesen müssen Sie bei `read()` das erste Argument nach `char*` umwandeln, weil dies von `read()` erwartet wird. Zurückgegeben wird am Ende in Zeile 09 der Eingabe-Stream `is`.

In der Praxis können Sie jetzt die Methoden `DataClass::read()` bzw. `DataClass::write()` wie folgt verwenden:

```
00 // kapitel13/fstream/fstream09.cpp
...
01 std::ofstream file_w("data.dat", std::ios::binary);
02 if( !file_w ) { std::cerr << "Fehler bei Öffnen\n"; }

03 std::vector<DataClass> vec_dat;
04 vec_dat.push_back(DataClass(std::string("Ein Text"), 123));
05 vec_dat.push_back(DataClass(std::string("Mehr Text"), 321));
06 vec_dat.emplace_back( std::string("Viel mehr Text"), 333 );
07 for(size_t i=0; i<vec_dat.size(); ++i){
08     vec_dat.at(i).write(file_w);
09 }
10 file_w.close();

11 std::ifstream file_r("data.dat", std::ios::binary);
12 if( !file_r ) { std::cerr << "Fehler bei Öffnen\n"; }
13 DataClass dat_r;
14 while( file_r ) {
15     dat_r.read(file_r);
16     if( file_r.eof()) break;
17     dat_r.print_data();
18 }
...
```

Im Beispiel verwenden wir gleich `std::vector`, um mehrere Objekte vom Typ `DataClass` auf einmal speichern zu können. In den Zeilen 03 bis 06 packen wir ein paar `DataClass`-Objekte in den Vektor `vec_dat`. Die Version in Zeile 06 funktioniert nur mit neueren C++11-Compilern. In den Zeilen 07 bis 09 schreiben wir Element für Ele-

ment in den Datei-Stream `file_w`, welchen wir in Zeile 01 zum Schreiben geöffnet und angelegt haben. Hierfür kommt unsere Methode `DataClass::write()` in Zeile 08 zum Einsatz.

Ähnlich funktioniert auch das Einlesen der geschriebenen Daten, das wir hier in den Zeilen 11 bis 18 demonstrieren. In Zeile 11 öffnen wir zunächst die Datei zum Lesen. In der `while`-Schleife in den Zeilen 14 bis 18 lesen wir jetzt einen Datensatz nach dem anderen ein (Zeile 15) und geben diesen in Zeile 17 aus. Hier könnten Sie natürlich auch wieder alles in einen Vektor legen, aber hier sollte nur das Auslesen allein demonstriert werden. Dazwischen prüfen wir immer wieder auf `eof()` und brechen gegebenenfalls die Schleife ab, wenn `eof()` erreicht wurde. Die Schleife wird ebenfalls so lange durchlaufen, solange `file_r` gleich `true` ist und somit `failbit` nicht gesetzt ist.

13.6.4 Wahlfreier Zugriff

Beim blockweisen Lesen im Abschnitt zuvor (in Listing *fstream08.cpp*) haben Sie bereits den wahlfreien Zugriff auf Datei-Streams kennengelernt. Damit ist es möglich, die aktuelle Schreib- bzw. Leseposition zu ermitteln oder die Position zu verändern. In Tabelle 13.16 finden Sie eine Übersicht der vorhandenen Methoden.

Methode	Beschreibung
<code>istream& seekg(streampos);</code> <code>istream& seekg(streampos, ios::seekdir);</code>	Setzt die Leseposition auf eine bestimmte Position vom Dateianfang oder eine relativ zum optional zweiten Argument angegebene Bezugsposition.
<code>streampos tellg();</code>	Gibt die aktuelle Leseposition zurück.
<code>ostream& seekp(streampos);</code> <code>ostream& seekp(streampos, ios::seekdir);</code>	Setzt die Schreibposition auf eine bestimmte Position vom Dateianfang oder eine relativ zum optional zweiten Argument angegebene Bezugsposition.
<code>streampos tellp();</code>	Gibt die aktuelle Schreibposition zurück.

Tabelle 13.16 Methoden für einen wahlfreien Zugriff

Die möglichen Werte für das zweite Argument von `seekg()` bzw. `seekp()` (g steht für get und p für put) für die Bezugsposition vom Typ `ios::seekdir` lauten:

- ▶ `ios::beg`: Position relativ zum Anfang der Datei
- ▶ `ios::cur`: Position relativ zur aktuellen Position der Datei
- ▶ `ios::end`: Position relativ zum Dateieinde

streampos vs. long

In der Vergangenheit wurde die Position in einer Datei einfach als `unsigned long` definiert, weshalb hierfür häufig auch einfach Folgendes verwendet wurde, um die aktuelle Position zu sichern:

```
// Position in pos sichern
unsigned long pos = file.tellg(); // Veraltet !!!
...
// Position wiederherstellen
file.seekg(pos);
```

Das ist allerdings nicht mehr richtig. Ohne hier auf die Details des Klassen-Templates `fpos<>` einzugehen, sollten Sie statt `unsigned long` jetzt entweder den Typ `std::streampos` oder `std::ios::pos_type` für die Dateiposition verwenden.

Hierzu einige Beispiele:

```
// 10 Zeichen weiter von der aktuellen Position positionieren
file.seekg(10, std::ios::cur);
...
// Position auf den Anfang der Datei
file.seekg(0, std::ios::beg);
...
// 5 Zeichen vor dem Ende positionieren
file.seek(-5, std::ios::end);
```

Dass Sie die Dateiposition auf einen gültigen Bereich setzen, müssen Sie hier selber sicherstellen. Wenn Sie die Position vor den Anfang oder nach das Ende setzen, ist das weitere Verhalten undefiniert.

13.7 Streams für Strings

Wenn Sie einen Blick auf die vereinfachte Klassenhierarchie der Stream-Klasse in Abbildung 13.1 werfen, finden Sie hier, ausgehend von `ios`, mit `istream`, `ostream` und `stringstream` drei Streams für Strings. Für alle drei Klassen müssen Sie den Header `<sstream>` inkludieren. Diese Streams sind bestens dafür geeignet, unterschiedliche Datentypen von bzw. in Strings umzuwandeln – auch hier wieder mit allen Vorteilen der vorausgehenden Streams und deren Methoden und Manipulatoren. In Tabelle 13.17 finden Sie die verschiedenen String-Streams aufgelistet.

Stream	Bedeutung
istringstream	Erzeugt einen String-Stream, aus dem gelesen werden kann. Der Stream wird im Modus <code>ios::in</code> geöffnet.
ostringstream	Erzeugt einen String-Stream zum Schreiben. Standardmäßig wird dieser Stream mit dem Modus <code>ios::out</code> verwendet.
stringstream	Erzeugt einen String-Stream zum Lesen und Schreiben. Der Standardmodus ist hier <code>ios::in</code> und <code>ios::out</code> .

Tabelle 13.17 String-Streams der Standardbibliothek

Bei allen drei Konstruktoren können Sie neben einem leerem Konstruktor als Argument entweder den Öffnungsmodus wie bei den Datei-Streams (siehe Tabelle 13.13), einen String (`std::string`), womit der Stream gleich initialisiert werden soll, oder eine Mischung aus dem String (erstes Argument) und dem Öffnungsmodus (zweites Argument) verwenden. Zu den Öffnungsmodi muss noch hinzugefügt werden, dass bei den String-Streams vorwiegend `ios::in`, `ios::out` und `ios::ate` richtig Sinn machen. Die anderen Werte des Typs `ios::openmode` sind zwar ebenfalls implementiert, aber ob diese tatsächlich einen Effekt auf ein `stringstream`-Objekt haben, hängt von der Implementierung ab.

Altes `strstream`

Im alten C++98-Standard wurde eine Stream-Klasse noch mit dem Typ `char*` realisiert. Heute wird hierfür der Typ `string` (genauer `basic_string`) verwendet. Die alten String-Stream-Klassen `istrstream`, `ostrstream` und `strstream` sind daher nur aus abwärts kompatiblen Gründen vorhanden und sollten in neuen Projekten nicht mehr verwendet werden (da *deprecated*). Die Klassen sollten hier trotzdem kurz erwähnt werden, weil diese in älteren Dokumentationen und Quellcodes immer mal wieder auftauchen können.

Da die String-Streams sämtliche `public`-Methoden von den davon abgeleiteten Basis-klassen erben, stehen Ihnen hier auch alle bisher bekannten und vorgestellten Methoden mit den String-Streams zur Verfügung. Die Hauptmethoden für String-Streams bietet aber die Klasse selbst mit der Methode `str()` an (siehe Tabelle 13.18).

Methode	Beschreibung
<code>string str() const;</code>	Gibt den String-Stream-Puffer als String zurück.

Tabelle 13.18 Die Hauptfunktion »`str()`« ist häufig Dreh- und Angelpunkt bei der Verwendung von String-Streams.

Methode	Beschreibung
<code>void str(const string& s);</code>	Der String wird komplett neu in den String-Stream-Puffer gesteckt.

Tabelle 13.18 Die Hauptfunktion »str()« ist häufig Dreh- und Angelpunkt bei der Verwendung von String-Streams. (Forts.)

Das folgende einfache Beispiel soll kurz demonstrieren, wie Sie einfache Datentypen wie `double` oder `int` in einen String konvertieren können:

```
00 // kapitel13/sstream/ssstream01.cpp
01 #include <sstream>
...
02 std::ostringstream ostr;
03 std::string sval;
04 double dval = 3.1415;
05 int ival = 4321;

06 ostr << dval << " : " << ival;
07 sval = ostr.str();
08 std::cout << sval << std::endl; //3.1415 : 4321
...
```

In Zeile 02 legen wir den String-Stream zum Schreiben an, welchen wir für die Konvertierung benötigen. In Zeile 06 schieben wir ganz einfach den `double`-Wert und das Integer getrennt mit einem Doppelpunkt in den String-Stream. Mithilfe der Methode `str()` in Zeile 07 kopieren wir den kompletten Inhalt des String-Stream-Puffers in den String `sval`, was die Ausgabe in Zeile 08 auch bestätigt.

Dasselbe funktioniert natürlich auch in die andere Richtung. Der folgende Codeausschnitt soll zeigen, wie Sie aus einem String einzelne Typen extrahieren können:

```
00 // kapitel13/sstream/ssstream02.cpp
01 #include <sstream>
...
02 std::istringstream istr;
03 std::string sval("3.1415 : 4321");
04 std::string none;
05 double dval=0.0;
06 int ival=0;

07 istr.str(sval);
08 istr >> dval >> none >> ival;
```

```

09  if( ! istr.eof() ) {
10      std::cerr << "Fehler beim Konvertieren\n";
11  }
12  std::cout << dval << std::endl << ival << std::endl;
...

```

In Zeile 02 finden Sie den String-Stream zum Lesen, welchen wir verwenden, um die beiden Werte aus dem String `sval` in Zeile 03 zu extrahieren und in die Variablen `dval` und `ival` (Zeilen 05 und 06) zu sichern. In Zeile 07 stecken wir den String `sval` in den String-Stream-Puffer `istr`. In Zeile 08 holen wir uns die einzelnen Daten aus dem String-Stream-Puffer, und die ASCII-Darstellung wird in eine binäre Darstellung konvertiert. Für den Doppelpunkt verwenden wir einfach einen Platzhalter mit dem String `none`. In Zeile 09 prüfen wir noch, ob bei der Konvertierung irgendwelche Fehler aufgetreten sind. Definitiv ist ein Fehler aufgetreten, wenn das String-Stream-Objekt nicht gleich `eof()` ist. Das `eofbit` wird gesetzt, wenn der String-Stream ordentlich zu Ende gelesen wurde. Wenn alles geklappt hat, sind bei der Ausgabe in Zeile 12 die Werte entsprechend in `double dval` und `int ival` enthalten.

Die Konvertierung von der Binär-Darstellung zur ASCII-Darstellung und umgekehrt lässt sich recht einfach mit den String-Streams realisieren. Da allerdings die Arbeit der Konvertierung fast immer recht ähnlich verläuft, bietet es sich an, hierfür ein Funktions-Template zu erstellen. Das folgende Funktions-Template dient als einfaches Rezept, mit dem Sie jederzeit einen Basisdatentyp von C++ in einen String oder einen String in einen Basisdatentyp umwandeln können:

```

00  // kapitel13/sstream/ssstream03.cpp
01  #include <sstream>
...
02  class ConvErr{
03      std::string err;
04  public:
05      ConvErr(const std::string& e="?") : err(e) {}
06      void what() const {
07          std::cout << err << std::endl;
08      }
09  };

10  template <class T1, class T2>
11  void myConvert(const T1& in, T2& out) {
12      std::stringstream ss;
13      ss << in;
14      ss >> out;
15      if( ! ss.eof() ) {

```

```

16     throw ConvErr("Fehler beim Konvertieren");
17 }
18 }
...
19 std::string sval;
20 float fval=3.1415f;
21 std::string sdval("5.321");
22 double dval=0;
23 std::string gehtnicht("geht nicht");
24 try {
25     myConvert(fval, sval);
26     std::cout << sval << std::endl;
27     myConvert(sdval, dval);
28     std::cout << dval << std::endl;
29     myConvert(gehtnicht, dval); // Fehler !!!
30 }
31 catch(const ConvErr& e) {
32     e.what();
33 }
...

```

Der Hauptfokus liegt in diesem Beispiel beim Funktions-Template in den Zeilen 10 bis 18, wo der Typ `T1` in den Typ `T2` konvertiert werden soll. Hierzu schicken wir die Daten einfach in Zeile 13 in den String-Stream hinein und holen diese dann eine Zeile später konvertiert wieder heraus. Ob alles geklappt hat, prüfen wir in den Zeilen 15 bis 17 und werfen im Falle eines Fehlers eine Ausnahmeklasse `ConvErr`, welche wir in den Zeilen 02 bis 09 definiert haben. Im `try`-Block wird in Zeile 25 ein `float`-Wert in einen String konvertiert und in Zeile 27 ein String-Wert in einen `double`-Wert. Die Konvertierung in Zeile 29 wirft eine Ausnahme, weil eine Konvertierung einer Textfolge »geht nicht« in ein `double` nicht funktionieren wird.

String-Streams statt Datei-Streams

String-Streams lassen sich zwar prima für die Konvertierung von und in Strings verwenden, aber in der Praxis werden diese noch viel häufiger zu Pufferungszwecken verwendet, um hierbei auf temporäre Dateien zu verzichten. Im Grunde müssen Sie hierzu einfach statt eines Datei-Streams einen entsprechenden String-Stream verwenden. Der Vorteil liegt auf der Hand: String-Streams sind erheblich schneller (da im Hauptspeicher) als langsame temporäre Dateien, bei denen ein eher langsamer Zugriff auf ein Speichermedium nötig ist. Dies hängt allerdings auch vom Anwendungsfall ab. So dürfte auch klar sein, dass Daten, welche lediglich im Puffer des Arbeitsspeichers liegen, im Falle eines Programmabsturzes verloren sind.

13.8 Stream-Puffer

Auch wenn Sie eher selten direkt darauf zugreifen müssen, sollen zum Abschluss der Stream-Klassen hier noch kurz die Stream-Puffer-Klassen beschrieben werden. Wenn Sie einen Blick auf die Klassenhierarchie in Abbildung 13.1 werfen, finden Sie diese dort mit `streambuf`, `filebuf` und `stringbuf` etwas an den Rand gedrängt. In Wirklichkeit sind diese Stream-Puffer-Klassen aber eher so etwas wie das Kernstück der I/O-Stream-Bibliothek. Wenn Sie sich schon immer gefragt haben, wo die einzelnen Bytes der Stream-Klassen bearbeitet bzw. verarbeitet werden, dann finden Sie die Antwort in den Stream-Puffer-Klassen.

Die Stream-Puffer-Klassen bieten grundlegende Methoden zum Puffern von Streams ohne große Formatierungsmöglichkeiten. Jeder Stream (String-Stream, Datei-Stream und Standard-I/O-Stream) arbeitet mit Stream-Puffern. Die abstrakte Basisklasse `streambuf` wird von allen anderen Teilen der IO-Stream-Klassen verwendet. Für die Pufferung der Ein- und Ausgabe von Dateien kommt `filebuf` und für die Pufferung von Strings im Speicher kommt `stringbuf` zum Einsatz, welche beide von `streambuf` abgeleitet sind.

Bei Pufferung werden nicht einzelne Zeichen in den Stream gesendet, sondern die Daten werden hierbei in Blöcken übertragen. Die Ausgabe erfolgt bspw. erst, wenn eine Blockeinheit des Puffers voll ist. Hierbei können Sie allerdings auch manuell mit der Methode `flush()` eingreifen, welche explizit das Entleeren des Puffers durchführt, bspw.:

```
01 std::ofstream file("logfile.log");
02 file << "Text für das Protokoll";
03 file.flush();
```

Sollten bspw. die Daten in Zeile **02** noch nicht in den Datei-Stream `file` übertragen worden sein, weil die Blockeinheit des Puffers noch nicht so weit war, dann erledigt dies in Zeile **03** ein einfacher Aufruf der Methode `flush()`.

Anwendungsgebiet

Für den normalen Hausgebrauch werden Sie eher selten direkt auf die Stream-Puffer-Klassen zugreifen. Diese spielen eher dann eine wichtigere Rolle, wenn Sie neue Klassen von Streams für die Ein- und Ausgabe erstellen wollen (bspw. Sockets oder GUIs). Auch wenn Sie Streams umleiten wollen, ist der direkte Zugriff auf den Stream-Puffer sehr nützlich. Ebenso ist es mithilfe von Stream-Puffer-Klassen möglich, Streams zu kombinieren (bspw. die Ausgabe zu verschlüsseln, bevor diese in einen anderen Stream geschrieben wird).

Zugriff auf den Stream-Puffer von iostream-Objekten

Für Sie ist am wichtigsten zu wissen, dass für jedes `iostream`-Objekt ein Zeiger auf ein `streambuf`-Objekt zur Verfügung steht. Über diesen Zeiger ist es jederzeit möglich, direkt auf die rohen Bytes im Stream-Puffer zuzugreifen. Hierfür stehen verschiedene Methoden zur Verfügung, welche mit einem `streambuf`-Objekt aufgerufen werden können.

Um auf einen Stream-Puffer zuzugreifen, bietet jedes I/O-Stream-Objekt eine Methode `rdbuf()` an, welche einen Zeiger auf das `streambuf`-Objekt zurückgibt. Über diesen Weg können Sie jetzt auf die Methoden von `streambuf` zugreifen. Hierbei können Sie bspw. den `streambuf`-Zeiger mit einem anderen Stream verbinden, indem Sie `operator<<()` dazu verwenden. Damit werden auf einmal alle Zeichen vom Stream-Puffer-Objekt auf das Stream-Objekt ausgegeben, welches links von `<<` steht. Hierzu ein einfaches Beispiel:

```
01 #include <fstream>
...
02 std::ifstream file("logfile.log");
03 std::cout << file.rdbuf();
```

Mit Zeile 03 und `file.rdbuf()` wird der komplette Inhalt der Datei auf einmal nach `std::cout` geschoben und ausgegeben. Diese Methode, einen kompletten Inhalt in einem Rutsch an einen anderen Stream zu übergeben, ist sogar erheblich effizienter, als den kompletten Inhalt mit `read()` und `write()` auszugeben, wie Sie dies im Abschnitt »Blockweises Lesen und Schreiben« in Abschnitt 13.6.3 verwendet haben.

Im folgenden Listing will ich Ihnen zeigen, wie Sie den Inhalt einer Datei aus dem Puffer in ein rohes `char`-Array legen und dort gegebenenfalls weiterverarbeiten können. Im Beispiel geben wir den Inhalt einfach mit `write()` auf dem Bildschirm aus:

```
00 // kapitel13/streambuf/streambuf01.cpp
...
01 std::fstream file("data.txt");
02 char *buffer=nullptr;
...
03 std::streambuf* bufptr = file.rdbuf();
04 std::streamsize size = bufptr->pubseekoff(0, file.end);
05 bufptr->pubseekoff(0, file.beg);
06 buffer = new char[size];
07 bufptr->sgetn(buffer, size);
08 std::cout.write(buffer, size);
09 file.close();
```

Dank Zeile 03 haben Sie jetzt mit dem `streambuf`-Zeiger die volle Kontrolle über den `streambuf`-Puffer. In Zeile 04 setzen wir daher zunächst den Stream-Puffer-Zeiger auf

das Ende des Puffers, um so die Größe des Puffers zu erhalten. Da wir anschließend auch noch aus dem Puffer lesen wollen, setzen wir die Position des Stream-Puffer-Zeigers in Zeile 05 wieder auf den Anfang des Puffers. In Zeile 06 reservieren wir jetzt einen Speicher vom Heap, in dem wir den kompletten Inhalt des Puffers unterbringen können. Mit Zeile 07 und der Methode `sgetn()` kopieren wir jetzt `size`-Bytes vom Stream-Puffer in das `char`-Array. Das war es auch schon. Hiermit hätten wir den kompletten Stream-Puffer in ein rohes `char`-Array übertragen. Zur Demonstration geben wir in Zeile 08 noch den Inhalt des rohen `char`-Arrays auf dem Bildschirm aus.

Sie wissen jetzt, wie Sie mit der Methode `rdbuf()` an den Stream-Puffer (jeder Art) kommen. Mithilfe des zurückgegebenen Zeigers können Sie jetzt mit den verschiedensten Methoden innerhalb des Streams die Position ändern, Zeichen oder ganz Blöcke lesen und natürlich auch Zeichen in den Stream-Puffer schreiben oder zurückstellen. Auf die Methoden von `streambuf` soll hier jetzt nicht mehr näher eingegangen werden. Hierzu empfehle ich Ihnen, die Dokumentation Ihres Compilers oder Online-Dokumentationen, wie bspw. <http://en.cppreference.com>, zu verwenden.

filebuf

Die von `streambuf` abgeleitete Klasse `filebuf` enthält zusätzlich noch Methoden, um eine Datei direkt zu öffnen und einen Stream-Puffer anzulegen. Folgender Codeausschnitt soll dies kurz demonstrieren:

```
00 // kapitel13/filebuf/filebuf01.cpp
...
01 std::ofstream file;
02 std::filebuf* bufptr = file.rdbuf();

03 bufptr->open("file.txt", std::ios::out|std::ios::app);

04 const char text[]="Das kommt in die Datei";
05 bufptr->sputn(text, sizeof(text)-1);
06 bufptr->close();
```

Bei diesem Beispiel öffnen oder erzeugen wir eine Datei Namens *file.txt* und verbinden den Inhalt der Datei gleich mit dem Datei-Puffer, um direkte Arbeiten darauf auszuführen. Im Beispiel schreiben wir mit der `streambuf`-Methode `sputn()` den Text `text` in den Puffer und gleichzeitig auch in die Datei.

stringbuf

Auch `stringbuf` enthält alle Methoden aus der Basisklasse `streambuf`. Zusätzlich ist hier auch noch, passend zu den Strings eben, die Methode `str()` wie schon bei der Klasse `stringstream` mit derselben Bedeutung vorhanden, nur hier eben in Verbindung mit einem String-Stream-Puffer.

Kapitel 14

Reguläre Ausdrücke (C++11)

C++ ist durchaus geeignet, um Text zu verarbeiten. Dank der Klasse `std::string` und fertigen Algorithmen ist es auch keine große Sache, nach Strings zu suchen und sie weiterzuverarbeiten. Wenn Sie allerdings komplexere Sachen wünschen, wie bspw. innerhalb einer Datei im HTML-Format spezielle Tag-Paare zu finden und alle durch ein anderes Tag-Paar zu ersetzen, dann kann der Aufwand, einen solchen Code zu schreiben, schon zu groß werden. Für komplexere Suchmuster wurden mit C++11 die regulären Ausdrücke in den Standard eingeführt.

Die regulären Ausdrücke (engl. *regular expressions*) sind eine formale Sprache, mit der Sie eine bestimmte Menge an Zeichenketten beschreiben können. Bei den regulären Ausdrücken handelt es sich um eine echte eigene Sprache mit einer formalen Grammatik, in der jeder Ausdruck eine feste Bedeutung hat. Viele Texteditoren und wahrscheinlich auch Ihre Entwicklungsumgebung bieten Funktionen an, um nach einem bestimmten Muster in einem Text zu suchen und diesen gegebenenfalls durch einen anderen Text zu ersetzen. Solche Funktionen werden meistens mithilfe von regulären Ausdrücken realisiert.

Als Programmierer kann es Ihnen im Prinzip egal sein, wie die regulären Ausdrücke funktionieren. Sie müssen im Grunde nur um zwei Dinge wissen:

- die Grammatik für die regulären Ausdrücke, welche die `regex`-Bibliothek von C++11 verwendet
- die Algorithmen, mit denen Sie die regulären Ausdrücke verwenden können, und wie Sie diese aufrufen

Verfügbarkeit von regulären Ausdrücken

Zur Drucklegung des Buches wurden die regulären Ausdrücke mit GCC (hier Version 4.8.1) noch nicht 100 %ig unterstützt und nur teilweise implementiert. Erfreulich hingegen ist, dass Microsoft die regulären Ausdrücke schon seit der Version 2010 von Visual C++ und Clang sie bereits seit der Version 3.0 implementiert hat.

Boost-Bibliothek regex

Wenn Ihr Compiler noch keine regulären Ausdrücke kennt oder noch nicht unterstützt, können Sie auch auf die regulären Ausdrücke der Boost-Bibliothek zurückgreifen. Im Grunde handelt es sich bei der Implementierung der regulären Ausdrücke im neuen C++11-Standard lediglich um die Aufnahme dieser Bibliothek in den Standard. Wenn Sie *Boost.Regex* statt der neuen C++11-regex-Bibliothek verwenden müssen, können Sie die Listings im Buch fast 100 %ig übernehmen. Sie müssen lediglich die Headerdatei auf `<boost/regex.hpp>` ändern und den Namensraum `boost` verwenden. Zusätzlich müssen Sie Ihrem Compiler noch den Pfad zu dieser Bibliothek mitgeben, damit dieser beim Übersetzen weiß, wo er diese Bibliothek findet.

14.1 Grammatik-Grundlagen zu den regulären Ausdrücken

Das Thema der regulären Ausdrücke ist mächtig, und nicht umsonst gibt es hierzu eigene Bücher. Daher finden Sie zu diesem Thema logischerweise nur eine sehr einfache Einführung, damit Sie selbst einfache und gängige Ausdrücke formulieren oder verstehen können.

ECMAScript-Grammatik

Auch bei den regulären Ausdrücken gibt es verschiedene Grammatiken. Die regulären Ausdrücke von C++11 verwenden die ECMAScript-Grammatik.

Eine häufige Arbeit von regulären Ausdrücken ist es, spezielle Zeichenketten in einer Menge von Zeichenketten zu suchen. Die einfachste Möglichkeit ist es natürlich, die entsprechenden Zeichen direkt (wörtlich) als Zeichenliteral (bspw. "text") zu notieren. Spezielle Zeichen können auch mithilfe von Oktal- oder Hexadezimalcode (`\ooo` oder `\xhh`) oder hexadezimalen Unicode (`\uhhhh`) angegeben werden.

Metazeichen

Neben den gewöhnlichen Zeichenketten werden auch sogenannte Metazeichen mit den Zeichen des Alphabets kombiniert, um einen regulären Ausdruck zu bilden. Jedes dieser Metazeichen `[ ] ( ) { } | ? + - * ^ $ \ .` hat dabei eine eigene besondere Bedeutung. Sollen diese Metazeichen selbst als Zeichen interpretiert werden, können diese durch das Voranstellen eines Backslashes (`\`) außer Kraft gesetzt werden. In Tabelle 14.1 finden Sie einen kurzen Überblick über die Metazeichen und deren Bedeutung.

Metazeichen	Bedeutung	Beispiel
[]	[abcdef]	Zeichenauswahl
()		einen Teilausdruck definieren
{ }	{5}	Wiederholungsangabe
	abc xyz	alternative definieren
?	a?	Wiederholung (null- oder einmal)
+	a+	Wiederholung (mindestens einmal)
-	[a-z]	Zeichenbereich definieren
*	a*	Wiederholung (beliebig oft)
\$	ende\$	Zeilenende
^	^anfang	Zeilenanfang
\		Metazeichen schützen Zeichenklassen einleiten Rückwärtsreferenz einleiten
. (Punkt)		beliebiges Zeichen (außer Zeilenende-Zeichen)

Tabelle 14.1 Metazeichen und deren Bedeutung bei regulären Ausdrücken

Zeichenauswahl

Wie in Tabelle 14.1 zu sehen ist, definieren Sie eine Zeichenauswahl zwischen eckigen Klammern. Alles, was zwischen diesen eckigen Klammern steht, gilt dann exakt für ein Zeichen dieser Auswahl. So geben Sie bspw. mit `[abc]` eines der Zeichen »a«, »b« oder »c« an. Es können auch ganze Bereiche verwendet werden. Beispielsweise legen Sie mit `[a-g]` den Bereich für Kleinbuchstaben von »a« bis »g« fest. Dasselbe geht natürlich mit Ziffern wie `[1-5]`, womit die Ziffern 1 bis 5 festgelegt werden. Hierbei lassen sich auch mehrere Zeichenauswahlen zusammenfassen. Mit `[a-ev-z0-5]` geben Sie bspw. die Kleinbuchstaben von »a« bis »e«, »v« bis »z« und die Ziffern 0 bis 5 an. Setzen Sie innerhalb der Zeichenauswahl das Zeichen `^` hin, wird der Bereich negiert. Mit `[^A-Z]` schließen Sie bspw. alle Großbuchstaben von »A« bis »Z« aus.

Vordefinierte Zeichenklassen

Für die eben gezeigten Zeichenauswahlen gibt es mit den vordefinierten Zeichenklassen vereinfachte Kurzschreibweisen. So kann bspw. statt `[0-9]` ganz einfach `\d` geschrieben werden. In Tabelle 14.2 finden Sie einige vordefinierte Zeichenklassen.

Zeichenklasse	Zweck	Entspricht
<code>\d</code>	Ziffern von 0 bis 9	<code>[0-9]</code>
<code>\D</code>	keine Ziffern von 0 bis 9	<code>[^0-9]</code>
<code>\s</code>	Whitespace-Zeichen	<code>[\f\n\r\t\v]</code>
<code>\S</code>	keine Whitespace-Zeichen	<code>[^\f\n\r\t\v]</code>
<code>\w</code>	Buchstaben, Ziffern und Unterstrich	<code>[a-zA-Z0-9_]</code>
<code>\W</code>	keine Buchstaben, Ziffern und Unterstrich	<code>[^a-zA-Z0-9_]</code>

Tabelle 14.2 Vordefinierte Zeichenklassen

Weitere vordefinierte Zeichenklassen

Neben den vordefinierten Zeichenklassen in Tabelle 14.2 gibt es noch einige, die etwas spezifischer sind. So können Sie beispielsweise mit den vordefinierten Zeichenklassen `[:upper:]` und `[:lower:]` zwischen Groß- und Kleinschreibung unterscheiden.

Wiederholungen (Quantoren)

Mithilfe von Wiederholungen (auch *Quantoren* genannt) können Sie einen vorherigen Ausdruck mehrmals in einer Zeichenkette zulassen. In Tabelle 14.3 finden Sie einen kurzen Überblick über die einzelnen Zeichen und deren Bedeutung.

Quantifizierer	Bedeutung
<code>?</code>	Der vorangehende Ausdruck ist optional. Er kann einmal vorkommen, muss aber nicht. Der Ausdruck kommt entweder null oder einmal vor.
<code>+</code>	Der Ausdruck muss mindestens einmal vorkommen, darf aber auch mehrmals vorhanden sein.
<code>*</code>	Der Ausdruck kann beliebig oft vorkommen.
<code>{n}</code>	Der vorangehende Ausdruck muss genau n-mal vorkommen.
<code>{min, max}</code>	Der Ausdruck muss mindestens min-mal, darf aber nur maximal max-mal vorkommen.
<code>{min, }</code>	Der Ausdruck muss mindestens min-mal vorkommen.
<code>{0,max}</code>	Der Ausdruck darf höchstens max-mal vorkommen.

Tabelle 14.3 Quantoren für reguläre Ausdrücke

Ein solcher Quantifizierer bezieht sich immer auf den vorangehenden regulären Ausdruck. So wird mit `z+` sowohl ein `»z«` als auch `»zzzzzz«` abgedeckt. Ein `[ab]+` deckt ein `»a«`, ein `»b«`, ein `»bb«`, ein `»ababaa«` usw. ab.

Mit `[0-9]{4,6}` muss der Ausdruck aus mindestens 4 bis maximal 6 Ziffern bestehen. Das würde einen Ausdruck mit `»12345«` oder `»123456«` abdecken. Auch ein Ausdruck wie `»123456789«` würde passen, weil der Teilstring `»123456«` die Bedingung erfüllt. Falsch hingegen wären Ausdrücke wie `»a12345«`, `»1234.56«` usw.

Gruppierung

Ausdrücke können auch zwischen runden Klammern gruppiert werden. Damit werden die Gruppierungen abgespeichert und können im regulären Ausdruck bzw. der Textersetzung erneut mit `\1` verwendet werden. Hiermit können Sie nun weitere Muster abspeichern (`\1`, `\2`, ..., `\9`).

Zum Beispiel:

```
(text1) (text2) (text3) \3 \2 \1
```

Hiermit bewirken Sie, dass jedes Vorkommen von `"text1 text2 text3"` umgeändert wird in `"text3 text2 text1"`. `\1` bezieht sich somit auf das erste, `\2` auf das zweite und `\3` hier auf das dritte Klammerpaar.

Alternativen

Auch Alternativen lassen sich mit dem `|`-Zeichen recht einfach realisieren, bspw.:

```
(sie|Sie)
```

Hiermit wird nach der Zeichenfolge `»sie«` oder der Zeichenfolge `»Sie«` gesucht, nicht aber nach einer Zeichenfolge wie `»SIE«` oder `»sIe«`.

14.2 Ein Objekt für reguläre Ausdrücke erzeugen

Nach der sehr bescheidenen Einführung zu den regulären Ausdrücken, können wir uns nun dem eigentlichem Grund dieses Kapitels widmen, der Verwendung von regulären Ausdrücken in C++. Um überhaupt reguläre Ausdrücke verwenden zu können, müssen Sie den Header `<regex>` inkludieren. Auch hier sind alle Dinge wieder im Namensraum `std` enthalten.

Als Typ für Objekte von regulären Ausdrücken dient hier das Klassen-Template `std::basic_regex`, welches wieder für alle Charaktere von Zeichen spezialisiert wurde. Hier gibt es `std::regex` für `char*` und `std::string` bzw. `std::wregex` für `wchar*` und `std::wstring`. Das Prinzip kennen Sie ja bereits von `std::basic_string` mit `std::string` und `std::wstring`.

Referenz

In diesem Abschnitt finden Sie eine Einführung, wie Sie die regulären Ausdrücke in der Praxis verwenden können. Für eine genauere Beschreibung der Syntax und der Methoden sollten Sie auch hier wieder die Dokumentation Ihres Compilers zu Rate ziehen oder bei Online-Dokumentationen, wie bspw. bei <http://en.cppreference.com> nachschlagen.

Hierzu nun einige Beispiele, wie Sie Objekte für reguläre Ausdrücke erstellen können:

```
01 #include <regex>
...
01 const char* cexpr = "[a-z]+";
02 std::string sexpr("[a-zA-Z]+");

03 std::regex expr1(cexpr);
04 std::regex expr2(sexpr);

05 std::regex expr3("[0-9]+");
06 std::regex expr4("\\d{8}");
07 std::regex expr5(R"(\d{8})");
```

In den Zeilen **03** und **04** erzeugen Sie ein `regex`-Objekt aus einem klassischen C-String und einem `std::string`, was ohne Probleme möglich ist. Der Ausdruck in Zeile **03** prüft, ob die Zeichen alles Kleinbuchstaben zwischen »a« bis »z« sind. Der Ausdruck in Zeile **04** hingegen prüft, ob alle Zeichen Groß- und Kleinbuchstaben zwischen »A« bis »Z« und »a« bis »z« sind.

Ebenso ist es möglich, gleich beim Anlegen des Objekts den Ausdruck anzugeben, wie Zeile **05** demonstriert. Hier wird überprüft, ob nur Ziffern von 0 bis 9 enthalten sind. Der Ausdruck in Zeile **06** prüft, ob ein Ausdruck aus acht Dezimalzahlen besteht. Damit der Compiler das Metazeichen `\d` nicht als Escape-Sequenz betrachtet, müssen wir es hier mit einem Backslash davor schützen.

Allerdings kann mit C++11 jetzt auch ein roher String (Raw-String) verwendet werden, womit Sie solche Backslash-Orgien vermeiden können. In Zeile **07** wird ein solcher Raw-String verwendet. Mehr über Raw-Strings finden Sie in Abschnitt 4.2.4, »Raw-stringlitterale (Raw-String) (C++11)«.

Optionen

Zusätzlich zu den regulären Ausdrücken können Sie noch weitere Optionen mithilfe der Bitmaske `std::regex_constants` verwenden. Diese Option können Sie optional beim Konstruktor-Aufruf von `std::regex` wie folgt verwenden:

```
01 std::string extract(R"((\w+))");
02 std::regex expr(extract, std::regex_constants::icase);
```

Mit der Option `icase` hier in Zeile **02** wird bei der Suche der Unterschied zwischen Groß- und Kleinschreibung ignoriert. In Tabelle 14.4 finden Sie einen kurzen Überblick über weitere Optionen, die Sie verwenden können.

Flag	Bedeutung
<code>icase</code>	Groß- und Kleinschreibung ignorieren
<code>nosubs</code>	Teilausdrücke nicht im Matchergebnis speichern
<code>optimize</code>	Matchgeschwindigkeit optimieren, indem der Algorithmus den Ausdruck vorprozessiert. Macht eigentlich nur Sinn, wenn ein regulärer Ausdruck extrem häufig aufgerufen wird. Ansonsten sollte diese Optimierung gut überlegt sein, weil hierbei auch mehr Speicher benötigt wird.
<code>collate</code>	Die eingestellte Landessprache des Systems soll bei Ausdrücken wie <code>[a-zA-Z]</code> berücksichtigt werden.

Tabelle 14.4 Optionen, die bei Konstruieren eines regulären Ausdrucks verwendet werden können

ECMAScript-Syntax ändern

Neben diesen Optionen gibt es noch weitere, mit denen Sie die Syntax von regulären Ausdrücken ändern können. Standardmäßig verwendet C++ die ECMAScript-Syntax. Hierfür ist standardmäßig immer das Flag `std::regex_constants::ECMAScript` gesetzt. Allerdings kann auch die Syntaxvariante geändert werden. Hierfür bietet der Standard mindestens noch `awk` (Awk Posix Syntax), `grep` (Grep POSIX Syntax), `egrep` (Egrep POSIX Syntax), `extended` (Extended POSIX Syntax) und `basic` (Basic POSIX Syntax) an.

Die Algorithmen für reguläre Ausdrücke

Nachdem die Grundlagen zum Formatieren von regulären Ausdrücken bekannt sind, fehlt noch der wichtigste Baustein. Das Formulieren von regulären Ausdrücken allein hilft noch nicht viel ohne den entsprechenden Algorithmus, diese zu verwenden. Hierfür stehen Ihnen drei Algorithmen `std::regex_match`, `std::regex_search` und `std::regex_replace` (die alle mehrfach überladen sind) zur Verfügung, welche in Tabelle 14.5 aufgelistet sind. In den nachfolgenden Abschnitten sollen diese drei Algorithmen noch genauer behandelt werden.

Algorithmus	Bedeutung
<code>std::regex_match</code>	Überprüft, ob ein String dem regulären Ausdruck entspricht.
<code>std::regex_search</code>	Sucht nach dem regulären Ausdruck in einem Text.
<code>std::regex_replace</code>	Ersetzt jedes Vorkommen des regulären Ausdrucks durch einen anderen String.

Tabelle 14.5 Die klassischen Algorithmen für reguläre Ausdrücke

14.3 Suchergebnis analysieren

Bevor wir uns den einzelnen Algorithmen etwas genauer widmen können, müssen Sie noch Bescheid wissen, wie Sie an die Informationen des Suchergebnisses kommen. Für solche Zwecke steht mit `std::match_result` ein sequenzieller Container zur Verfügung. Abhängig davon, was für einen Typ Sie zur Suche verwenden, können Sie diesen Container mit folgenden Suchtypen instanziiieren:

- ▶ `smatch` für `std::string`
- ▶ `cmatch` für `const char*`
- ▶ `wsmatch` für `std::wstring`
- ▶ `wcmatch` für `const wchar_t*`

Mithilfe dieser Typen stehen Ihnen Informationen wie die Anzahl der Gesamttreffer, die Position, Teiltreffer, Länge usw. zur Verfügung. Je nachdem, wie viele **Gruppierungen** (welche Sie mit den runden Klammerungen definieren) Sie verwendet haben, finden Sie das Gesamtergebnis dann in `smatch[0]` und die Gruppenergebnisse in `smatch[1]`, `smatch[2]` usw.

Hierzu ein einfaches Beispiel:

```
00 // kapitel14/regex01.cpp
01 #include <regex>
...
02 std::string data("blabla Januar:1234:3221 blabla");
03 std::string extract(R"((\w+):(\d+):(\d+))"); // Raw-String
04 // Für Compiler ohne Raw-String-Support:
05 // std::string extract("(\\w+):(\\d+):(\\d+)");
06 std::regex expr(extract, std::regex_constants::icase);

07 std::smatch m;

08 std::regex_search(data, m, expr);
```

```

09  std::cout << m[0] << std::endl;    // = Januar:1234:3221
10  std::cout << m[1] << std::endl;    // = Januar
11  std::cout << m[2] << std::endl;    // = 1234
12  std::cout << m[3] << std::endl;    // = 3221

```

Zeile 02 dient als Suchstring. Den regulären Ausdruck legen wir in Zeile 03 fest. Und zwar soll hier Text gefunden werden mit beliebigen Zeichen (`\w+`), gefolgt von einem Doppelpunkt, dann mit einer beliebigen Anzahl von Ziffern (`\d+`), wieder gefolgt von einem Doppelpunkt und nochmals mit einer beliebigen Anzahl von Ziffern. Den regulären Ausdruck instanziiieren wir in Zeile 06, wobei hier zusätzlich noch mit der Option `icase` angegeben wird, dass wir nicht an der Groß- und Kleinschreibung interessiert sind.

In Zeile 07 definieren wir den Typ für die Teilergebnisse. `smatch` ist im Grunde nichts anderes als ein `typedef` für ein Template von `match_result` für die Suche in einem String. Im Beispiel haben wir den Ausdruck in drei Gruppen aufgeteilt (die Ausdrücke mit den runden Klammern). In Zeile 08 verwenden wir den Algorithmus `regex_search` zur Suche in `data` nach dem Ausdruck `expr`. Die Detaillieregebnisse werden in `m` gespeichert.

Das Ergebnis der Suche finden Sie jetzt in `m[0]` (was die Ausgabe von Zeile 09 auch bestätigt). Die einzelnen Gruppierungen werden auf den nächsten Indexnummern verteilt. Wie bereits erwähnt, wurden hier drei Gruppierungen definiert, weshalb Sie diese Teilergebnisse davon jetzt in den Indexnummern 1 bis 3 vorfinden, wie die Ausgaben der Zeilen 10 bis 12 beweisen sollen.

Ergebnis und Teilergebnis

Das Beispiel hat jetzt sehr schön gezeigt, dass es hier Ergebnisse und Teile davon gibt. Hier gibt es auch gleich noch eine Feinheit, die Sie kennen sollten. Das Suchergebnis von `smatch` ist vom Typ `match_results`. Die Erfassungsgruppe ist allerdings vom Typ `sub_match`. `sub_match` ist dabei eine Sequenz von Zeichen.

In der Praxis können Sie natürlich nicht einfach hergehen und das Ergebnis und die Teilergebnisse einfach so ohne Überprüfung verwenden. Das oben geschriebene Listing soll daher jetzt umgeschrieben werden, und dabei sollen auch gleich einige nützliche Methoden zum Einsatz kommen:

```

00  // kapitel14/regex02.cpp
01  #include <regex>
...
02  std::string data("blabla Januar:1234:3221 blabla");
03  std::string extract(R"((\w+):(\d+):(\d+))"); // Raw-String
04  // Für Compiler ohne Raw-String-Support:

```

```

05 // std::string extract("(\\w+):(\\d+):(\\d+)");
06 std::regex expr(extract, std::regex_constants::icase);
07 std::smatch m;

08 if (std::regex_search(data, m, expr) ) {
09     for(unsigned int i=0; i<m.size(); ++i) {
10         std::cout << "Position: " << m.position(i) << "/";
11         std::cout << "Länge: " << m.length(i) << " -> ";
12         std::cout << m.str(i) << std::endl;
13     }
14     std::cout << "prefix():" << m.prefix().str() << std::endl;
15     std::cout << "suffix():" << m.suffix().str() << std::endl;
16 }

```

Der Suchstring `data` und der reguläre Ausdruck wurde gegenüber Listing *regex01.cpp* nicht geändert. In Zeile **08** wurde jetzt der Aufruf des Algorithmus, in dem wir im Suchstring `data` nach dem Ausdruck `expr` suchen, in eine `if`-Bedingung gesteckt. Wenn die Suche erfolgreich verlief, liefert `regex_search` gleich `true` zurück, und der Anweisungsblock dahinter wird ausgeführt.

In den Zeilen **09** bis **13** durchlaufen wir jetzt das Suchergebnis mit einer `for`-Schleife. Als Abbruchbedingung dient hier `m.size()`. In der Methode `size()` ist die Anzahl der gefundenen Matches enthalten. Für jede Erfassungsgruppe geben wir jetzt die Position (Zeile **10**), die Länge (Zeile **11**) und den eigentliche String (Zeile **12**) mit den entsprechenden Methoden aus `match_results` aus.

Mehrere Möglichkeiten, um auf den String zuzugreifen

In der Praxis gibt es mehrere Möglichkeiten, um auf den gefundenen String in einem `match_result` zuzugreifen. Neben der Möglichkeit mit `m.str(i)` wie in Zeile **12** sind noch folgende Möglichkeiten gleichwertig:

```

m[i].str();    // wie m.str(i)
*(m.begin()+i) // wie m.str(i)

```

Mithilfe der Methoden `prefix()` und `suffix()` in den Zeilen **14** und **15** wird jeweils ein Iterator auf den Bereich vor der ersten Erfassungsgruppe (`m[0]`) bzw. ein Iterator eine Position nach der letzten Erfassungsgruppe (hier `m[3]`) zurückgegeben.

Das Programm bei der Ausführung:

```

Position: 7/Länge: 16 -> Januar:1234:3221
Position: 7/Länge: 6 -> Januar
Position: 14/Länge: 4 -> 1234

```

```
Position: 19/Länge: 4 -> 3221
prefix() : blabla
suffix() : blabla
```

Es spricht auch nichts dagegen, die gefundenen Erfassungsgruppen mit einem Iterator zu durchlaufen, da `match_results` letztendlich auch nur ein sequenzieller Container ist, in dem die Suchergebnisse gespeichert werden und der auch die Methoden `begin()` und `end()` anbietet. Hier der Codeausschnitt dazu:

```
00 // kapitel14/regex03.cpp
01 #include <regex>
...
02 if (std::regex_search(data, m, expr) ) {
03     for(auto it=m.begin(); it<m.end(); ++it) {
04         std::cout << it->str() << std::endl;
05         std::cout << it->length() << std::endl;
06     }
07     std::cout << "prefix():" << m.prefix().str()
               << std::endl;
08     std::cout << "suffix():" << m.suffix().str()
               << std::endl;
09 }
...
```

Hier muss jetzt allerdings hinzugefügt werden, dass in den Zeilen **04** und **05** die Methoden `str()` und `length()` von `sub_match` verwendet wurden. Das erklärt auch den indirekten Zugriff der beiden Methoden gegenüber dem Beispiel zuvor.

14.4 Algorithmen für reguläre Ausdrücke

Wie bereits kurz erwähnt, stehen Ihnen mit `regex_match`, `regex_search` und `regex_replace` drei Algorithmen zur Verfügung. Der Text, nach dem wir suchen, kann dabei auf unterschiedliche Weise übergeben werden. Ein Auflisten der überladenen Versionen an dieser Stelle ist nicht sehr schön anzusehen. Alles in allem gibt es hierbei zahlreiche Varianten, wie Sie die Algorithmen verwenden können. Für die Algorithmen `regex_match` und `regex_search` kann man die Varianten wie folgt zusammenfassen:

- Der erste Parameter ist in der Regel der Text, nach dem gesucht werden soll. Hierfür stehen Versionen für `std::string`, `const char*` oder ein Paar von Iteratoren für einen Bereich zur Verfügung.
- Optional kann als zweiter Parameter `match_results` mit bspw. `smatch`, `cmatch` etc. zur anschließenden Analyse des Suchergebnisses übergeben werden.

- Als zweiter oder (bei `match_result`) dritter Parameter wird das Muster `regex` verwendet, das gesucht werden soll.
- Als letzten Parameter können Sie hier noch optional weitere Flags für `regex_constants` verwenden.

Etwas anders hingegen sieht es beim Ersetzen mit `regex_replace` aus. Dieser Algorithmus lässt sich wie folgt zusammenfassen:

- Der erste Parameter ist der Text, nach dem gesucht werden soll. Auch hier gibt es wieder Versionen für `std::string`, `const char*` und ein Paar von Iteratoren für den Bereich.
- Der nächste Parameter ist immer der reguläre Ausdruck mit dem Muster in `regex`, nach dem gesucht werden soll.
- Als dritten Parameter geben Sie den Text an, welcher ersetzt werden soll, wenn ein Suchmuster gefunden wurde. Für den Ersetzungstext können Sie entweder `std::string` oder `const char*` verwenden.
- Als letzten Parameter können Sie auch hier noch optional weitere Flags für `regex_constants` verwenden.

14.4.1 Genaue Treffer mit `regex_match`

Der Algorithmus `regex_match` ist sehr gut dafür geeignet, um zu ermitteln, ob ein String dem regulären Ausdruck entspricht. Oft will man ja nur prüfen, ob eine Eingabe des Anwenders ein bestimmtes Kriterium erfüllt. Und dafür reicht häufig ein einfaches `regex_match` aus, welches `true` zurückgibt, wenn der String mit dem regulären Ausdruck übereinstimmt, oder bei nicht Übereinstimmung eben `false`.

Hier ein einfaches Beispiel dazu, in dem auf die Suchergebnisse mit `regex_results` verzichtet wurde. Hier soll einfach der reguläre Ausdruck auf `true` oder `false` getestet werden:

```
00 // kapitel14/regex04.cpp
01 #include <regex>
...
02 std::regex expr(R"(\d{5})"); // RAW-String!
03 std::string number;
04 std::cout << "Bitte 5 Zahlen eingeben: ";
05 std::getline(std::cin, number);

06 if( std::regex_match(number, expr) ) {
07     std::cout << "Eingabe korrekt\n";
08 }
09 else {
```

```

10  std::cerr << "Fehler bei der Eingabe\n";
11  }

12  std::vector<char> vecnum{'1', '8', '9', '2'}; // C++11-like
13  if( std::regex_match( vecnum.begin(), vecnum.end(),
                        std::regex(R"([0-9]+)") ) ) {
14      std::cout << "Alle Werte sind Ziffern\n";
15  }
16  else {
17      std::cerr << "Nicht alle Werte sind Ziffern\n";
18  }
...

```

In den Zeilen 04 bis 05 werden Sie aufgefordert, eine fünfstellige Ziffer einzugeben. Mit `regex_match` in Zeile 06 wird der eingegebene String mit dem regulären Ausdruck aus Zeile 02 überprüft. Der Ausdruck `\d{5}` entspricht fünf dezimalen Ziffern. Je nachdem, ob die Eingabe korrekt war (`true`), wird die Anweisung dahinter, oder bei `false` die `else`-Anweisung, ausgeführt.

In Zeile 13 wird auch noch demonstriert, wie Sie ein Paar von Iteratoren mit `regex_match` verwenden können. Hier prüfen wir, ob `vector<char>` in Zeile 12 nur Ziffern (`[0-9]+`) als Elemente enthält, was im abgedruckten Beispiel mit `true` bestätigt werden muss. Den regulären Ausdruck haben wir hier gleich im `regex_match`-Aufruf formuliert, was ebenfalls möglich ist.

Nicht immer reicht ein einfaches Wahr (`true`) oder Unwahr (`false`) bei `regex_match` aus. In dem Fall ist es hiermit auch möglich, das Suchergebnis wieder mit `match_result` zu analysieren. Hierzu will ich Ihnen noch ein Beispiel demonstrieren, welches zunächst eine E-Mail-Adresse vom Anwender erwartet und diese E-Mail-Adresse auf Gültigkeit hin überprüft. Wurde eine gültige E-Mail-Adresse eingegeben, soll diese noch in einen lokalen und einen Domain-Teil zerlegt werden. Hier das Beispiel dazu:

```

00 // kapitel14/regex05.cpp
01 #include <regex>
...
02 std::regex expr(
    R"(([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,6}))",
    std::regex_constants::icase );
03 // Für Compiler ohne Raw-String-Support:
04 // std::regex expr(
    //      "([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,6})",
    //      std::regex_constants::icase );
05 std::string mail;
06 std::cout << "Bitte E-Mail-Adresse eingeben: ";

```



```

07  std::getline(std::cin, mail);
08  std::smatch m;

09  if( std::regex_match(mail, m, expr) ) {
10    if(m.size() == 3) {
11      std::cout << "E-Mail-Adresse:" << m.str(0) << std::endl;
12      std::cout << "Lokaler Teil  :" << m.str(1) << std::endl;
13      std::cout << "Domain-Teil   :" << m.str(2) << std::endl;
14    }
15  }
16  else {
17    std::cerr << "Keine gültige E-Mail-Adresse\n";
18  }
...

```

Der reguläre Ausdruck in Zeile 02 sieht komplexer aus, als er ist, und prüft auf eine gültige E-Mail-Adresse. Zusätzlich wurde der Ausdruck in zwei Gruppierungen (die runden Klammern) aufgeteilt, zum einen in den lokalen Teil der E-Mail-Adresse und zum anderen in den Domain-Teil. In Zeile 08 legen wir den Typ für unser Suchergebnis an, welcher `match_results` ist. In Zeile 09 prüfen wir dann die eingegebene E-Mail-Adresse `mail` mit dem regulären Ausdruck `expr`. Wenn hier `true` zurückgegeben wird, dann handelt es sich um eine gültige E-Mail-Adresse. Anschließend prüfen wir in Zeile 10, ob hier auch alle drei Erfassungsgruppen ermittelt wurden. Hierbei müssen Sie immer die Anzahl der Gruppierungen +1 rechnen. In den Zeilen 11 bis 13 werden dann die Strings der einzelnen Erfassungsgruppen ausgegeben.

Das Programm bei der Ausführung:

```

Bitte E-Mail-Adresse eingeben: webmaster@elementaria.org
E-Mail-Adresse: webmaster@elementaria.org
Lokaler Teil   : webmaster
Domain-Teil    : elementaria.org
...
Bitte E-Mail-Adresse eingeben: wolf@ pronix.de>
Keine gültige E-Mail-Adresse

```

Gültige E-Mail-Adresse

Natürlich ist der reguläre Ausdruck für eine gültige E-Mail-Adresse hier nicht wasserdicht. Es ist trotzdem möglich, eine zwar gültige, aber dafür unechte Adresse wie `qwer@qwer.qwe` einzugeben. Wollen Sie etwas tiefer in diesen Ausdruck einsteigen, finden Sie auf der Webseite <http://www.regular-expressions.info/email.html> eine hervorragende Einführung dazu.

Im Allgemeinen gilt, dass die Überprüfung von E-Mail-Adressen keine solch triviale Angelegenheit ist, weshalb ich den oben erstellten Quellcode nicht in dieser Form in der realen Welt verwenden würde. Der Code dient einfach nur der Demonstration für `regex_match`.

14.4.2 Erweiterte Suche mit `regex_search`

Wenn sich die Suche nicht mehr nur auf einen ganzen String beschränken soll, sondern auf Teile eines Strings erweitert werden soll, dann ist der Algorithmus `regex_search` besser für eine weitergreifende Suche geeignet. Bezogen auf unser Beispiel in Listing *regex05.cpp* gibt folgender Ausdruck `false` zurück:

```
Bitte E-Mail-Adresse eingeben: J.Wolf, <wolf@elementaria.org>
Keine gültige E-Mail-Adresse
```

Diese Form von E-Mail-Adressen findet man z.B. durchaus häufig bei Kontaktverwaltungen. Hier scheitert `regex_match` dann logischerweise, weil dieser Algorithmus nur den kompletten String vergleicht und nur dann `true` zurückgibt, wenn der komplette String dem Ausdruck entspricht.

Ändern Sie jetzt in Listing *regex05.cpp* Zeile 07 wie folgt um:

```
00 // kapitel14/regex06.cpp
...
07 if( std::regex_search(mail, m, expr) ) {
```

Jetzt ist unser Listing fähig, den E-Mail-Teil in einem String zu finden, falls einer vorhanden ist. Hierzu das Beispiel erneut bei der Ausführung, nur jetzt mit dem Algorithmus `regex_search` statt `regex_match`:

```
Bitte E-Mail-Adresse eingeben: J.Wolf, wolf@elementaria.org
E-Mail-Adresse: wolf@elementaria.org
Lokaler Teil  : wolf
Domain-Teil   : elementaria.org
```

Dieses kleine Beispiel demonstriert sehr schön den Unterschied zwischen `regex_match` und `regex_search`.

Mit `regex_search` ist es jetzt möglich, einzelne Ausdrücke aus ganzen Dateien zu finden. Das folgende Listing soll eine beliebige Datei nach allen E-Mail-Adressen durchsuchen und ausgeben:

```

00 // kapitel14/regex07.cpp
01 #include <regex>
...
02 std::regex expr(
    R"(([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,6}))",
    std::regex_constants::icase );
03 // Für Compiler ohne Raw-String-Support:
04 // std::regex expr(
    // "([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,6})",
    // std::regex_constants::icase );
05 std::string line, fname;
06 std::cout << "Dateinamen eingeben: ";
07 std::getline(std::cin, fname);
08 std::ifstream file(fname);
09 if( !file ) { return -1; }
10 std::smatch m;
11 while( std::getline(file, line)) {
12     if( std::regex_search(line, m, expr) ) {
13         if(m.size() == 3) {
14             std::cout << "E-Mail-Adresse: " << m.str(0)
                << std::endl;
15             std::cout << "Lokaler Teil  : " << m.str(1)
                << std::endl;
16             std::cout << "Domain-Teil  : " << m.str(2)
                << std::endl;
17         }
18     }
19 }

```

In den Zeilen 03 bis 09 kümmern wir uns um das Öffnen einer Datei. Die Fehlerbehandlung mit `return -1` sollten Sie natürlich mit einer passenden Fehlermeldung machen. Wenn die Datei erfolgreich geöffnet wurde, lesen wir in der `while`-Schleife (Zeile 11) eine Zeile der Datei ein und prüfen in Zeile 12 mit `regex_search`, ob etwas gefunden wurde (also `true` zurückgegeben wurde). Ist dies der Fall, erfolgt die Ausgabe der E-Mail-Adresse mitsamt dem lokalen und dem Domain-Teil. Weiter geht es dann mit dem Einlesen der nächsten Zeile in Zeile 11. Dies geschieht so lange, bis das Dateiende erreicht wurde.

Das Programm bei der Ausführung:

```

Dateinamen eingeben: data.html
E-Mail-Adresse: JohnWayne@wayne2000.de
Lokaler Teil  : JohnWayne
Domain-Teil   : wayne2000.de

```

E-Mail-Adresse: StephanKing1@kingsworld.at
 Lokaler Teil : StephanKing1
 Domain-Teil : kingsworld.at

14.4.3 Ersetzen mit `regex_replace`

Als dritter Algorithmus mit `regex_replace` steht Ihnen eine Funktion zur Verfügung, mit der Sie einen durch einen regulären Ausdruck gefundenen String durch einen neuen String ersetzen können. Als Rückgabewert wird der verarbeitete String als `basic_string` zurückgegeben. Wenn Sie statt des Strings, der ersetzt werden soll, zwei Iteratoren verwenden, dann wird stattdessen die Kopie eines Ausgabe-Iterators, wie bspw. `std::ostream` verwendet.

Im folgenden Listing soll eine beliebige Datei geöffnet werden, wo wir alle `@`-Zeichen durch ein `(AT)` ersetzen wollen, was bspw. nützlich sein kann um Spammern das Ausfiltern von E-Mail-Adressen zu erschweren. Hier das Beispiel mit einer anschließenden Erläuterung dazu:

```
00 // kapitel14/regex08.cpp
01 #include <regex>
...
02 std::regex expr("@");
03 std::string line, fname, full;
04 std::cout << "Dateinamen eingeben: ";
05 std::getline(std::cin, fname);
06 std::ifstream file(fname);
07 if( !file ) { return -1; }
08 std::ofstream backup("backup.bak");
09 if( !backup ) { return -1; }
10 while( std::getline(file, line) ) {
11     std::cout << std::regex_replace(line, expr, "(AT)");
12     backup << std::regex_replace(line, expr, "(AT)");
13     full+=(std::regex_replace(line, expr, "(AT)"))+'\n';
14 }
15 std::cout << std::endl << full << std::endl;
...
```

In den Zeilen **04** bis **07** werden Sie aufgefordert, eine Datei zu öffnen, in der alle `@`-Zeichen anschließend durch den String `»(AT)«` ersetzt werden sollen. Die Originaldatei bleibt unberührt. In Zeile **08** erzeugen wir eine Datei, in die wir die anschließende Ersetzung auch reinschreiben. Jetzt lesen wir in Zeile **10** die geöffnete Datei Zeile für Zeile aus. In den Zeilen **11** bis **13** werden drei verschiedene Möglichkeiten demonstriert, durch die die Ersetzung immer jeweils woanders ausgegeben wird.

In Zeile 11 geben wir die Ersetzung der Zeile ganz einfach auf `std::cout` aus. Hierbei wird jeweils in der Zeile `line` nach dem Ausdruck `expr` (hier einfach das Zeichen `@`) gesucht, und er wird durch die Zeichenfolge »(AT)« ersetzt. Auch wenn keine Ersetzung stattgefunden hat, also kein `@` in der Zeile `line` gefunden wurde, wird der komplette String in `line` durch `regex_replace` zurückgegeben.

Dasselbe machen wir erneut mit derselben Zeile in Zeile 12, nur schreiben wir hier die Zeile in den Datei-Stream `backup`. Zu guter Letzt stecken wir dieselbe Zeile `line` auch noch in einen String in Zeile 13 – genauer wir hängen mit `operator+=` immer die neu eingelesene Zeile hinten mit einem Newline-Zeichen an. Wie gesagt, alle drei Beispiele in den Zeilen 11 bis 13 dienen nur der Demonstration, wie Sie in gängigen Situationen (Standardausgabe, in eine Datei oder einen String) den Algorithmus verwenden können.

In STL-Manier mit Iteratoren

Neben der eben gezeigten Version, wo der String der Ersetzung als Rückgabewert zurückgegeben wird, bietet die `regex`-Bibliothek auch noch eine typische STL-Version mit Iteratoren an, welche ich Ihnen hier nicht vorenthalten will. Das folgende, vereinfachte und verkürzte Beispiel soll diese Version demonstrieren:

```
00 // kapitel14/regex09.cpp
01 #include <regex>
...
02 std::regex expr("@");
03 std::string fname, full, newFull;
04 std::cout << "Dateinamen eingeben: ";
05 std::getline(std::cin, fname);
06 std::ifstream file(fname);
07 if( !file ) { return -1; }
08 std::stringstream ss;
09 ss << file.rdbuf();
10 full = ss.str();
11 std::regex_replace( back_inserter(newFull), full.begin(),
                    full.end(), expr, "(AT)");
12 std::cout << newFull << std::endl;
...
```

Das Beispiel macht im Grunde nichts anderes als unser Beispiel in Listing *regex8.cpp* zuvor. Wir öffnen eine Datei, in der alle `@`-Zeichen durch die Zeichenfolge »(AT)« ersetzt werden sollen. Zuvor schieben wir den kompletten Inhalt der Datei in Zeile 09 mit `rdbuf()` in einen Stream-Puffer, wo wir eine Zeile später den Inhalt an einen String zuweisen. Bei der Ersetzung in Zeile 11 werden jetzt im Iteratorbereich `full.begin()` bis `full.end()` alle Ausdrücke `expr` durch die Textfolge »(AT)« ersetzt

und an das Ende (`back_inserter()`, siehe den Abschnitt »Inserter (Einfüge-Iteratoren)« in Abschnitt 12.4.3) des Strings `newFull` gehängt.

14.4.4 Suchen mit `regex_iterator` und `regex_token_iterator`

Die Bibliotheken zu den regulären Ausdrücken bieten zusätzlich noch zwei mächtige Iteratoren mit an, welche Sie unbedingt kennen sollten. Dies wären der `regex_iterator` und der `regex_token_iterator`.

`regex_iterator`

Betrachten Sie nochmals das Beispiel aus Listing *regex07.cpp* aus Abschnitt 14.4.2, »Erweiterte Suche mit `regex_search`«, wo wir aus einer beliebigen Datei alle E-Mail-Adressen gesucht und ausgegeben haben. In dem Beispiel haben wir in einer Schleife Zeile für Zeile aus einer Datei ausgelesen und mit `regex_search` nach E-Mail-Adressen in dieser Zeile gesucht und diese mithilfe eines `match_result`-Objekts in die einzelnen Teile (lokaler Teil und Domain-Teil) zerlegt.

Mit `regex_iterator` geht dies noch viel einfacher. Dieser Iterator ist eine Adapterklasse und benutzt intern `regex_search` für die Suche. Damit können Sie bequem über alle gefundenen Treffer iterieren, ohne eine Schleife verwenden zu müssen. Auch hier gibt es wieder jeweils Spezialisierungen für `std::string`, `const char*`, `wstring` und `const wchar_t*` mit den entsprechenden Präfixen am Anfang (`sregex_iterator`, `cregex_iterator` usw.). Hierzu das Beispiel aus *regex07.cpp*, jetzt umgeschrieben mit `sregex_iterator`:

```
00 // kapitel14/regex10.cpp
01 #include <regex>
...
02 std::regex expr(
    R"(([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,6}))",
    std::regex_constants::icase );
03 // Für Compiler ohne Raw-String-Support:
04 // std::regex expr(
    // "([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,6})",
    // std::regex_constants::icase );
05 std::string full, fname;
06 std::cout << "Dateinamen eingeben: ";
07 std::getline(std::cin, fname);
08 std::ifstream file(fname);
09 if( !file ) { return -1; }
10 std::stringstream ss;
11 ss << file.rdbuf();
12 full = ss.str();
```

```

13  std::sregex_iterator pos(full.begin(), full.end(), expr);
14  std::sregex_iterator end;
15  for(; pos!=end; ++pos) {
16      std::cout << "E-Mail : " << pos->str(0) << std::endl;
17      std::cout << "Lokal  : " << pos->str(1) << std::endl;
18      std::cout << "Domain : " << pos->str(2) << std::endl;
19  }
...

```

Das Einlesen einer beliebigen Datei in den String `full` wird in den Zeilen 03 bis 12 durchgeführt und bedarf wohl keiner Erläuterung mehr. Auf `std::smatch` können Sie hier verzichten, weil unsere Adapterklasse `sregex_iterator` bereits auf eine entsprechende `match_result`-Instanz verweist. Sie müssen nur noch über die Ergebnisse iterieren. Wenn Zeile 13 ausgeführt wurde, enthält `pos` bereits alle gefundenen Treffer aus dem Bereich `full.begin()` bis `full.end()`, welche auf das Muster `expr` passen. Das ist quasi schon unsere Schleife, in der nach dem Ausdruck `expr` gesucht wird. Mit Zeile 14 verwenden Sie den Standardkonstruktor, der automatisch als Iterator für das Ende definiert wird. In der `for`-Schleife (Zeile 15 bis 19) werden jetzt alle gefundenen Treffer durchlaufen und ausgegeben. Wollen Sie wissen, wo Übereinstimmungen (hier komplette E-Mail-Adressen) gefunden wurden, können Sie die Iterator-Funktion `std::distance()` (siehe Abschnitt »distance()« in Abschnitt 12.4.2) dafür wie folgt verwenden:

```
std::cout << std::distance(pos, end) << " Elem. gefunden\n";
```

Die Ausgabe der Elemente in den Zeilen 15 bis 19 können Sie dank der beiden vorhandenen Iteratoren jetzt auch mit Algorithmen verwenden. Eine gleichwertige Alternative zu den Zeilen 15 bis 19 mit dem Algorithmus `for_each` und der Lambda-Funktion könnte wie folgt aussehen:

```

...
std::for_each(pos, end, [](const std::smatch& m) {
    std::cout << "E-Mail : " << m.str(0) << std::endl;
    std::cout << "Lokal  : " << m.str(1) << std::endl;
    std::cout << "Domain : " << m.str(2) << std::endl;
});
...

```

Suchergebnis komfortabel analysieren

Jeder (Teil-)String, welcher zum regulären Ausdruck passt, wird über eine `match_result`-Instanz zur Verfügung gestellt. Damit steht Ihnen der komplette Funktionsumfang von `match_result` für die Weiterverarbeitung zur Verfügung.

regex_token_iterator

Während `regex_iterator` auf ein `regex_match`-Ergebnis zeigt, zeigt `regex_token_iterator` auf ein `sub_match`. Des Weiteren können Sie mit `regex_token_iterator` gezielt angeben, welche Komponenten einer Gruppierung (in den runden Klammern) angesprochen werden soll.

Auch hier gibt es wieder jeweils Spezialisierungen für `std::string`, `const char*`, `wstring` und `const wchar_t*` mit den entsprechenden Präfixen am Anfang (`sregex_token_iterator`, `cregex_token_iterator` usw.).

Um Ihnen den Iterator etwas besser demonstrieren zu können, soll jetzt erst einmal das Beispiel aus *regex10.cpp* für `sregex_token_iterator` umgeschrieben werden:

```
00 // kapitel14/regex11.cpp
01 #include <regex>
...
02 std::regex expr(
    R"(([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,6}))",
    std::regex_constants::icase );
...
// Einlesen der Datei in einen String wie in regex10.cpp
...

03 std::sregex_token_iterator pos( full.begin(),
                                full.end(),expr);
04 std::sregex_token_iterator end;
05 for(; pos!=end; ++pos) {
06     std::cout << "E-Mail: " << pos->str() << std::endl;
07 }
```

Die Verwendung ist ähnlich wie schon mit `regex_iterator`. Nur können Sie jetzt beim Suchergebnis, wie hier in Zeile 06, nicht mehr den Indexwert verwenden, weil `regex_token_iterator` auf ein `sub_match`-Ergebnis zeigt. Im Beispiel wird daher nur die komplette E-Mail-Adresse ausgegeben, wenn eine gefunden wurde.

Das Programm bei der Ausführung:

```
Dateinamen eingeben: data.html
E-Mail: JohnWayne@wayne2000.de
E-Mail: StephanKing1@kingworld.at
```

Auch hier können Sie natürlich wieder einen Algorithmus verwenden. Die Zeilen 04 bis 07 nochmals mit dem Algorithmus `for_each` und der Lambda-Funktion als gleichwertiges Gegenstück:


```
std::for_each(pos, end, [](const std::ssub_match& m) {
    std::cout << "E-Mail : " << m.str() << std::endl;
});
```

Auch wenn Sie hier einen Zeiger auf `sub_match`-Ergebnisse haben, können Sie trotzdem die Erfassungsgruppe mit einem Index ansprechen. Diesen Indexwert müssen Sie beim Konstruktor von `regex_token_iterator` mit angeben. Wollen Sie bspw. nur durch die zweite Gruppierung des regulären Ausdrucks (in unserem Beispiel den Domain-Namen) iterieren, so brauchen Sie nur Zeile 03 in Listing *regex11.cpp* wie folgt zu ändern:

```
03 std::sregex_token_iterator pos( full.begin(),
                                full.end(),expr, 2 );
```

Jetzt sieht die Ausgabe des Programms wie folgt aus:

```
Dateinamen eingeben: data.html
E-Mail: wayne2000.de
E-Mail: kingsworld.at
```

Wollen Sie auf die erste Gruppierung mit dem lokalen Namen zugreifen (also auf die ersten runden Klammerungen im regulären Ausdruck), müssten Sie eben 1 anstelle von 2 verwenden. Über diesen Index können Sie somit jederzeit auf einzelne Gruppierungen des Ausdrucks zugreifen. Die Standardeinstellung ist hierbei der Wert 0. Hierzu nochmals der Überblick, welches Element der Gruppierung erfasst werden soll, wenn Sie einen optionalen vierten Parameter im Konstruktor verwenden:

- ▶ 0 – das ist die Standardeinstellung, womit Sie an den regulären Ausdruck als Ganzes wollen.
- ▶ n – ein Wert mit n bedeutet, dass Sie an dem Inhalt einer bestimmten Gruppierung interessiert sind. Die erste Gruppierung hat den Index 1. Der maximale Wert für n hängt natürlich auch davon ab, in wie viele Gruppierungen (runde Klammern) Sie den regulären Ausdruck aufgeteilt haben.
- ▶ -1 – Mit -1 geben Sie an, dass Sie an allem anderen außer dem regulären Ausdruck interessiert sind.

Die letzte Option mit dem Wert -1 ist besonders nützlich, wenn Sie bspw. Daten, die anhand eines bestimmten Zeichens getrennt sind, verarbeiten müssen. Ein klassisches und einfaches Beispiel wären da bspw. Daten, welche durch ein Komma oder Semikolon getrennt werden – sogenannte CSV-Dateien (*Comma-Seperated-Values*). Eine solche Datei kann bspw. wie folgt aussehen:

```
Tag;Einnahmen;Ausgaben
Montag;1342;1234
Dienstag;3243;2342
```

```
Mittwoch;3123;1243
Donnerstag;2342;3234
Freitag;1432;923
```

Das folgende Listing soll jetzt demonstrieren, wie Sie eine solche CSV-Datei mithilfe eines `regex_token_iterator` ordentlich formatiert ausgeben können:

```
00 // kapitel14/regex12.cpp
01 #include <regex>
...
02 std::string full, fname;
03 std::cout << "Dateinamen eingeben: ";
04 std::getline(std::cin, fname);
05 std::ifstream file(fname);
06 if( !file ) { return -1; }
07 std::stringstream ss;
08 ss << file.rdbuf();
09 full = ss.str();

10 std::regex expr("[;\\n\\r]");
11 std::sregex_token_iterator pos( full.begin(), full.end(), expr, -1 );
12 std::sregex_token_iterator end;

13 int count_for_nl = 0;
14 for(; pos!=end; ++pos) {
15     std::cout << std::setw(15) << std::left
                << pos->str() << '|' << '\t';
16     ++count_for_nl;
17     if(count_for_nl == 3) {
18         std::cout << std::endl;
19         count_for_nl = 0;
20     }
21 }
...
```

Durch den Wert `-1` als vierten Parameter beim Konstruktor `sregex_token_iterator` in Zeile 10 wird jetzt alles erfasst, was nicht dem regulären Ausdruck aus Zeile 10 entspricht. Hier werden praktisch `';`, `'\n'` und `'\r'` ignoriert. Die Zeichen für einen Zeilenumbruch müssen hier natürlich auch berücksichtigt werden. Der restliche Code in den Zeilen 13 bis 21 kümmert sich nur noch um die formatierte Ausgabe der einzelnen Werte, welche nicht dem regulären Ausdruck entsprechen.

Das Programm bei der Ausführung:

Dateinamen eingeben: **woche01.csv**

Tag		Einnahmen		Ausgaben	
Montag		1342		1234	
Dienstag		3243		2342	
Mittwoch		3123		1243	
Donnerstag		2342		3234	
Freitag		1432		923	

Kapitel 15

Multithreading (C++11)

Mit C++11 wurde endlich die Multithreading-Fähigkeit eingeführt, um auch in dieser Sprache standardisierte Programme für moderne Mehrkern- und Mehrprozessorsysteme zu schreiben. Bisher mussten hierfür immer Third-Party-Bibliotheken verwendet werden.

Hier muss gleich zu Beginn angemerkt werden, dass Multithreading ein extrem umfangreiches Thema ist. Daher kann dieses Kapitel nur auf die Grundlagen eingehen. Sie werden erfahren, welche Funktionen es gibt und wann und wofür Sie diese einsetzen können. Das Kapitel streift also nur die berühmte Spitze des Eisberges (vielleicht auch ein bisschen mehr).

15.1 Die Grundlagen

Zunächst soll hier noch kurz der Unterschied zwischen *Multiprocessing* und *Multithreading* deutlich gemacht werden. Ihre bisherigen Anwendungen, die Sie geschrieben haben, waren Multiprocessing-Programme.

Ein Prozess ist das, was Sie bei den bisherigen Beispielen des Buches immer erstellt haben – also ein Programm bei seiner Ausführung. Diesem Programm steht zum Zeitpunkt seiner Ausführung die komplette Verwaltungseinheit des Betriebssystems zur Verfügung, als wäre es das einzige Programm, das ausgeführt wird.

Zum Zeitpunkt der Ausführung erhält ein Prozess somit die volle Aufmerksamkeit der CPU. Eine solche alleinige Verfügbarkeit ist natürlich bei einem Multitasking-Betriebssystem zeitlich begrenzt. Üblicherweise laufen auf einem Betriebssystem mehrere Prozesse »gleichzeitig« ab (daher auch der Begriff Multiprocessing). Auch wenn Sie selbst noch kein Programm gestartet haben, laufen bereits viele Dienste des Systems im Hintergrund ab. Egal, welches Programm Sie nun ausführen, jede Anwendung bekommt eine gewisse Rechenzeit der CPU zur Verfügung gestellt, bevor die CPU die Rechenzeit einem anderen Prozess widmet (siehe Abbildung 15.1).

Mit dieser Technik, dem Multitasking, hat es den Anschein, dass alle Prozesse quasi gleichzeitig ablaufen – was natürlich nicht möglich ist, da pro CPU immer nur ein Task ausgeführt werden kann.

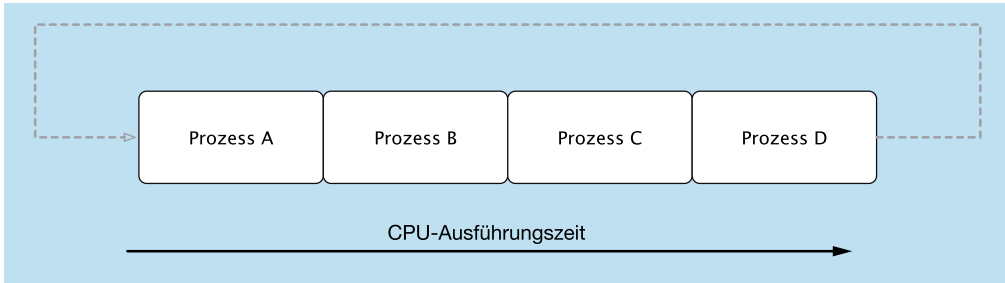


Abbildung 15.1 Beim Multiprocessing werden einzelne Prozesse Task für Task abgearbeitet.

Der gravierende Unterschied zwischen den Threads und den Prozessen besteht darin, dass Threads unabhängige Befehlsfolgen innerhalb eines Prozesses sind. Man könnte auch sagen, Threads sind in einem Prozess gefangen oder gekapselt (siehe Abbildung 15.2).

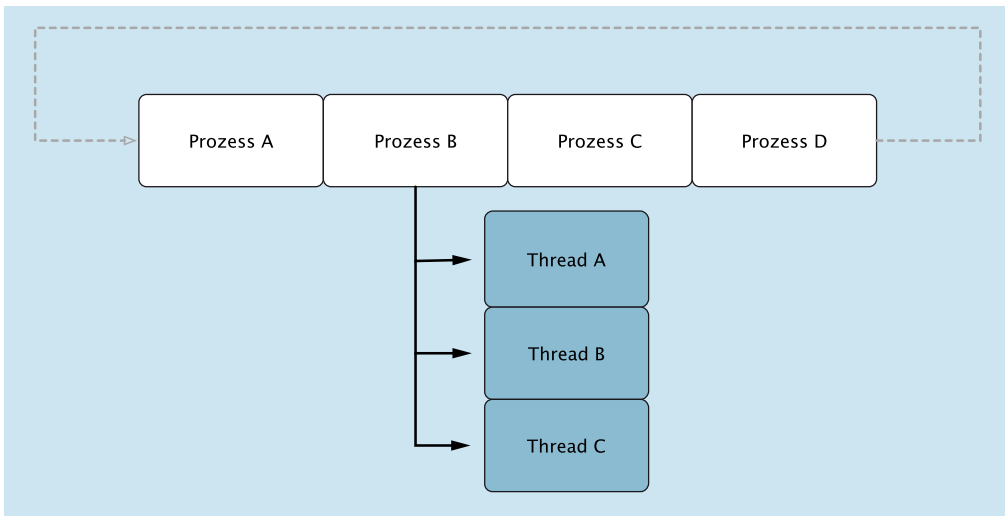


Abbildung 15.2 Ein Prozess (Main-Thread) ist immer nötig, um weitere (Unter-)Threads zu starten.

Prozesse werden gegenüber Threads immer als schwergewichtig dargestellt. Das kommt daher, weil Prozesse einen eigenen Adressraum haben und Threads nicht. Beim Wechsel von Prozessen kostet dies sehr viel Rechenleistung, von einem Adressraum in den anderen zu wechseln, u.a. weil hierfür bspw. das Prozessorregister umgeschrieben werden muss usw. Bei den Threads hat man diesen Aufwand nicht, da diese in einem gemeinsamen Adressraum ablaufen. Somit stehen den einzelnen Threads dasselbe Codesegment, Datensegment, der Heap und alle anderen Zustandsdaten, die ein »gewöhnlicher« Prozess besitzt, zur Verfügung – was somit auch die

Arbeit beim Austausch von Daten und bei der Kommunikation untereinander erheblich erleichtert. Weil aber kein Speicherschutzmechanismus unter den Threads vorhanden ist, bedeutet dies auch, dass ein Thread, wenn er abstürzt, alle anderen Threads mitreißt. Einen Vergleich des Verwaltungsaufwands zwischen den Threads und Prozessen finden Sie in Abbildung 15.3 und Abbildung 15.4.

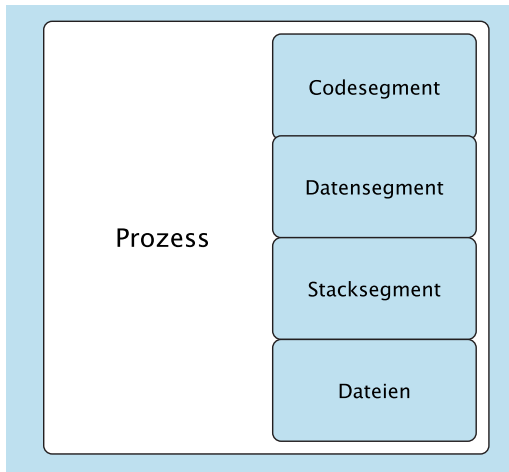


Abbildung 15.3 Vereinfachte Aufteilung eines Prozesses

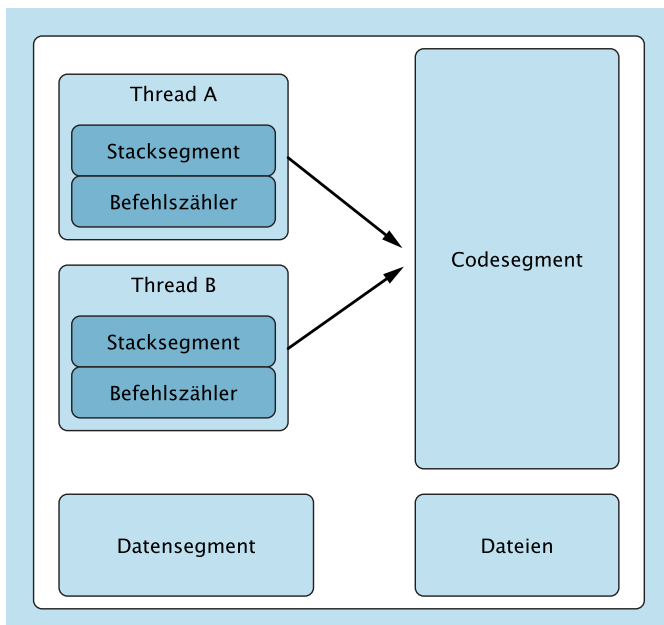


Abbildung 15.4 Threads teilen sich die Ressourcen. Sinngemäß hat jeder Thread ein eigenes Stacksegment und einen eigenen Befehlszähler.

In Abbildung 15.4 lässt sich schön erkennen, dass sich Threads den Adressraum teilen. Allerdings macht gerade dies die Threads etwas aufwendiger zu programmieren, denn die Threads teilen sich hierbei die globalen Variablen, geöffneten Dateien – eben alle globalen Ressourcen außerhalb eines Threads –, weshalb hierbei gewisse Synchronisationsmechanismen nötig sind.

Mehrprozessor-System

Um echtes Multithreading zu verwenden, ist es selbstverständlich nötig, dass ein Rechner mit mehreren Prozessoren bzw. Prozessorkernen vorhanden ist. Nur in diesem Fall ist echte Parallelität möglich. Um die Aufteilung der einzelnen Threads und darum, was für Arbeit welcher Prozessor erhält, kümmert sich das Betriebssystem (das wird auch als *Scheduling* bezeichnet).

15.2 Threads erzeugen

Um Multithreading von C++11 verwenden zu können, müssen Sie immer die Headerdatei `<thread>` inkludieren. Auch bei den Threads finden Sie wieder alles im Namensbereich `std`. Der Typ, den Sie für Multithreading benötigen, lautet `thread`.

Um einen neuen Thread zu erzeugen stehen Ihnen in C++ drei verschiedene Möglichkeiten zur Verfügung:

- ▶ eine normale globale Funktion
- ▶ ein Funktionsobjekt
- ▶ eine Lambda-Funktion

Das erste Beispiel soll Ihnen gleich alle drei Möglichkeiten in der Praxis demonstrieren:

```
01 // kapitel15/thread01.cpp
02 #include <thread>
...
03 void func() {
04     std::cout << "Thread als Funktion\n";
05 }

06 class funcObj {
07 public:
08     void operator()() const {
09         std::cout << "Thread als Funktionsobjekt\n";
10     }
11 };
```

```

...
12  std::thread t1(func);
13  funcObj fobj;
14  std::thread t2(fobj);
15  std::thread t3([] {
    std::cout << "Thread als Lambda-Funktion\n";
  });
16  t1.join();
17  t2.join();
18  t3.join();
...

```

In Zeile 12 erzeugen wir einen neuen Thread mit einer Funktion, welche in den Zeilen 03 bis 05 definiert wurde. Einen Thread aus einem Funktionsobjekt, welches in den Zeilen 06 bis 11 definiert wurde, erzeugen wir in Zeile 14. Und zu guter Letzt erzeugen wir in Zeile 15 noch einen Thread mit einer Lambda-Funktion. Damit sich der Hauptthread (main-Funktion) nicht vor den einzelnen Threads beendet, warten wir mit `join()` in den Zeilen 16 bis 18 auf das Ende der einzelnen Threads. Aber dazu gleich mehr.

Wenn Sie das Programm ausführen, lässt es sich nicht genau sagen, in welcher Reihenfolge die Ausgabe erfolgt. Alle drei Threads verwenden das gleiche Stream-Objekt `cout` und schreiben wahllos etwas rein. Es kann sogar passieren, dass irgendein Datensalat von Buchstaben ausgegeben wird. Wie Sie gemeinsam genutzte Daten vor dem unkontrollierten Zugriff mehrerer Threads schützen können, erfahren Sie noch in Abschnitt 15.3, »Gemeinsame Daten synchronisieren«.

15.2.1 Argumente für den Thread

Argumente können einem Thread ebenfalls übergeben werden. Diese können Sie beim Anlegen des Objekts gleich mit übergeben. Folgender Codeausschnitt demonstriert diese wieder für alle drei Fälle (Funktion, Funktionsobjekt und Lambda-Funktion):

```

01  // kapitel15/thread02.cpp
02  #include <thread>
...
03  void func(std::string str) {
04      std::cout << "Thread als Funktion\n";
05      std::cout << "Daten(Funktion): " << str << std::endl;
06  }

07  class funcObj {
08  public:

```



```

09 void operator()(double val) const {
10     std::cout << "Thread als Funktionsobjekt\n";
11     std::cout << "Daten(Funktor): " << val << std::endl;
12 }
13 };
...
14 std::string text("Ein Text");
15 std::thread t1(func, text);
16 funcObj fobj;
17 std::thread t2(fobj, 3.1415);
18 int ival = 1234;
19 std::thread t3([] (int ival, string text) {
    std::cout << "Thread als Lambda-Funktion\n";
    std::cout << "Daten(Lambda): " << ival << std::endl;
    std::cout << "Daten(Lambda): " << text << std::endl;
}, ival, text);
20 t1.join();
21 t2.join();
22 t3.join();
...

```

In den Zeilen **15**, **17** und **19** werden jeweils Argumente an den entsprechenden Thread übergeben. Natürlich können Sie hier auch mehrere Argumente übergeben, wie dies bspw. bei der Lambda-Funktion in Zeile **19** gemacht wurde.

Kopien statt Referenzen bzw. Zeiger

Bei den Threads werden die Daten an den Thread als Wert übergeben, wie es im Beispiel auch zu sehen ist, anstatt Referenzen oder Zeiger zu verwenden. Dadurch werden Probleme mit der Lebenszeit der Daten minimiert, welche auftreten würden, wenn eine Variable ihre Gültigkeit verliert, was durchaus passieren kann, wenn der Thread den Aufrufer überlebt. Im unserem Beispiel könnte dies zwar nicht konkret passieren, weil wir in den Zeilen **20** bis **22** mit `join()` auf das Ende der Threads warten. Würden Sie hier allerdings die Threads mit `detach()` vom Hauptthread loslösen, dann hätten Sie in Zeile **14** mit dem String `text` einen solchen Kandidaten. Wenn sich hier der Hauptthread bspw. vor dem Thread `t1` beendet und Sie hätten in Zeile **15** einen Zeiger statt einer Kopie übergeben, dann würden Sie einen String verwenden, der offiziell gar nicht mehr vorhanden ist. Daher gilt, wenn Sie trotzdem Referenzen oder Zeiger verwenden wollen/müssen, sollten Sie diese Daten extrem kritisch überprüfen.

Bei der Ausgabe des Programms finden Sie auch hier wieder dasselbe Chaos vor, weil alle Threads, wie es Ihnen passt, auf `cout` schreiben. Wenn Sie das ändern wollen, müssen Sie das Stream-Objekt `cout` schützen. Sie könnten dies bspw. mit einem Mutex wie folgt machen:

```
#include <mutex>
std::mutex m; // global definieren
...
m.lock(); // Exklusivrecht für einen Thread
// Hier die Ausgabe auf std::cout machen
m.unlock() // Exklusivrecht wieder aufheben
```

Wenn Sie jetzt in jedem Thread die Ausgabe zwischen `m.lock()` und `m.unlock()` schreiben, steht dem Thread, der als erstes `m.lock()` setzen kann, für die Ausgabe das Stream-Objekt `cout` zur Verfügung. Alle anderen Threads müssen dann allerdings warten, bis der Thread die gemeinsame Ressource wieder mit `m.unlock()` freigibt.

Für einfache Dinge, wie eben im Beispiel gesehen, sind solche Mutexe recht praktisch, aber die Gefahr ist doch recht groß, dass man bei intensiver Anwendung von einem Mutex mal vergisst, diesen wieder zu freigeben (`unlock()`). In dem Fall würden andere Threads, die auf diese Ressource warten, nicht mehr ausgeführt werden, weil das Programm jetzt komplett blockiert würde. Für solche Zwecke bietet die C++11-Standardbibliothek Besseres. Aber dazu später ab Abschnitt 15.3, »Gemeinsame Daten synchronisieren«, mehr.

Parameterübergabe als Referenz

Wenn Sie trotzdem versuchen, die Parameter als Referenz einem Thread mitzugeben, werden Sie feststellen, dass es gar nicht funktioniert. Entweder der Compiler verweigert die Übersetzung oder der Parameter wird trotzdem als Kopie übergeben. Betrachten Sie folgendes Beispiel:

```
...
01 void func(std::string& msg) {
02     std::cout << msg << std::endl;
03     msg.assign("Ein neuer String");
04 }
...
05 std::string s("Ein einfacher String");
06 std::thread t1(func, s);
07 t1.join();
08 std::cout << s << std::endl;
...
```

Hier wurde versucht, den String `s` als Referenz an den Thread `func()` mitzugeben. In Zeile 03 wird der String verändert, was sich ja bei einer Referenz auch beim Original auswirkt und daher in Zeile 08 auch ausgegeben würde. Allerdings hat sich die Zeile des Strings nicht geändert, obwohl dieser als Referenz übergeben wurde. Bei einigen Systemen lässt sich das Listing ohnehin nicht übersetzen.

Müssen Sie trotzdem unbedingt entgegen aller Ratschläge den Wert als Referenz übergeben (weil es bspw. eine Schnittstelle so verlangt), müssen Sie das Argument mit `std::ref()` wie folgt übergeben:

```
06 std::thread t1(func, std::ref(s));
```

Mit `std::ref()` teilen sich jetzt der Elternthread und der Kinderthread gemeinsam den Speicher des Strings `s` (auch *Shared Memory* genannt).

Parameterübergabe mit `std::move`

Wie bereits erwähnt, kann ein gemeinsames Speicherobjekt (Shared Memory) zwischen den Threads zu schwerwiegenden Fehlern führen, wenn sich der Erzeugerthread, welcher das Speicherobjekt an den Kinderthread übergibt, vorher beendet. Bisher wurden Sie im Buch immer darauf hingewiesen, die Übergabe als Kopie sei ebenfalls ineffizient, was gerade bei umfangreichen Speicherobjekten der Fall ist. Hier bietet sich aber dann das Verschieben mit dem neuen `std::move()` an. Verschieben ist zum einen sicher und zum anderen sehr effizient. Hierzu jetzt dasselbe Beispiel mit `std::move()`:

```
...
01 void func(std::string msg) {
02     std::cout << msg << std::endl;
03     msg.assign("Ein neuer String");
04 }
...
05 std::string s("Ein einfacher String");
06 std::thread t1(func, std::move(s));
07 t1.join();
08 std::cout << s << std::endl; // Ist jetzt leer
...
```

Hier verschieben wir jetzt in Zeile **06** den Inhalt des Strings `s` an den Thread in den Zeilen **01** bis **04**. Die Zuweisung mit `assign()` in Zeile **03** hat allerdings keinen Effekt mehr auf den String des Elternthreads in Zeile **05**. Durch das Verschieben sollte allerdings jetzt auch klar sein, dass der String `s` in Zeile **05** leer ist, was die Ausgabe in Zeile **08** auch beweist.

Einen Thread einem anderen Thread zuweisen

Ein Thread kann zwar nicht mit einer Zuweisung wie `th2=th1` zugewiesen und kopiert werden, aber mit `std::move()` haben Sie auch hier die Möglichkeit, den Thread zu verschieben. Wenn Sie den Eigentümer eines Threads wechseln wollen, haben Sie die Möglichkeit, dies mit `th2=std::move(th1)` zu tun. Anschließend sollten Sie sich allerdings noch um die Lebenszeit von `th2` via `join()` oder `detach()` kümmern.

15.2.2 Methoden für Threads

Bisher wurde noch kaum auf die Methoden von `std::thread` eingegangen, womit Sie die Threads kontrollieren oder wichtige Informationen ermitteln können.

Auf Thread warten oder loslösen

Bisher haben Sie in der `main()`-Funktion, was in diesem Fall der Elternthread oder auch Hauptthread ist, immer mit der Methode `join()` sichergestellt, dass der Hauptthread (oder auch Erzeuger des Threads) den erzeugten Kinderthread überlebt (Abbildung 15.5). Es ist wichtig, dass Sie verstehen, dass beim Erzeugen eines Threads auch der Erzeuger (Hauptthread) parallel weiterläuft. Daher ist es wichtig, dass Sie sich darum kümmern, was mit dem Erzeuger des Kinderthreads passieren soll. Überlassen Sie einen Thread komplett sich selbst, wird der Thread recht unsanft und unordentlich beendet. Es sollte noch angemerkt werden, dass Sie in jeder beliebigen Funktion ebenfalls neue Threads erzeugen und starten können und der Erzeugerthread nicht zwangsläufig in `main()` sein muss.

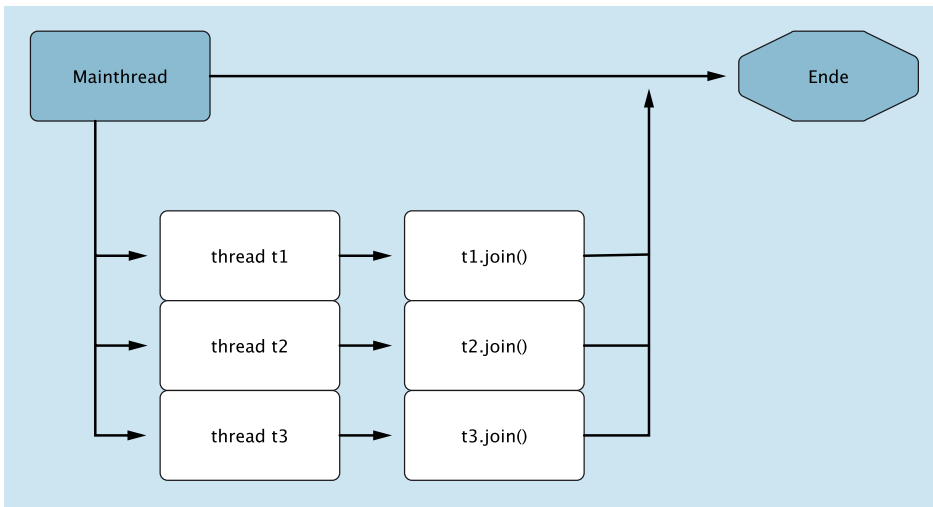


Abbildung 15.5 In der Praxis wird häufig auf das Ende aller Kinderthreads gewartet. Erst wenn diese beendet wurden, wird auch der Hauptthread beendet.

Eine zweite Möglichkeit, mit der Sie Einfluss auf die Lebenszeit des Threads nehmen, ist die Methode `detach()`. Die Methode `detach()` ist das Gegenstück zu `join()` und trennt den Kinderthread vom Erzeuger, so dass hiermit der Kinderthread seinen Erzeuger überleben kann. Im Gegensatz zu `join()` wird der Aufrufer damit nicht mehr blockiert, auf das Ende des Kinderthreads zu warten. Ein solcher losgelöster Thread wird unabhängig ausgeführt und lässt sich nach dem Prinzip »fire and forget« einsetzen.

Ein kurzer Codeausschnitt dazu:

```
01 // kapitel15/thread03.cpp
02 #include <thread>
...
03 std::thread t1([] {
04     std::this_thread::sleep_for(std::chrono::seconds(3));
05     std::cout << "Thread als Lambda-Funktion\n";
06 });
07 t1.detach();
08 std::cout << "Ende Hauptthread\n";
...
```

In diesem Beispiel lösen wir in Zeile **07** den Thread `t1` vom Hauptthread. Da wir hier außerdem den Kinderthread für 3 Sekunden schlafen legen, wird die Ausgabe in Zeile **08** des Hauptthreads zwar noch gemacht, aber die Ausgabe von Zeile **05** im Kinderthread werden Sie nicht mehr sehen, weil sich der Hauptthread bis dahin längst beendet hat und Sie den Kinderthread in Zeile **07** vom Hauptthread gelöst haben. Der Thread `t1` ist jetzt auf sich alleine gestellt. Wollen Sie wieder auf das Ende von `t1` warten, müssen Sie lediglich in Zeile **07** die Methode `detach()` wieder durch `join()` ersetzen.

Beenden Sie Threads kontrolliert

Die Methoden `join()` und `detach()` sind die einzigen Möglichkeiten, womit Sie Einfluss auf die Lebenszeit des Threads haben. Kümmern Sie sich nicht um das Ende der Threads mittels `join()` oder `detach()`, werden die noch laufenden Threads recht unsanft mit `std::terminate()` beendet, sobald sich der Erzeuger beendet. So etwas müssen Sie unbedingt vermeiden.

Informationen zum Thread

In der Praxis können Sie einen Thread nur ein einziges Mal mit `join()` oder mit `detach()` verwenden. Es ist also nicht mehr möglich, einen Thread erneut mit `join()` oder mit `detach()` zu verwenden. Würden Sie bspw. einen Thread erneut mit `join()` oder `detach()` aufrufen, würde das Programm abstürzen. Um zu prüfen, ob ein Thread mit `join()` oder `detach()` aufgerufen werden kann, können Sie die Methode `joinable()` verwenden. Diese liefert `true` zurück, wenn ein Thread `join()` oder `detach()` unterstützt. Wenn ein Thread bereits einmal `join()` oder `detach()` aufgerufen hat, wird ein künftiges Überprüfen auf `joinable()` immer `false` zurückgeben, da ein Thread eben nur einmal mit `join()` oder `detach()` verwendet werden kann.

Hierzu ein kurzes Beispiel:

```

01 // kapitel15/thread04.cpp
02 #include <thread>
...
03 std::thread t1([] {
04     std::this_thread::sleep_for(std::chrono::seconds(1));
05     std::cout << "Thread als Lambda-Funktion\n";
06 });
07 if( t1.joinable()) {
08     t1.join();
09 }
10 else {
11     std::cerr << "Kein join() oder detach() möglich\n";
12 }
13 if( t1.joinable()) {
14     t1.join(); // wird nie ausgeführt
15 }
...

```

In Zeile **07** überprüfen wird, ob auf den Thread `t1` die Methoden `join()` oder `detach()` anwenden können. Dies sollte hier `true` sein, und daher wird der Code in Zeile **08** ausgeführt. In dem Fall richten wir mit `join()` ein, dass unser Elternthread auf das Ende von Thread `t1` wartet. Das zweite `joinable()` in Zeile **13** wurde nur zu Demonstrationszwecken dort hingeschrieben. Damit stellen wir nur sicher, dass nicht aus Versehen ein zweites Mal `join()` oder `detach()` für einen Thread verwendet werden, was unweigerlich zum Absturz des Programms führen würde. Es ist daher eine gute Angewohnheit, vor einem `join()` oder `detach()` auf `joinable()` zu testen.

Eine weitere wichtige Information, welche man von einem laufenden Thread benötigt, ist der Identifier (kurz ID) für den Thread. Hierfür gibt es die Methode `get_id()`, die den eindeutigen Identifier für einen Thread zurückgibt. Benötigen Sie den Identifier innerhalb des Threads, müssen Sie die gleichnamige Funktion `this_thread::get_id()` dafür verwenden. Das folgende Beispiel zeigt die Identifier für Threads im Einsatz:

```

01 // kapitel15/thread05.cpp
02 #include <thread>
...
03 std::thread t1([] {
04     std::this_thread::sleep_for(std::chrono::seconds(2));
05     std::cout << "Thread " << std::this_thread::get_id()
               << " bei der Ausführung\n";
06 });

```

```

07 if( t1.joinable()) {
08     std::cout << "Mainthread: Warte auf "
                << t1.get_id() << '\n';
09     t1.join();
10 }
11 std::cout << "Mainthread: Thread ist fertig\n";
...

```

In Zeile 08 geben wir mit der Methode `get_id()` die ID des Threads aus, auf den unser Elternthread mit `join()` wartet. In Zeile 05 bringt uns die Methode `get_id()` im laufenden Thread nichts mehr. Hier können Sie dann die globale Funktion `this_thread::get_id()` verwenden. Die globale Funktion können Sie auch verwenden, wenn Sie die ID des Hauptthreads ermitteln wollen.

Das Programm bei der Ausführung:

```

Mainthread: Warte auf 0x1000ba000
Thread 0x1000ba000 bei der Ausführung
Mainthread: Thread ist fertig

```

Kinderthread abfangen

Nicht immer läuft alles nach Plan, und wenn sich der Elternthread vor dem Kinderthread beendet, ohne dass er sich vorher um die Lebenszeit des Kinderthreads mit `join()` oder `detach()` gekümmert hat, wird der Kinderthread mit `std::terminate()` abgeschossen und zerstört. Das folgende Beispiel soll das demonstrieren:

```

00 // kapitel15/thread06.cpp
01 #include <thread>
...
02 void start_threads() {
03     std::thread t1[] {
04         std::cout << "Thread bei der Ausführung\n";
05         std::this_thread::sleep_for(std::chrono::seconds(2));
06         std::cout << "Thread beendet\n";
07     });
08     for(int i=0; i<5; ++i) {
09         // Arbeiten des Elternthreads
10         // Ein Fehler tritt auf...
11         throw 1;
12     }
13     if(t1.joinable())
14         t1.join();
15 }

```

In der Funktion `start_threads()` wird in den Zeilen **03** bis **07** ein neuer Thread `t1` gestartet. Der Elternthread in den Zeilen **08** bis **12** führt ebenfalls eine Arbeit aus, wobei plötzlich in Zeile **11** eine Ausnahme geworfen wurde. Dadurch wird der Elternthread `start_threads()` beendet, bevor wir uns um den Kinderthread mit `join()` oder `detach()` in Zeile **13** kümmern konnten.

Das Programm bei der Ausführung:

```
libc++abi.dylib: terminate called without an active exception
Thread bei der Ausführung
(lldb)
```

Für solche Fälle müssen Sie sich mit einer Ausnahmebehandlung rüsten. Das folgende Listing wurde etwas umgeschrieben, damit jetzt auch auf das Ende des Kinderthreads gewartet wird:

```
00 // kapitel15/thread07.cpp
01 #include <thread>
...
02 void start_threads() {
03     std::thread t1([] {
04         std::cout << "Thread bei der Ausführung\n";
05         std::this_thread::sleep_for(std::chrono::seconds(2));
06         std::cout << "Thread beendet\n";
07     });
08     try {
09         for(int i=0; i<5; ++i) {
10             // Arbeiten des Elternthreads
11             // Ein Fehler tritt auf...
12             throw 1;
13         }
14     }
15     catch(...) {
16         if(t1.joinable())
17             t1.join();
18         throw;
19     }
20     if(t1.joinable())
21         t1.join();
22 }
...
```

Den kritischen Code haben wir jetzt in den `try`-Block in den Zeilen **08** bis **14** verpackt. Sollte sich der Elternthread hier jetzt tatsächlich vorzeitig beenden, fangen wir diese

Ausnahmen jetzt in den Zeilen 15 bis 19 ab, um ordnungsgemäß `join()` aufzurufen, wodurch der Kinderthread ordnungsgemäß bis zum Ende ausgeführt wird. In Zeile 18 werfen wir die Ausnahme weiter, in der Hoffnung, dass sich jemand darum im Aufrufer der Funktion `start_threads()` kümmert und die Ausnahme behandelt.

Das Programm bei der Ausführung:

Thread bei der Ausführung
Thread beendet

Übersicht zu den Methoden

In Tabelle 15.1 finden Sie einen kurzen Überblick über die vorhandenen Methoden von `std::thread`. Die meisten davon haben Sie bereits in den Abschnitten zuvor kennengelernt.

Methode	Beschreibung
<code>joinable()</code>	Prüft, ob ein Thread mit <code>join()</code> oder <code>detach()</code> verwendet werden kann. Gibt <code>true</code> zurück, wenn es möglich ist, ansonsten <code>false</code> .
<code>get_id()</code>	Gibt den Identifier für einen Thread zurück.
<code>join()</code>	Der Elternthread wartet auf den Thread, bis dieser seine Ausführung beendet hat.
<code>detach()</code>	Der Thread wird vom Elternthread losgelöst und unabhängig davon ausgeführt.
<code>native_handle()</code>	Damit kann ein Zugriff auf eine entsprechende Thread-Implementierung gemacht werden. Dies setzt allerdings voraus, dass Sie mit der Thread-Implementierung auf Ihrem System vertraut sind (bspw. sind POSIX-Threads weitverbreitet auf Linux-/Unix-Systemen). Hiermit können Sie dann Funktionen verwenden, welche von der C++11-Implementierung nicht angeboten werden.
<code>swap(std::thread)</code>	Tauscht zwei Thread-Objekte miteinander aus.

Tabelle 15.1 Methoden für »std::thread«

15.2.3 Funktionen für Threads

Zusätzlich gibt es noch einige globale Funktionen, welche Sie nur direkt im laufenden Thread (`this_thread`) verwenden können. Die Verwendung haben Sie bereits schon mit `this_thread::get_id()` im Beispiel von Listing *thread05.cpp* gesehen. In Tabelle 15.2 finden Sie einen Überblick über die Funktionen.

Funktion	Beschreibung
<code>std::this_thread::get_id()</code>	Gibt den Identifier für den laufenden Thread zurück.
<code>std::this_thread::yield()</code>	Damit geben Sie dem Betriebssystem die Möglichkeit, einen anderen Thread auszuführen.
<code>std::this_thread::sleep_for(abs_time)</code>	Blockiert den laufenden Thread für eine bestimmte Zeit.
<code>std::this_thread::sleep_until(rel_time)</code>	Blockiert den laufenden Thread bis zu einem bestimmten Zeitpunkt.
<code>std::this_thread::swap(th1, th2)</code>	Tauscht die Thread-Objekte <code>th1</code> und <code>th2</code> aus.

Tabelle 15.2 Funktionen für den laufenden Thread

15.2.4 Wie viele Threads sollen es sein?

Das ist in der Tat eine wichtige Frage, wenn Sie die Threads wirklich effektiv einsetzen wollen. Wenn Sie jetzt hergehen und einfach unzählige Threads starten und ausführen (wie dies in vielen Beispielen im Buch der Fall ist), dann führt das eher zu einer Überzeichnung (engl. *oversubscription*) von Threads. Eine solche Überzeichnung von Threads ist nicht so ideal, weil in diesem Fall mehr Threads ausgeführt werden, als CPUs zur Verfügung stehen. Und eine zu hohe Überzeichnung von Threads würde im Endeffekt wiederum nur die Performance drücken.

Für solche Zwecke bietet die Multithreading-Bibliothek die `static`-Funktion `std::thread::hardware_concurrency()` an, welche zurückgibt, wie viele gleichzeitig laufende Threads von der Implementation unterstützt werden. Hierbei wird entweder die Anzahl der Prozessoren zurückgegeben oder 0, wenn das System diese Funktionalität nicht implementiert hat. Diese Funktion ist besonders nützlich, weil Sie erst damit das Programm auf die Anzahl der tatsächlich vorhandenen CPUs anpassen können.

Thread == CPUs?

Eine zu hohe Überzeichnung findet übrigens noch nicht statt, nur wenn Sie mehr Threads ausführen, als CPUs vorhanden sind. Das Betriebssystem kümmert sich auch dann noch sehr schön um die Vergabe der Arbeit der einzelnen Threads, wenn Sie doppelt so viele Threads starten, als CPUs vorhanden sind. So etwas lässt sich hier nicht pauschalisieren. Wenn nur gerechnet wird, dann ist CPU-Anzahl gleich Thread-Anzahl recht gut, sofern man sicherstellen kann, dass alle Threads gleichzeitig fertig sind. Sind viele Ein-/Ausgabe-Operationen zu erledigen, dann können Sie gerne auch `CPUs*2` Threads starten.

15.3 Gemeinsame Daten synchronisieren

Wenn gemeinsame Daten mit mehreren Threads verwendet werden sollen, dann wird eine Technik benötigt, um die gemeinsam verwendeten Daten zu schützen. C++11 bietet hierfür verschiedene Techniken an, welche in den folgenden Abschnitten etwas genauer erläutert werden sollen. Als Negativ-Beispiel soll zunächst noch ein Listing geschrieben werden, in dem mehrere Threads auf dieselbe Variable zugreifen (Abbildung 15.6). Das Listing ist sehr primitiv und dient natürlich ausschließlich der Anschauung:

```

01 // kapitel15/thread08.cpp
02 #include <thread>
...
03 class Working_class {
04     int var;
05 public:
06     Working_class() : var(0) {}
07     void run() {
08         var++;
09         if(var == std::numeric_limits<int>::max()) {
10             throw -1; // Beim Überlauf von int werfen
11         }
12     }
13     int getvar() const { return var; }
14 };

15 Working_class wc;

16 void calc() {
17     for(auto i=0; i<100000; ++i) {
18         wc.run();
19     }
20 }
...
21 std::vector<std::thread> th(5);
22 for(auto &t : th) {
23     t = std::thread(calc);
24 }
25 for(auto &t : th) {
26     t.join();
27 }
28 std::cout << wc.getvar() << std::endl;

```

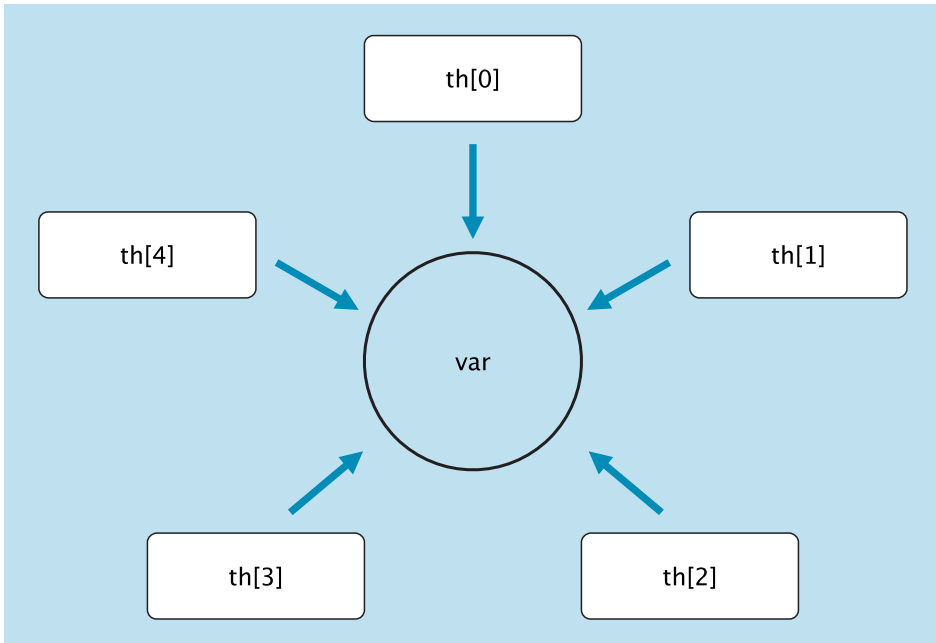


Abbildung 15.6 Ein kritischer Wettlauf mehrerer Threads um dieselben Daten kann zu schwer auffindbaren Fehlern im Programm führen.

In diesem Beispiel werden in den Zeilen 22 bis 24 fünf Threads in einer Schleife gestartet, welche alle die Funktion `calc` (Zeilen 16 bis 20) ausführen. Die Funktion `calc()` macht nichts anderes, als in einer Schleife 100.000x die Methode `run()` für das Objekt `wc` aufzurufen, welches wir in Zeile 15 global definiert haben. In der Klassendefinition in den Zeilen 03 bis 14 passiert nichts anderes, als dass die Methode `run()` die Eigenschaft `var` mit jedem Aufruf um 1 inkrementiert. Im Grunde ein primitives Listing, welches am Ende immer 500.000 in Zeile 28 ausgeben sollte, wenn alle Threads beendet wurden.

Sollte! Aber in diesem Fall ist es nicht immer der Fall. Es kommt vor, dass der Wert `Working_class::var` am Ende weniger als 500.000 ausgibt. Was wir hier haben, nennt man einen kritischen Wettlauf um die Daten `Working_class::var`. Da jetzt jeder Thread unkontrolliert darauf zugreift, kann das zu unauffindbaren Fehlern führen.

Race Condition

Es liegt keine Race Condition vor, wenn alle beteiligten Threads nur lesend auf die Ressource zugreifen. Erst bei mindestens einem Schreiber, kann es ein Race Condition geben.

Um eine solche *Race Condition* zu vermeiden, stehen Ihnen verschiedene Möglichkeiten zur Verfügung, die in den folgenden Abschnitten behandelt werden:

- ▶ **Mutex manuell setzen und wieder freigeben**
- ▶ **`std::lock_guard`**: Hier wird RAII verwendet, womit ein Scope Locking möglich ist. Sprich, der Lock wird mit dem Konstruktor gesetzt und beim Verlassen des Gültigkeitsbereichs wieder freigegeben.
- ▶ **`std::unique_lock`**: Ähnlich wie `std::lock_guard`, nur noch weitaus mächtiger mit vielen weiteren Methoden.

Weitere Möglichkeiten, eine Race Condition zu vermeiden, wäre hier noch die Condition-Variablen, aber diese werden erst zu einem späteren Zeitpunkt behandelt.

15.3.1 Schutz der Daten über einen Mutex

Die einfachste Möglichkeit, um gemeinsame Ressourcen zu schützen, ist der Mutex. Das Prinzip ist einfach. Jeder Thread, der die gemeinsamen Daten verwendet, muss zuerst einen Mutex setzen, bevor er den kritischen Codeausschnitt ausführen darf. Am Ende der Ausführung des kritischen Codeausschnitts muss der Mutex natürlich wieder freigegeben werden. Wenn ein Thread den geschützten Bereich verwenden will und der Mutex von einem anderen Thread bereits gesetzt wurde, wird der Thread so lange blockiert, bis der Mutex wieder freigegeben wird. Mithilfe eines Mutex kann also immer nur ein Thread auf einen kritischen Codebereich zugreifen.

Im oben verwendeten Beispiel ist ein solcher Mutex schnell integriert, wie das folgende Listing zeigen soll:

```
01 // kapitel15/thread09.cpp
02 #include <thread>
03 #include <mutex>
04 ...
05 std::mutex m;

06 class Working_class {
07     int var;
08 public:
09     Working_class() : var(0) {}
10     void run() {
11         m.lock();
12         var++;
13         m.unlock();
14         if(var == std::numeric_limits<int>::max()) {
15             throw -1; // Beim Überlauf von int werfen
```

```

15     }
16 }
17 int getvar() const { return var; }
18 };

```

Jeder Thread, der jetzt die Methode `run()` ausführt, setzt bei jeder Ausführung in Zeile 10 den Mutex, wodurch alle anderen Threads an der Stelle warten müssen. Jetzt kann der Thread, der den Mutex gesetzt hat, die Variable `var` in Zeile 11 inkrementieren. Am Ende gibt der Thread den Mutex in Zeile 12 wieder frei, so dass ein anderer Thread jetzt wieder den Mutex in Zeile 10 setzen kann, und alles geht wieder von vorne los. Hier haben wir keinen kritischen Wettlauf mehr um die Variable `var` und das Problem durch eine Synchronisation behoben (Abbildung 15.7).

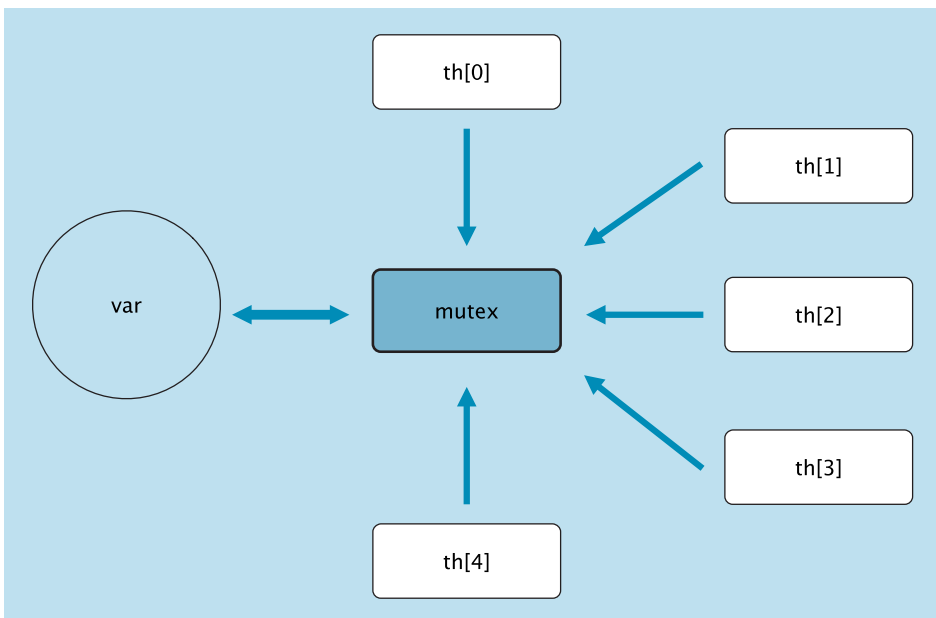


Abbildung 15.7 Mithilfe eines Mutex kann ein kritischer Wettlauf mehrerer Threads um die Daten im Zaum gehalten werden.

Verschiedene Mutexe

C++11 hat neben dem gewöhnlichen Mutex (`std::mutex`) verschiedene Varianten grundlegender Mutex-Funktionalitäten im Programm, um gemeinsame Daten zu schützen. Da ist etwa noch ein rekursiver Mutex mit `std::recursive_mutex`, welcher eine rekursive Semantik enthält. Was bedeutet hier **rekursiv**? Im Prinzip ganz einfach. Wenn ein Thread mehrfach von einem thread gelockt wurde (bspw. durch einen rekursiven Aufruf), so kann bei einem nicht rekursiven Mutex nicht gesagt werden,

was passiert. Ein rekursiver Mutex hingegen hat einen internen Zähler, damit dieser genauso oft auch wieder freigegeben wird, wie er gelockt wurde.

Nützlich ist auch ein Mutex mit einer relativen oder absoluten Zeitangabe, wie dieser mit `std::timed_mutex` angeboten wird. Zur Vervollständigung gibt es auch vom Mutex für die Zeitangabe ein rekursives Gegenstück mit `std::recursive_timed_mutex`. Für die Verwendung von Mutex mit einer **Zeitangabe** stehen Ihnen die Methoden `try_lock_for()` für eine absolute und `try_lock_until()` für eine relative Zeitangabe zur Verfügung. Bei der Verwendung einer Zeitangabe versucht der Mutex den Lock für die angegebene Zeit des Timeouts zu halten, bspw.:

```
...
01 #include <mutex>
02 #include <chrono>

03 std::timed_mutex m;
04 class Working_class {
...
05     void run() {
06         m.try_lock_for(std::chrono::milliseconds(100));
07         var++;
08         m.unlock();
...
09     }
10 };
...
```

C++14: `std::shared_mutex`

Für den nächsten Standard von C++ ist noch ein fünfter Mutex mit `std::shared_mutex` angedacht, mit dem es möglich werden wird, dass mehrere Threads zum Eigentümer desselben Mutex erklärt werden können – also ein gemeinsamer Mutex.

Methoden für Mutexe

Mit allen vorhandenen Mutex-Typen können Sie die bereits bekannten Methoden `lock()` zum Sperren eines Mutex und `unlock()` zum Freigeben verwenden. Zusätzlich bieten alle Mutex-Typen noch die Methode `try_lock()` an, womit ein Thread versucht, den Lock zu erhalten, aber auch nicht blockiert, wenn er keinen bekommt. Für die Mutex-Typen `std::timed_mutex` und `std::recursive_timed_mutex` gibt es zusätzlich noch die Methoden `try_lock_for()` und `try_lock_until()`, um eine absolute bzw. relative Timeout-Angabe für einen Mutex beim »locken« zu verwenden.

native_handle

Sind Sie mit der Thread-Implementierung auf Ihrem System vertraut (bspw. POSIX-Thread), dann können Sie auch hier mit der Methode `native_handle()` nativ auf den Lock zugreifen.

Gefahren bei der Verwendung eines Mutex

Mutexe sind die wohl primitivste Form, mehrere Threads vor einem *Data Race* (kritischer Wettlauf um Daten) zu schützen. Trotzdem sind Mutexe nicht unbedingt optimal, um einen gemeinsamen Bereich zu schützen, weil man jedes Mal eine Sperre setzen und darauf achten muss, diese auch wieder freizugeben. Nehmen wir nochmals unseren Codeausschnitt unter der Methode `run()` unter die Lupe:

```
01 // kapitel15/thread09.cpp
...
02 void run() {
03     m.lock();
04     var++;
05     m.unlock();
06     if(var == std::numeric_limits<int>::max()) {
07         throw -1;
08     }
09 }
...
```

In diesem Beispiel prüfen wir nach der Freigabe des Locks, ob der Wert von `var` schon gleich dem maximalen Wert von `int` ist. Zugegeben, das hätte man hier auch viel besser machen können, aber es wurde absichtlich so gemacht, um auf das Problem mit dem Mutex hinzuweisen. Das Problem ist jetzt, dass sich die Anweisung hinter `unlock()` befindet, und hier jederzeit ein anderer Thread bereits die Variable `var` um 1 erhöhen **könnte**. Würde der Wert von `var` jetzt tatsächlich `std::numeric_limits<int>::max()` sein und vor der Überprüfung findet ein `var++` durch einen anderen Thread statt, dann gibt die Bedingung `false` zurück, weil jetzt `std::numeric_limits<int>::max()+1` ist, obwohl der Wert von `var` bereits überschritten wurde. Damit **könnte** quasi eine Wertebereichsüberschreitung von `int` *übersprungen* werden.

Okay, werden Sie sich sagen, dann bauen Sie einfach die Fehlerüberprüfung innerhalb des gelockten Bereichs zwischen `lock()` und `unlock()` ein:

```
01 // kapitel15/thread09.cpp
...
02 void run() {
```



```

03  m.lock();
04  var++;
05  if(var == std::numeric_limits<int>::max()) {
06      throw -1;
07  }
08  m.unlock();
09  }
...

```

Die Idee ist an sich nicht schlecht, nur müssen Sie jetzt allerdings sicherstellen, dass sich die Ausnahmebehandlung, welche den Fehler abfängt, um den Mutex kümmert und den Lock wieder mit `unlock()` freigibt. Machen Sie dies nicht, haben Sie Ihr Programm »totgesperrt« (*Deadlock*). Alle anderen Threads warten jetzt darauf, dass der Thread, welcher die Ausnahmen ausgelöst hat, den Mutex wieder freigibt. Hierzu noch das Beispiel, wie dies nun mit einer Ausnahmebehandlung aussehen sollte/könnte:

```

00  // kapitel15/thread10.cpp
...
01  void run() {
02      m.lock();
03      try {
04          var++;
05          if(var == std::numeric_limits<int>::max()) {
06              throw -1;
07          }
08          m.unlock();
09      }
10      catch(...) {
11          m.unlock();
12          throw;
13      }
14  }
...

```

Zugegeben, das Problem war hier doch sehr hausgemacht. Aber es kann jederzeit vorkommen, dass innerhalb eines kritischen Codes, der mit einem `lock()` gesichert wurde, eine Ausnahme geworfen wird, auf die man vielleicht gar nicht vorbereitet war. Und das Aufheben eines Mutex an einer ganz anderen Stelle im Code wird dann doch gerne mal vergessen, oder es wird vielleicht gar nicht daran gedacht, dass eine unerwartete Ausnahme eintreten könnte. Und wenn Sie mal angefangen haben, an sehr unterschiedlichen Stellen im Code einen Mutex mit `lock()` zu setzen und mit `unlock()` wieder freizugeben, wird es schnell chaotisch.

Verzichten Sie auf die direkte Verwendung von Mutexen

Wenn Sie den Mutex unbedingt direkt verwenden wollen, dann müssen Sie immer selbst sicherstellen, dass Sie den Lock wieder freigeben. Daher sollten Sie solche Bereiche immer nur im Zusammenhang mit einer Ausnahmebehandlung verwenden. Als bessere Alternative zur direkten Verwendung von Mutexen würden sich fertige Locks wie `lock_guard` oder `unique_lock` anbieten, welche im anschließenden Abschnitt erläutert werden.

15.3.2 Schutz der Daten über Locks

Um in der Praxis die eben beschriebenen Probleme mit einem Mutex bei den Threads zu vermeiden, wird allgemein empfohlen, diese nicht direkt zu verwenden, sondern sogenannte Locks. C++11 bietet für diese Zwecke mit `std::lock_guard` und `std::unique_lock` zwei Versionen von Locks an.

`std::lock_guard`

Hier unser bekanntes Beispiel mit `std::lock_guard` im Einsatz:

```
00 // kapitel15/thread11.cpp
...
01 std::mutex m;

02 class Working_class {
03     int var;
04 public:
05     Working_class() : var(0) {}
06     void run() {
07         std::lock_guard<std::mutex> guard(m);
08         var++;
09         if(var == std::numeric_limits<int>::max()) {
10             throw -1;
11         }
12     }
13     int getvar() const { return var; }
14 };
```

In Zeile **07** ruft `std::lock_guard` hier intern `m.lock()` auf und sperrt diesen Codebereich. Der Destruktor von `std::lock_guard` zum Aufheben der Sperren (mit `m.unlock()`) wird automatisch aufgerufen, wenn der Gültigkeitsbereich verlassen wird – jetzt auch, wenn eine Ausnahme geworfen wird. In diesem Beispiel wäre der

Gültigkeitsbereich für den Lock von den Zeilen 06 bis 12. Hier kommt das klassische RAIL zum Einsatz – hierbei spricht man auch vom *Scope Locking*. Ansonsten lässt sich nicht viel mehr zu diesem Lock sagen. Dieser bietet auch keine weiteren Funktionen an. Aber das Prinzip ist simpel: Der Konstruktor lockt den Mutex, und der Destruktor gibt diesen wieder frei.

Deadlock – nichts geht mehr

Nachdem Sie jetzt wissen, wie Sie mit Race Condition umgehen können, müssen Sie sich noch mit einem anderen Problem herumschlagen, das bei den Threads auftreten kann, wenn Sie gemeinsame Daten synchronisieren wollen. Und zwar geht es um einen Deadlock. Ein solcher Deadlock sperrt ein Programm so, dass gar nichts mehr geht. Das kann passieren, wenn mindestens zwei Threads auf eine Freigabe einer Ressource des anderen Threads warten. Der folgende Quellcode soll das *Worst-Case-Szenario* demonstrieren, in dem zwei Locks verwendet werden:

```
00 // kapitel15/thread12.cpp
...
01 class Work {
02     int var;
03 public:
04     std::mutex m;
05     Work() : var(0){}
06 };

07 void function(Work& w1, Work& w2) {
08     std::lock_guard<std::mutex> guard1(w1.m);
09     std::cout << "Work w1 ist aktiv\n";
10     //std::this_thread::sleep_for(
11         //std::chrono::milliseconds(1000));
12     std::lock_guard<std::mutex> guard2(w2.m);
13     std::cout << "Work w2 ist aktiv\n";
14 }
...
14 Work w1, w2;
15 std::thread th01( [&] {function(w1, w2);} );
16 std::thread th02( [&] {function(w2, w1);} );
17 th01.join();
18 th02.join();
...
```

Den Mutex finden Sie in der Klasse als öffentliches Mitglied in Zeile 04. In den Zeilen 08 und 11 werden in der Funktion zwei »Ressourcen« gelockt – auf den ersten Blick ein normaler Vorgang. Aber wenn die Laufzeit ungünstig ist, kann es passieren, dass

sich die beiden Threads, welche in den Zeilen 15 und 16 gestartet wurden, gegenseitig blockieren, indem jeder Thread auf den ersten Lock wartet (Deadlock). Damit es in diesem Beispiel auch auf Anhieb mit dem Deadlock »klappt«, simulieren wir die Arbeit des Threads mit einer Pause und halten diesen in Zeile 10 kurz an.

Das Programm bei der Ausführung:

```
Work w1 ist aktiv
Work w1 ist aktiv
```

Hier warten jetzt beide Threads auf die Freigabe des jeweils anderen Locks.

Auch hier ist das Problem natürlich wieder hausgemacht, und wenn Sie beim Starten des Threads die Reihenfolge betrachten, könnten Sie das Problem ganz einfach umgehen, indem Sie beide Locks in der gleichen Reihenfolge setzen. In unserem Beispiel würde es daher bereits ausreichen, wenn Sie lediglich beim Starten der beiden Threads beide Locks in der gleichen Reihenfolge starten. Also:

```
15 std::thread th01( [&] {function(w1, w2);} );
16 std::thread th02( [&] {function(w1, w2);} );
```

Aber in der Praxis ist es natürlich nicht immer so offensichtlich, und wir wollen unsere alte Ausgangssituation mit dem Deadlock nochmals betrachten. Neben der Möglichkeit, die Reihenfolge der Locks zu beachten, haben Sie noch die Möglichkeit, mehrere Locks oder Mutexe *atomar* (sozusagen »pseudo-gleichzeitig«) zu setzen.

std::unique_lock

Um unseren Threads aus dem Beispiel zuvor jetzt etwas nachzuhelfen und den Deadlock zu beseitigen, können Sie `std::unique_lock` verwenden. Folgender Codeausschnitt behebt jetzt unser Deadlock-Problem:

```
00 // kapitel15/thread13.cpp
...
01 void function(Work& w1, Work& w2) {
02     std::unique_lock<std::mutex> guard1(
        w1.m, std::defer_lock);
03     std::cout << "Work w1 ist aktiv\n";
04     //std::this_thread::sleep_for(
        //std::chrono::milliseconds(1000));
05     std::unique_lock<std::mutex> guard2(
        w2.m, std::defer_lock);
06     std::cout << "Work w2 ist aktiv\n";
07     std::lock(guard1, guard2);
08 }
...
```

In den Zeilen 02 und 05 verwenden wir jetzt das wesentlich mächtigere `std::unique_lock` statt `std::lock_guard`. Wir verwenden hier `std::unique_lock`, weil uns dieser einen verzögerten Mutex mit der Option `std::defer_lock` anbietet. Diese Option ist nötig, damit wir anschließend in Zeile 07 mit dem Funktions-Template `std::lock()` mehrere Mutexe gemeinsam und atomar locken. Im vorliegenden Fall bedeutet das, entweder hat der aufrufende Thread beide Locks oder keinen.

std::lock()

Neben dem Funktions-Template `std::lock()` finden Sie hier auch noch das Funktions-Template `std::try_lock()`. Die Anzahl der Argumente dieses Funktions-Templates sind allerdings nicht nur auf zwei Locks beschränkt. Da es sich hier um Variadic Templates handelt, können auch noch mehr als nur zwei Locks gleichzeitig atomar binden.

`std::unique_lock` ist allerdings noch weitaus mächtiger. Damit können Sie bspw. eine Instanz ohne einen assoziierten Mutex erzeugen. Ebenso können Sie eine Instanz mit einem Mutex erzeugen, der nicht gleich automatisch gelockt ist. Des Weiteren bietet `std::unique_lock` eine Menge Methoden an, um (versuchsweise) zu locken, das (versuchsweise) Locken auf Zeit, das Tauschen von Locks, die Freigabe von Locks, die Rückgabe von Locks usw.

Genau genommen ist `std::unique_lock` ein Wrapper um einen Mutex und sollte dann verwendet werden, wenn `std::lock_guard` nicht mehr ausreicht. Obwohl `std::unique_lock` ein Wrapper um einen Mutex ist, verwendet es ebenfalls das klassische RAII. Daher wird hier auch beim Verlassen des Gültigkeitsbereichs automatisch der Mutex durch den Destruktor wieder freigegeben.

So können Sie mit `std::unique_lock`, im Gegensatz zum `std::lock_guard`, jederzeit einen Lock mit `unlock()` wieder freigeben, wenn Sie diesen nicht mehr benötigen. Hier müssen Sie nicht bis zum Ende des Gültigkeitsbereichs warten. Ein einfaches Beispiel:

```
00 // kapitel15/thread14.cpp
...
01 void function(Work& w1, Work& w2) {
02     std::unique_lock<std::mutex> guard1(w1.m);
03     std::cout << "Work w1 ist aktiv\n";
04     std::this_thread::sleep_for(std::chrono::milliseconds(1000));
05     guard1.unlock();
06     std::unique_lock<std::mutex> guard2(w2.m);
07     std::cout << "Work w2 ist aktiv\n";
08 }
```

Hier geben Sie in Zeile **05** den Mutex manuell frei. Der zweite Lock in Zeile **06** hingegen wird freigegeben, wenn der Gültigkeitsbereich verlassen wurde. Hier können Sie aber auch jederzeit manuell eingreifen.

std::unique_lock verschieben

Ein `std::unique_lock` können Sie zwar nicht kopieren, aber auch hier können Sie jederzeit einen Lock mit `std::move` verschieben.

Sie können den Lock mit `std::unique_lock` auch so einrichten, dass dieser nicht sofort gelockt wird. Sie haben bereits die Option `std::defer_lock` kennengelernt, als es darum ging, einen Deadlock zu vermeiden. Mit der Option haben Sie die Möglichkeit, den Lock selbst manuell mit `lock()` und `unlock()` zu setzen und wieder freizugeben. Das Beispiel dazu:

```
00 // kapitel15/thread15.cpp
...
01 void function(Work& w1, Work& w2) {
02     std::unique_lock<std::mutex> guard1(w1.m, std::defer_lock);
03     // tue hier noch irgendwas anderes ...
04     guard1.lock();
05     std::cout << "Work w1 ist aktiv\n";
06     std::this_thread::sleep_for(std::chrono::milliseconds(1000));
07     guard1.unlock();
08     std::unique_lock<std::mutex> guard2(w2.m, std::defer_lock);
09     // auch hier können wir noch was anderes tun
10     guard2.lock();
11     std::cout << "Work w2 ist aktiv\n";
12     guard2.unlock();
13 }
```

In den Zeilen **02** und **08** verwenden wir jeweils einen `std::unique_lock`, der noch nicht gesperrt wird. Hier können Sie praktisch noch vorher, wie in den Zeilen **03** und **09** als Kommentar angegeben wurde, Arbeiten ausführen, solange der Lock noch nicht gesetzt wurde. Den Lock setzen Sie hier erst in Zeile **04** bzw. **10** manuell und geben diesen auch in den Zeilen **07** und **12** manuell wieder frei. Geben Sie den Lock nicht explizit frei, dann geschieht dass trotzdem wieder am Ende des Gültigkeitsbereichs. Im Fall von `guard1.unlock()` in Zeile **07** ist das Weglassen aber als kritisch zu betrachten, weil hier wieder die Gefahr eines Deadlocks droht. Sie können übrigens den Lock durchaus mehrmals setzen und wieder freigeben.

std::unique_lock vs. std::lock_guard

Wie Sie in den letzten Beispielen sehen konnten, ist `std::unique_lock` extrem flexibel und vielseitig. Da stellt sich natürlich die Frage, wozu es überhaupt einen `std::lock_guard` braucht? Hier muss natürlich darauf hingewiesen werden, dass eine solche Flexibilität wie `std::unique_lock` auch mehr Speicher braucht. `std::unique_lock` benötigt somit mehr Rechenaufwand als `std::lock_guard`. Daher gilt: Wenn Sie die Flexibilität von `std::unique_lock` nicht unbedingt benötigen, sollten Sie `std::lock_guard` verwenden. Benötigen Sie Locks für komplexere Sachen, dann greifen Sie auf `std::unique_lock` zurück.

Deadlocks vermeiden

Hier ließen sich noch weitere Techniken beschreiben, wie Sie Deadlocks vermeiden können. Um das Thema hier nicht allzu umfassend darzustellen, sollen kurz ein paar Punkte zusammengefasst werden, wie Sie Deadlocks vermeiden können:

1. Der einfachste Punkt dürfte wohl die Frage sein, ob es unbedingt zwei Locks gleichzeitig sein müssen? Es ist im Grunde ja kein Problem, zwei Locks in einer Funktion zu verwenden, aber meistens sind gar keine zwei Locks im gleichen **Gültigkeitsbereich** nötig. Der folgende Codeausschnitt zeigt, wie dies gemeint ist:

```
01 void function(Work& w1, Work& w2) {
02     {
03         std::lock_guard<std::mutex> guard1(w1.m);
04         // Arbeiten ausführen
05     } // ab hier ist Mutex wieder frei
06     { // und kann ab hier wieder gesetzt werden
07         std::lock_guard<std::mutex> guard2(w2.m);
08         // Arbeiten ausführen
09     } // hier wird der Mutex wieder freigegeben
10 }
```

2. Vermeiden Sie es, wenn Sie einen Lock gesetzt haben, eine eigene definierte globale Funktion aufzurufen. Häufig wird hierbei versucht, einen weiteren Mutex oder sogar denselben Mutex erneut zu locken. Führen Sie daher stattdessen die Funktion vorher aus, bevor Sie den Mutex setzen.
3. Wenn Sie wirklich zwei Locks gleichzeitig benötigen, verwenden Sie `std::lock()` dafür, weil dieses Funktions-Template atomare Funktionalitäten anbietet, womit sich Deadlocks vermeiden lassen.
4. Häufig geht es noch einfacher, indem Sie einfach die Locks für jeden Thread in derselben Reihenfolge setzen.

15.3.3 Daten sicher initialisieren

Mutexe und Locks sind tolle Schnittstellen, um auf gemeinsame Ressourcen sicher zuzugreifen und um Race Condition zu vermeiden. Aber es gibt Fälle, wo ein Mutex oder eben ein Lock zu aufwendig sind. Der folgende Codeausschnitt zeigt einen solchen Fall:

```
00 // kapitel15/thread16.cpp
...
01 class Log {
02     std::mutex m1, m2;
03     std::ofstream file;
04 public:
05     Log(){}
06     void log_file(std::string s, int val) {
07         {
08             std::unique_lock<std::mutex> lock1(m1);
09             if( !file.is_open()) {
10                 file.open("Logfile.txt");
11             }
12         }
13         std::unique_lock<std::mutex> lock2(m2);
14         file << s << ":" << val << std::endl;
15     }
16 };
...
17 Log lfile;
18 std::thread th01( [&lfile] {
19     for(int i=0; i<50; ++i) {
20         lfile.log_file(std::string("Lambda1"), i);
21     }
22 });
23 std::thread th02( [&lfile] {
24     for(int i=0; i<50; ++i) {
25         lfile.log_file(std::string("Lambda2"), i);
26     }
27 });
28 th01.join();
29 th02.join();
...
```

Dieses Beispiel wurde mithilfe von Locks absolut threadsicher geschrieben. In den Zeilen 18 bis 22 und 23 bis 27 wird jeweils ein Thread gestartet, wo jeder Daten in ein gemeinsames Logfile schreibt. In der Methode `log_file()` haben wir die Methoden

`is_open()` und `open()` vor anderen Threads mit einem Lock synchronisiert, indem wir diesen Code in einen eigenen Gültigkeitsbereich gesteckt haben. Für das Prüfen und Testen, ob eine Datei bereits offen ist, und ein gegebenenfalls nötiges Öffnen der Datei verwenden wir einen anderen Mutex, als für den Teil, in dem in die Datei geschrieben wird. Soweit ist das Beispiel recht ordentlich und eben threadsicher.

Trotzdem sollte der Codeausschnitt in den Zeilen 07 bis 12 noch etwas genauer untersucht werden. Jeder Thread setzt jetzt beim Aufruf dieser Methode den Lock, überprüft, ob die Datei offen ist, und öffnet diese dann, wenn es nötig ist. Wenn viele Threads gleichzeitig diesen Vorgang abarbeiten, ist das im Grunde eine totale Verschwendung von Rechenleistung. Schlimmer noch, dieser Lock in Zeile 08 verhindert eigentlich, dass das Programm auch wirklich simultan abläuft. Wenn man es genau überlegt, würde es eigentlich ausreichen, wenn die Datei nur ein einziges Mal geöffnet wird.

std::call_once und std::once_flag

Für das eben beschriebene Problem liefert die C++11-Standardbibliothek das neue Funktions-Template `std::call_once()`, womit eine Funktion genau einmal von einem Thread aufgerufen wird. Hierzu wird neben der aufrufbaren Einheit (gegebenenfalls mit beliebig vielen Argumenten) noch das Flag `std::once_flag` benötigt. Hierzu nun unsere Klasse `Log` mit dem Funktions-Template `std::call_once()`:

```
00 // kapitel15/thread17.cpp
...
01 class Log {
02     std::mutex m2;
03     std::once_flag flag;
04     std::ofstream file;
05 public:
06     Log(){}
07     void log_file(std::string s, int val) {
08         std::call_once(flag, [&]() {file.open("Logfile.txt");});
09         std::unique_lock<std::mutex> lock2(m2);
10         file << s << ":" << val << std::endl;
11     }
12 };
...
```

Der Aufruf von `std::call_once` in Zeile 08 bekommt als erstes Argument das Flag vom Typ `std::call_flag` und eine aufrufbare Einheit, hier in Form einer Lambda-Funktion, wo die Datei jetzt wirklich nur ein einziges Mal geöffnet wird.

Der Vorteil bei `std::call_once` ist, dass hier die eigentlichen Arbeiten verborgen bleiben. Sie müssen nur wissen, dass hiermit eine aufrufbare Einheit ein einziges Mal aufgerufen wird. Für die korrekte Synchronisation sorgen dann im inneren das Flag `std::once_flag` zusammen mit dem Funktions-Template `std::call_once`.

`std::call_once` ist ein Variadic Template

Da `std::call_once` ebenfalls ein Variadic Template ist, können damit beliebig viele Argumente verwendet werden, um die aufzurufende Funktion mit Parametern zu versehen.

Anstatt, wie hier, den Zugriff auf eine Datei zu synchronisieren, wird diese Technik auch gerne verwendet, wenn ein Thread dynamisch Speicher zur Laufzeit anfordert und alle anderen Threads auch diesen Speicher verwenden sollen. Auch hier soll die Speicherreservierung dann nur einmalig ausgeführt werden.

15.3.4 Statische Variablen

Nicht immer muss eine threadsichere Initialisierung über `std::call_once` erfolgen. In manchen Fällen lässt sich auch eine einfache statische Variable im Blockbereich verwenden. Das folgende Beispiel demonstriert den Vorgang relativ vereinfacht:

```
00 // kapitel15/thread18.cpp
...
01 class Data{
02     int val;
03 public:
04     Data(int v=0):val(v){}
05     int getVal() const { return val; }
06 };

07 void function(int ival) {
08     static Data dat(ival);
09     std::cout << dat.getVal() << std::endl;
10 }
...
11 Data dat;
12 std::thread th01( function, 12345);
13 std::this_thread::sleep_for(
14     std::chrono::milliseconds(1000));
14 std::thread th02( function, 54321 );
15 th01.join();
16 th02.join();
```

In Zeile **08** schreiben wir einfach `static` vor dem Objekt bei der Initialisierung. Nach der ersten Initialisierung des ersten Threads in Zeile **12** erhält das Objekt seinen Wert mit 12345 zugewiesen. Dieser Wert bleibt uns jetzt zu Lebzeiten in diesem Blockbereich erhalten, egal, ob weitere Threads den Wert nachträglich ändern wollen, wie in Zeile **14** versucht wird. Auch der Thread `th02`, der in Zeile **14** gestartet wird, muss jetzt mit dem Wert 12345 arbeiten, wie jeweils die Ausgabe in Zeile **09** bestätigt. In beiden Fällen wird hier der Wert 12345 ausgegeben. Würde hier allerdings `th02` vor `th01` gestartet, wäre der Wert immer 54321. das ist somit ein einfaches implementiertes *Singleton* und wird manchmal auch *Meyers Singleton* genannt.

Threadsicher

Die Möglichkeit, eine statische Variable in einem Blockbereich zu verwenden, ist im C++11-Standard *threadsicher*, nicht aber in der Vorgängerversion. Wenn Sie allerdings ohnehin das neue C++11-Multithreading verwenden, dann kann Ihnen das im Grunde auch egal sein. Trotzdem sollte das hier erwähnt werden.

15.4 Threadlokale Daten

Wollen Sie gar keine gemeinsamen Daten in Threads verwenden, sondern wollen Sie, dass jeder Thread seine eigenen Daten hat, dann werden Sie bei dem Schlüsselwort `thread_local` fündig, das Sie vor der entsprechenden Variablen schreiben müssen. Hierbei handelt es sich dann um eine statische Variable für den Thread, welche bei der ersten Verwendung initialisiert wird und sich zur Lebenszeit des Threads nicht mehr ändert.

Speicherklassen-Spezifizierer `thread_local`

Das Schlüsselwort `thread_local` ist ein echter Speicherklassen-Spezifizierer neben Speicherklassen wie `register`, `static` und `extern`, es wurde neu mit C++11 eingeführt.

Die Speicherklasse lässt sich relativ einfach demonstrieren. Zunächst ein Beispiel ohne diese Speicherklasse:

```
00 // kapitel15/thread19.cpp
...
01 unsigned int val=0;

02 void run(std::string msg) {
03     for(unsigned int i=0; i<5; ++i)
04         ++val;
```

```

05  std::cout << msg << val << std::endl;
06  }
...
07  std::thread th01(run, "Thread 1: ");
08  th01.join();
09  std::thread th02(run, "Thread 2: ");
10  th02.join();
11  std::cout << "main : " << val << std::endl;
...

```

Die Ausgabe des Programms sollte an dieser Stelle nicht überraschend sein, da alle auf die globale Variable `val` aus Zeile 01 zugreifen:

```

Thread 1: 5
Thread 2: 10
main : 10

```

Schreiben Sie jetzt den Speicherklassen-Spezifizierer `thread_local` in Zeile 01 wie folgt hin:

```

01  thread_local unsigned int val=0;

```

Da jetzt jeder Thread ein eigene Variable `val` besitzt, sieht die Ausgabe entsprechend anders aus:

```

Thread 1: 5
Thread 2: 5
main : 0

```

15.5 Threads synchronisieren

Mit Mutexen und Locks kennen Sie bereits sehr praktische Techniken, um Threads zu synchronisieren. Manchmal will man aber auch einfach einzelne Threads nach Ereignissen steuern. Die Rede ist hier von einem Thread als Sender und einem anderen Thread als Empfänger. Es ist zwar möglich, auch dieses Problem mit einem Lock zu lösen, aber doch ziemlich ineffizient. Das folgende Listing soll diesen Vorgang mit unseren bisherigen Kenntnissen demonstrieren:

```

00  // kapitel15/thread20.cpp
...
01  std::stack<int> stck;
02  std::mutex m;

03  void server(int data) {

```

```

04  while(data > 0) {
05      std::unique_lock<std::mutex> lck(m);
06      stck.push(data);
07      lck.unlock();
08      std::this_thread::sleep_for(
          std::chrono::milliseconds(100));
09      --data;
10  }
11 }

12 void consumer() {
13     int data = 0;
14     while( 1 != data ) {
15         std::unique_lock<std::mutex> lck(m);
16         if( ! stck.empty() ) {
17             data = stck.top();
18             stck.pop();
19             lck.unlock();
20             std::cout << std::this_thread::get_id() << ":"
                << data << std::endl;
21         }
22         else { lck.unlock(); }
23     }
24 }
...
25 std::thread th01(consumer);
26 std::thread th02(server, 100);
27 th01.join();
28 th02.join();

```

Dies ist ein einfaches Beispiel, in dem wir in den Zeilen 25 bis 26 zwei Threads starten, einmal den Thread `consumer` und einmal den Thread `server`. Das Prinzip ist einfach: Der Thread `server` bereitet unsere Daten vor. Im Beispiel werden einfach 100 Elemente auf einen Stack gelegt. Dieser Thread ist quasi der Sender oder auch Produzent (engl. *producer*) der Daten. Der andere Thread `consumer` holt diese Daten vom Stack und bearbeitet sie. Dieser Thread ist der Empfänger bzw. Konsument (engl. *consumer*) der Daten. Im Beispiel werden diese dann auch nur wieder ausgegeben. Beide Threads haben wir ordentlich mit einem Lock gesichert, womit die gemeinsame Ressource (hier der Stack in Zeile 01) ordentlich synchronisiert ist.

Leider hat dieses Beispiel einen Schönheitsfehler: Der Thread `consumer` führt ständig eine Schleife aus (Zeilen 14 bis 21). Dort wird immer wieder überprüft, ob der Stack nicht leer ist (Zeile 16). Ist der Stack leer, wird sofort der Lock in Zeile 22 freigegeben,

und es wird gleich wieder der nächste Schleifendurchlauf ausgeführt. Es wird hier quasi wertvolle CPU-Zeit für das Warten auf Daten in Form einer ausführenden Schleife verschwendet, die ständig einen bestimmten **Zustand** überprüft.

Condition-Variablen

Und genau für solche Zwecke kommen die *Condition-Variablen* (*Condition* = Zustand, Bedingung, Voraussetzung) ins Spiel – häufig auch zu *Bedingungsvariablen* eingedeutscht. Mit den Condition-Variablen haben Sie die Möglichkeit, die Ausführungsreihenfolge von Threads zu synchronisieren. Dasselbe Beispiel, wie es eben geschrieben wurde, soll jetzt mit einer Condition-Variablen verwendet werden:

```
00 // kapitel15/thread21.cpp
...
01 std::stack<int> stck;
02 std::condition_variable cond;
03 std::mutex m;

04 void server(int data) {
05     while(data > 0) {
06         std::unique_lock<std::mutex> lck(m);
07         stck.push(data);
08         lck.unlock();
09         cond.notify_one();
10         std::this_thread::sleep_for(
11             std::chrono::milliseconds(100));
12         --data;
13     }
14 }

14 void consumer() {
15     int data = 0;
16     while( 1 != data ) {
17         std::unique_lock<std::mutex> lck(m);
18         cond.wait(lck);
19         data = stck.top();
20         stck.pop();
21         lck.unlock();
22         std::cout << std::this_thread::get_id() << ":"
23             << data << std::endl;
24     }
...

```

In Zeile 02 finden Sie die Condition-Variable `cond`. Im `consumer-Thread` entfällt nun die Überprüfung, ob der Stack leer ist, komplett. Stattdessen finden Sie hier in Zeile 18 die Condition-Variable mit der Methode `wait()`, welche auf eine Benachrichtigung wartet. Solange die Condition-Variable in Zeile 18 keine Nachricht erhält, wird diese schlafen gelegt, und es werden keine wilden Schleifendurchläufe mehr gemacht. Dass die Methode `wait()` hier den Lock als Parameter übergibt, hat damit zu tun, dass Sie in Zeile 17 den Lock gesetzt haben. Damit jetzt nicht andere Threads ausgesperrt bleiben, während der Thread mit `wait()` schlafen gelegt wird, sorgt die Methode vorher noch dafür, dass der Lock freigegeben wird. Und keine Sorge, sobald ein anderer Thread den Schlafzustand des `consumer-Threads` wieder aufhebt, sorgt die Condition-Variable auch hier dafür, dass der Lock wieder gesetzt wird, bevor mit Zeile 19 fortgefahren wird.

Aufgeweckt wird in diesem Beispiel der `consumer-Thread`, sobald Daten auf dem Stack vorhanden sind. Und dass Daten hinzugekommen sind, weiß keiner besser als unser `server-Thread`, und der weckt den `consumer-Thread` über die Condition-Variable mit der Methode `notify_one()` in Zeile 09 auf.

In unserem Beispiel kann der `consumer-Thread` somit nur vom `server-Thread` mit der Methode `notify_one()` aufgeweckt werden. Es gibt aber noch eine zweite Möglichkeit, wie sich der `consumer-Thread` selbst aufwecken kann. Die Methode `wait()` bietet noch eine zweite Version an, in der Sie ein Prädikat verwenden können. Gibt das Prädikat `true` zurück, kann der Thread ebenfalls aufgeweckt werden. In unserem Beispiel könnte dies dann wie folgt aussehen:

```
18      cond.wait(lck, [](){return !stck.empty();});
```

Hier verwenden wir eine Lambda-Funktion, welche überprüft, ob unser Stack nicht leer ist. Ist der Stack nicht leer (`= true`), dann wird der `consumer-Thread` jetzt ebenfalls aufgeweckt und führt seine Arbeit aus. Ist der Stack hingegen leer, sorgt die Condition-Variable dafür, dass der Thread weiterhin schläft.

Mehrere Threads aufwecken

In unserem Beispiel haben wir eine einfache 1:1-Beziehung zwischen dem Sender und Empfänger erstellt. In der Praxis ist es durchaus möglich, dass mehrere Threads Sender bzw. Empfänger sind. Wenn Sie hier eine Condition-Variable verwenden und alle Empfänger, die gerade schlafen, aufwecken wollen, dann müssen Sie die Methode `notify_all()` statt `notify_one()` verwenden. Sollten mehrere Threads auf eine Benachrichtigung über eine Bedingungsvariable warten und Sie verwenden trotzdem `notify_one()`, dann wird irgendein Thread aufgeweckt. Welcher das ist, kann nicht vorausgesagt werden. Des Weiteren gibt es noch eine globale Funktion `std::notify_all_at_thread_exit(cond, lock)`, mit der Sie den Lock freigeben und die wartenden Threads aufwecken, wenn der aktuelle Thread beendet wird.

Methoden für Condition-Variablen

Jetzt kennen Sie die grundlegende Anwendung von Condition-Variablen und wissen, wie Sie hiermit Threads über Ereignisse synchronisieren können. Ein Thread wartet als Empfänger auf einen bestimmten Zustand, und ein anderer Thread informiert als Sender den Empfänger, dass der gewünschte Zustand jetzt vorliegt, damit der Empfänger mit der Arbeit anfangen kann. Zusätzlich kann für den Empfänger ein optionales Prädikat verwendet werden, mit dem dieser selbst den Zustand überprüft und sich somit auch unter Umständen selbst aufwecken kann, wenn der Zustand vorliegt und `true` ist. Es gibt auch noch weitere Möglichkeiten mit einer Zeitangabe, die zusätzlich auch noch mit einem Prädikat verbunden werden kann. In Tabelle 15.3 finden Sie einen kurzen Überblick über die vorhandenen Methoden mit Condition-Variablen.

Methode	Bedeutung
<code>cond.notify_one()</code>	Weckt wartenden Thread, der mit der Condition-Variablen <code>cond</code> verbunden ist, auf.
<code>cond.notify_all()</code>	Weckt alle wartenden Threads, die mit der Condition-Variablen <code>cond</code> verbunden sind, auf.
<code>cond.wait(lock)</code> <code>cond.wait(lock, predicat)</code>	Legt Thread schlafen und wartet auf Nachricht der Condition-Variablen <code>cond</code> (optional mit Prädikat).
<code>cond.wait_until(lock, time)</code> <code>cond.wait_until(lock, time, predicat)</code>	Legt Thread mit einer absoluten Zeitangabe schlafen und wartet auf Nachricht der Condition-Variablen <code>cond</code> (optional mit Prädikat).
<code>cond.wait_for(lock, time)</code> <code>cond.wait_for(lock, time, predicat)</code>	Legt Thread mit einer relativen Zeitangabe schlafen und wartet auf Nachricht der Condition-Variablen <code>cond</code> (optional mit Prädikat).

Tabelle 15.3 Übersicht der Methoden mit Condition-Variablen

15.6 Asynchrones Arbeiten (Future und Promise)

Die Standardbibliothek bietet neben der direkten Verwendung von `std::thread` noch eine zweite Möglichkeit mit `std::async()` und `std::packaged_task()` aus der Headerdatei `<future>`, um Threads zu starten. Wo `std::thread` noch auf einer niedrigeren Ebene basieren, sind die asynchronen Aufrufe mittels `std::async()` Funktionalitäten auf einer höheren Ebene.

Um den Unterschied zwischen einem Thread mit `std::thread` und einem Thread mit `std::async` etwas zu verdeutlichen, folgt hierzu ein Beispiel mit `std::thread`, welches wir anschließend mit `std::async` implementieren. Hierzu zunächst das Beispiel mit `std::thread`:

```
00 // kapitel15/thread22.cpp
...
01 void fibonacci(int n) {
02     if(n<0) throw std::runtime_error("Negativer Wert");
03     int i=0;
04     unsigned long a=1, b=0, fibo=0;
05     while(n > i) {
06         b=fibo;
07         fibo=a+b;
08         a=b;
09         ++i;
10     }
11     std::cout << "Fibonacci-Wert von " << n << " = "
                << fibo << std::endl;
12 }
...
13 std::thread th01(fibonacci, -5); // Löst Ausnahme aus
14 std::thread th02(fibonacci, 8);
15 //th01.join();
16 //th02.join();
```

Das Beispiel wird gleich doppelt mit `terminate()` in den Abgrund gerissen. Wenn Sie in Zeile 13 den Thread mit einem negativen Wert starten, wird zunächst in Zeile 02 eine Ausnahme geworfen. Und bei innerhalb von Threads gestarteten Funktionen müssen Sie sich selbst um die Behandlung von Ausnahmen kümmern. Wenn eine Ausnahme den Thread verlässt, dann wird das Programm mit `terminate()` beendet. Hier müssten Sie eine Ausnahmebehandlung innerhalb des Threads `fibonacci` schreiben. Dasselbe passiert auch, wenn die Threads noch laufen und sich der Hauptthread beendet, ohne mit `join()` auf das Ende der Threads zu warten oder die Verbindung zu diesem mit `detach()` zu lösen.

Das Problem haben Sie hingegen bei der Verwendung von `async()` aus der Headerdatei `<future>` nicht. Zunächst also das gleichwertige Gegenstück zum eben erstellten Listing mit `async()`:

```
00 // kapitel15/thread23.cpp
01 #include <future> // für async()
...
02 void fibonacci(int n) {
...

```

```

13 }
...
14 auto f1 = std::async( fibonacci, -5);
15 auto f2 = std::async( fibonacci, 8);

```

Das erneute Abdrucken der kompletten Funktion können wir uns hier ersparen. Auch hier werden jeweils beide Ausführungen in einem eigenen Thread ausgeführt. Aber die Rückgabewerte und Ausnahmen werden hierbei anders behandelt. Die Ausnahmen erscheinen jetzt im Hauptthread, wo Sie das Ergebnis abholen und bearbeiten können. Auch der Hauptthread beendet sich hier nicht einfach vorher, sondern wartet darauf, dass der Thread zu Ende ausgeführt wird, bevor er (der Hauptthread) sich beenden kann. Sie finden hier also viele nützliche Automatismen, die das Thread-Leben erheblich vereinfachen.

Ausnahme abholen

Die magische Funktion, mit der Sie die Ausnahme jetzt abholen können, lautet `get()`. Folgendes Beispiel kümmert sich jetzt um die Ausnahme, die aus dem Thread `fibonacci` geworfen werden kann:

```

00 // kapitel15/thread24.cpp
01 #include <future>
...
02 auto f1 = std::async( fibonacci, -5);
03 auto f2 = std::async( fibonacci, 8);
04 try {
05     f1.get();
06     f2.get();
07 }
08 catch(const std::runtime_error& e) {
09     std::cout << e.what() << std::endl;
10 }
...

```

Dass `get()` für mehr als nur zum Ausnahmen behandeln verwendet werden kann und der Typ von `f1` und `f2` hier mit `auto` noch nicht behandelt wurde, soll nochmals außen vor gelassen werden. Im Grunde geht es hier erst mal nur darum, wie Sie sehr komfortabel auf Ausnahmen mit der Verwendung von `std::async()` oder besser mit der Verwendung von `future`, reagieren können.

15.6.1 Futures

Was ist, wenn Sie jetzt in der Funktion `fibonacci()` den berechneten Wert im Thread als Rückgabewert haben wollen, um im Hauptthread damit weiterzuarbeiten, anstatt

ihn auszugeben? Sie werden hierzu wohl einen gemeinsamen Speicher in Form einer globalen Variablen oder eben den Wert über `std::ref()` als Referenz im Hauptthread holen müssen. Egal, wie Sie es machen, sobald gemeinsame Variablen im Spiel sind, kommen Sie nicht um Mutexe, Locks und gegebenenfalls Condition-Variablen herum. Ein ziemlicher Aufwand, nur um einen Wert aus einem Thread zu holen.

Mit den *Futures* und dessen Variadic Template `std::async()` können Sie sich diese manuelle Synchronisation sparen. Ein Aufruf von einem `std::async()` liefert ein `std::future`-Objekt zurück. Dieses `std::future`-Objekt können wir verwenden, um das Ergebnis der Funktionsausführung auszuwerten. An das Ergebnis in `std::future` kommen Sie mit der Methode `get()` heran. Der Clou ist dabei, solange das Ergebnis nicht berechnet wurde, blockiert `std::async()` die Methode `get()` – oder genauer, es findet eine Synchronisation statt. Auch Ausnahmen, wie bereits kurz gesehen, erscheinen nur beim Aufruf mit `get()` und nicht mehr im Thread.

Das folgende Listing demonstriert Ihnen jetzt, wie Sie Ergebnisse aus einem Thread mit Futures zurückgeben und auswerten können, ohne einen großen Aufwand mit Synchronisationen erstellen zu müssen:

```
00 // kapitel15/thread25.cpp
01 #include <future>
...
02 unsigned long fibonacci(int n) {
03     int i=0;
04     if(n<0) throw std::runtime_error("Negativer Wert");
05     unsigned long a=1, b=0, fibo=0;
06     while(n > i) {
07         b=fibo;
08         fibo=a+b;
09         a=b;
10         ++i;
11     }
12     return fibo;
13 }
...
14 unsigned long x=0, y=0;
15 std::future<unsigned long> f1 = std::async( fibonacci, 10);
16 std::future<unsigned long> f2 = std::async( fibonacci, 25);
17 try {
18     x = f1.get();
19     y = f2.get();
20 }
21 catch(const std::runtime_error& e) {
22     std::cout << e.what() << std::endl;
```

```

23 }
24 std::cout << "Fibonacci von 10 ist " << x << std::endl;
25 std::cout << "Fibonacci von 25 ist " << y << std::endl;
...

```

In den Zeilen 15 und 16 starten wir jetzt mit `std::async()` jeweils einen neuen Thread, wo beide Male das Ergebnis einer bestimmten Fibonacci-Zahl zurückgegeben wird, welche wir als Argument mit `an fibonacci()` übergeben. Als Rückgabewert liefert der `std::async`-Aufruf ein `future<unsigned long>`-Objekt zurück. Entsprechend wurde hier auch die Funktion `fibonacci()` angepasst, welche jetzt als Rückgabewert in Zeile 02 ebenfalls ein `unsigned long` im Funktionskopf zurückgibt. Und statt der Ausgabe erfolgt jetzt auch die Rückgabe der Berechnung in Zeile 12. Natürlich ist es hier von besonderer Bedeutung, dass der Rückgabewert vom Thread auch mit dem Template-Wert von `std::future` zwischen den spitzen Klammern übereinstimmt.

Jetzt können Sie den Rückgabewert bequem in den Zeilen 18 und 19 über das `std::future`-Objekt und dessen Methoden `get()` den `unsigned long`-Typen zuweisen. Die `get()`-Methode wartet so lange, bis der Kinderthread beendet wurde. Falls eine Ausnahme im Thread geworfen wurde, sind wir auch hierfür gerüstet, weil wir `get()` in einen `try-catch`-Block gepackt haben, um darauf zu reagieren. Die Ausgabe in Zeile 24 und 25 gibt das berechnete Ergebnis im Hauptthread aus.

Die Ausgabe in den Zeilen 24 und 25 ist allerdings nicht sehr schön, wenn eine Ausnahme geworfen wird, weil dann gegebenenfalls einfach 0 als Ergebnis ausgegeben wird. Daher ist es in diesem Beispiel sinnvoller, die Ausgabe gleich wie folgt in den `try`-Block zu stecken:

```

...
std::future<unsigned long> f1 = std::async( fibonacci, 10);
std::future<unsigned long> f2 = std::async( fibonacci, 25);
try {
    std::cout << "Fibonacci von 10 = "
        << f1.get() << std::endl;
    std::cout << "Fibonacci von 25 = "
        << f2.get() << std::endl;
}
catch(const std::runtime_error& e) {
    std::cout << e.what() << std::endl;
}
...

```

Tatsächlich ein neuer Thread?

Im Beispiel oben wurde behauptet, dass mit `std::async()` ein neuer Thread gestartet wurde. Das ist allerdings nicht ganz richtig so. Wenn die Ausführung der Berechnun-

gen im Hauptthread gegebenenfalls schneller ist, wird diese im Hauptthread ausgeführt, und es wird kein neuer Thread gestartet. Die Entscheidung, was hier jetzt »besser« ist und was nicht, nimmt Ihnen die C++-Laufzeitumgebung ab.

Sie können als Anwender trotzdem Einfluss auf diese Entscheidung nehmen, indem Sie bei `std::async()` die Option `std::launch::async` oder `std::launch::deferred` angeben. Wobei mit der Option `std::launch::async` ein neuer Thread für die Berechnung gestartet wird und mit `std::launch::deferred` die Berechnung im selben Thread ausgeführt wird. Der Thread mit `std::launch::deferred` wird allerdings erst mit dem `get()`-Aufruf gestartet (also eine verzögerte Ausführung). Die Standardeinstellung ist eine ODER-Verknüpfung beider Werte, um die Laufzeitumgebung entscheiden zu lassen.

Hier die beiden Optionen im Einsatz:

```
01 auto f1 = std::async(std::launch::async, fibonacci, 10);
02 auto f2 = std::async(std::launch::deferred, fibonacci, 25);
...
```

Der Übersichtlichkeit zuliebe haben wir für `std::future<unsigned long>` wieder `auto` verwendet. Den ersten `fibonacci`-Aufruf führen wir in einem extra Thread (Zeile 01) aus. Der zweite Aufruf in Zeile 02 hingegen wird im Hauptthread ausgeführt. Das ist durchaus sinnvoll, weil genau genommen ohnehin nur zwei Berechnungen gestartet werden. Würden Sie den zweiten Aufruf auch als Thread (mit `std::launch::async`) ausführen, würden gegebenenfalls drei Threads gleichzeitig laufen.

`std::launch::async` oder `std::launch::deferred`

Wissen Sie nicht, was Sie verwenden sollen, dann lassen Sie die Laufzeitumgebung entscheiden und verwenden keine Optionen. Die Laufzeitumgebung stellt sicher, dass das System nicht mit zu vielen Threads überlastet wird und berücksichtigt auch die Anzahl vorhandener Kerne bzw. CPUs im System. Wenn es spezielle Anforderungen gibt, dann können Sie immer noch die eine oder andere Option verwenden.

15.6.2 Promise

Beim Ausführen von asynchronen Funktionen ist nicht nur möglich, einen Wert von einem Kinderthread an den Elternthread zurückzugeben, sondern Sie können auch einen Wert vom Elternthread an den Kinderthread übergeben. Für diese Art der Zusammenarbeit mit `std::future` stellt die Standardbibliothek `std::promise` zur Verfügung. Damit diese funktioniert, müssen Sie `std::future` mit `std::promise` verbinden.

Hierzu zunächst der Code, welcher *Future* und *Promise* als verbundene Einheit demonstriert:

```

00 // kapitel15/thread26.cpp
01 #include <future>
...
02 unsigned long fibonacci(std::future<unsigned long> f) {
03     int i=0;
04     unsigned long n = f.get();
05     unsigned long a=1, b=0, fibo=0;
06     while(n > i) {
07         b=fibo;
08         fibo=a+b;
09         a=b;
10         ++i;
11     }
12     return fibo;
13}
...
14 unsigned long x=0, y=0;
15 std::promise<unsigned long> p1;
16 std::promise<unsigned long> p2;
17 std::future<unsigned long> fu1 = p1.get_future();
18 auto fu2 = p2.get_future();
19 auto f1 = std::async( fibonacci, std::move(fu1));
20 auto f2 = std::async( fibonacci, std::move(fu2));
21 // nach einiger Arbeit
22 std::this_thread::sleep_for(std::chrono::seconds(1));
23 p1.set_value(10);
24 p2.set_value(25);
25 try { // eine Ausnahme könnte auftreten
26     x = f1.get();
27     y = f2.get();
28 }
29 catch(...) {
30     std::cout << "Ausnahme abgefangen" << std::endl;
31 }
32 std::cout << "Fibonacci von 10 ist " << x << std::endl;
33 std::cout << "Fibonacci von 25 ist " << y << std::endl;
...

```

In den Zeilen 15 und 16 definieren wir zwei `std::promise`-Objekte. Zu den `std::promise`-Objekten benötigen wir jetzt auch zwei `std::future`-Objekte. In den Zeilen 17 und 18 fordern wir ein zum `std::promise` gehörendes `std::future` an. Die neuen

Threads starten Sie in den Zeilen 19 und 20, wo Sie jetzt jeweils ein `std::promise`-Objekt als Parameter übergeben. Im Beispiel haben wir diese `std::promise`-Objekte mit `std::move` gleich verschoben. Alternativ könnten Sie ein `std::promise` auch als Referenz mit `std::ref()` übergeben. Allerdings müssen Sie dann auch den Funktionskopf entsprechend anpassen.

Und wenn Sie ein `std::future`-Objekt an den Thread übergeben, müssen Sie natürlich auch die Syntax des Threads anpassen. In Zeile 02 lautet der Parameter daher jetzt auch `std::future<unsigned long>`. Den übergebenen Wert vom Typ `unsigned long` holen wir in Zeile 04 wieder mit der Methode `get()` ab und weisen diesen `n` zu. Die Methode `get()` wartet auch hier wieder so lange, bis sie ihre Daten erhält. Der Rest der Funktion `fibonacci` bleibt unverändert.

Bis jetzt haben Sie allerdings noch nichts anderes gemacht, als dem Kinderthread versprochen (Promise) in Zukunft (Future) einen Wert zu senden. Der Kinderthread hat noch keinen Wert. Um das Versprechen zu halten, müssen Sie irgendwann den Wert übergeben. Dies erledigen wir in diesem Beispiel mit `set_value()` in den Zeilen 23 und 24. Mit dem Aufruf von `set_value()` bekommen die mit `get()` (Zeile 04) wartenden Threads ihre Daten, und die Threads können weiterarbeiten. Der Rest ist dann wieder wie gehabt. In den Zeilen 26 und 27 holen wir wieder den Wert der Berechnung und geben diesen in den Zeilen 32 und 33 aus.

Template-Klassen

Da `std::future` und `std::promise` Template-Klassen sind, können Sie wieder alle eingebauten Typen oder auch eigene Klassen verwenden. Wichtig ist hierbei natürlich auch, dass Sie bei `std::future` und `std::promise` in den spitzen Klammern immer den passenden Typ verwenden. Im Beispiel wurde hierfür immer `unsigned long` verwendet. Ganz besonders wichtig ist dann auch, dass Sie bei `set_value()` den versprochenen Typ abliefern.

Future und Promise nochmals Schritt für Schritt

Im Grunde ist das Future- und Promise-Prinzip nicht wirklich komplex, aber da hierfür recht viele Schritte nötig sind, verliert man anfangs gerne mal die Übersicht. Daher werden hier nochmals die einzelnen Schritte aufgelistet:

1. Erzeugen Sie ein neues `std::promise`.
2. Fordern Sie ein zum `std::promise` gehörendes `std::future` mit `get_future()` an.
3. Starten Sie einen neuen Thread mit `std::async()` oder auch `std::thread`, und geben Sie das zu `std::promise` gehörende `std::future` als Referenz über `std::ref()` mit, oder verschieben Sie es mit `std::move()`.
4. Weisen Sie dem `std::promise` den Wert mit `set_value()` zu.

5. Im Aufrufer richten Sie ein `std::future` als Parameter ein, der auf sein Versprechen wartet.
6. Holen Sie den Wert von `std::future` im Aufrufer mit `get()` ab.

std::shared_future

Wollen Sie einen gemeinsamen Zustand von mehr als einem Thread anfordern, können Sie entweder genauso viele `std::promise` wie `std::future` erzeugen, wie Sie es im Beispiel bereits gemacht haben, wo Sie zwei `std::promise`- und zwei `std::future`-Objekte erzeugt und verbunden haben. Es gibt aber noch eine zweite Möglichkeit mit `std::shared_future`. Im Gegensatz zu `std::future` können Sie mit `std::shared_future` den Wert als Kopie übergeben – was auch der Grund ist, dass es hiermit funktioniert. Ansonsten ist die Verwendung ziemlich einfach, weil die Schnittstelle ähnlich wie schon `std::future` ist. Ein Beispiel mit `std::shared_future`:

```
00 // kapitel15/thread27.cpp
01 #include <future>
...
02 void function(std::shared_future<int> f, int val) {
03     int n = f.get();
04     std::cout << n * val << std::endl;
05 }
...
06 std::promise<int> p;
07 auto f = p.get_future();
08 std::shared_future<int> sf = f.share();
09 auto f1 = std::async( function, sf, 2);
10 auto f2 = std::async( function, sf, 4);
11 auto f3 = std::async( function, sf, 8);
12 auto f4 = std::async( function, sf, 16);
...
13 p.set_value(10);
14 try {
15     f1.get();
16     f2.get();
17     f3.get();
18     f4.get();
19 }
20 catch(...) {
21     std::cout << "Ausnahme im Hauptthread abgefangen\n";
22 }
```

Ein solches `std::shared_future` können wir mithilfe der Methode `share()` von `std::future` erzeugen, wie in Zeile 08 zu sehen ist. Im Gegensatz zu einem gewöhnli-

chen `std::future` kann ein `std::shared_future` kopiert werden, weshalb wir bei `std::async()` (in den Zeilen 09 bis 12) `std::shared_future` als Kopie übergeben. Die Funktion, die als Thread aufgerufen werden soll, benötigt als Parameter jetzt ebenfalls den Typ `std::shared_future` (siehe Funktionskopf in Zeile 02). Zusätzlich wurde hier beim Starten mit `std::async()` und im Funktionskopf in Zeile 02 noch ein weiterer `int`-Wert verwendet und übergeben.

Wenn wir jetzt in Zeile 13 den Wert mit `set_value()` setzen, werden alle Threads, welche mit `get()` auf den Wert warten, ausgeführt. Im Beispiel wird der gemeinsam versprochene Wert 10 mit `set_value` übergeben und in den entsprechenden Threads in Zeile 04 mit jeweils übergeben Werten von `val` multipliziert. Hier wird 10 mit 2, 4, 8 und dann 16 multipliziert und das Ergebnis ausgegeben. Die Ausgabe über das Stream-Objekt `cout` in Zeile 04 müssten Sie hier gegebenenfalls noch synchronisieren.

Versprechen gebrochen

Nehmen wir mal an, wir können das Versprechen in unserem Beispiel nicht einhalten, weil der Wert für die Fibonacci-Zahl einfach zu hoch war. Hier können wir das Programm nicht einfach so beenden, weil wir ja ein Versprechen gegeben haben und ein bereits gestarteter Thread mit `get()` auf seinen Wert wartet, den wir eigentlich mit `set_value()` senden sollten. Wenn wir das Versprechen nicht halten können, dann sollten wir die Funktion `set_exception()` von `std::promise` verwenden, um dem mit `get()` wartenden Thread eine Ausnahme zu schicken. Natürlich muss dieser auch auf die Ausnahme reagieren. Das folgende Beispiel soll dieses Szenario demonstrieren:

```
00 // kapitel15/thread28.cpp
01 #include <future>
...
02 unsigned long fibonacci(std::future<unsigned long> f) {
03     int i=0;
04     unsigned long n = 1;
05     try {
06         n = f.get();
07     }
08     catch(std::runtime_error& e) {
09         std::cout << "Ausnahme in fibonacci : ";
10         std::cout << e.what() << std::endl;
11         throw; // Weiter werfen
12     }
13     unsigned long a=1, b=0, fibo=0;
14     while(n > i) {
...
15     }
```

```

16     return fibo;
17 }
...
18 unsigned long x1, x2;
19 std::promise<unsigned long> p1;
20 auto fu1 = p1.get_future();
21 auto f1 = std::async( fibonacci, std::move(fu1));
22 std::cout << "Fibonacci-Nr.: ";
23 std::cin >> x1;
24 if( x1 > 100 ) {
25     p1.set_exception(std::make_exception_ptr(
                std::runtime_error("Wert zu groß")));
26 }
27 else {
28     p1.set_value(x1);
29 }

30 try {
31     x2 = f1.get();
32     std::cout << "Fibonacci von " << x1 << " ist "
                << x2 << '\n';
33 }
34 catch(std::runtime_error& e) {
35     std::cout << "Ausnahme auch im Hauptthread abgefangen:";
36     std::cout << e.what() << std::endl;
37 }
...

```

In den Zeilen 22 und 23 werden Sie aufgefordert, einen Wert für eine Fibonacci-Nummer einzugeben. Ist der Wert größer als 100, lassen wir die Berechnung sein und setzen in Zeile 25 die Ausnahme mit `set_exception()` auf `std::runtime_error` (mithilfe der Funktion `std::make_exception_ptr()`) für den Thread, welchen wir in Zeile 21 gestartet haben und der in Zeile 06 auf unser Versprechen mit `get()` wartet. Wurde ein Wert kleiner als 100 eingegeben, bekommt unser Thread ganz gewöhnlich mit `set_value()` in Zeile 28 den Wert. Auch in unserer Funktion `fibonacci` haben wir bereits eine Ausnahmebehandlung eingerichtet.

Wie gesagt, wenn ein Wert größer als 100 eingegeben wurde, wird mit `set_exception()` die Ausnahme an den wartenden Thread übergeben. Damit geben wir Bescheid, dass wir unser Versprechen nicht einhalten können, und fangen in Zeile 06 die Ausnahme `std::runtime_error` ab, welche wir in den Zeilen 08 bis 12 behandeln. In Zeile 11 werfen wir die Ausnahme erneut aus, wo wir dem Hauptthread, welcher ebenfalls eine Ausnahmebehandlung in den Zeilen 30 bis 37 eingerichtet hat, informieren, dass keine Berechnung stattgefunden hat. In Zeile 31 würde dann auch hier

die Ausnahme `std::runtime_error` abgefangen werden und auch hier im Folgenden `catch` behandelt.

15.6.3 Methoden für `future` und `promise`

In Tabelle 15.4 und Tabelle 15.5 finden Sie einen kurzen Überblick über die Methoden der Klassen-Templates `std::future` und `std::promise`.

Methode	Beschreibung
<code>fu.share()</code>	Gibt ein <code>std::shared_future</code> zurück.
<code>fu.get()</code>	Wartet, bis ein Wert verfügbar ist, und gibt diesen zurück. Kann auch eine Ausnahme zurückgeben.
<code>fu.valid()</code>	Testet, ob ein gemeinsamer Zustand vorhanden ist.
<code>fu.wait()</code>	Blockiert, bis der gemeinsame Zustand eintritt.
<code>fu.wait_for(time)</code>	Blockiert für eine bestimmte Zeitspanne, bis der gemeinsame Zustand eintritt.
<code>fu.wait_until(time)</code>	Blockiert bis zu einem bestimmten Zeitpunkt, bis der gemeinsame Zustand eintritt.

Tabelle 15.4 Methoden für »`std::future`«

Ein paar Worte noch zu den `wait`-Methoden von `std::future`, da wir in den Beispielen immer nur `get()` verwendet haben. Die Methode `get()` ruft intern die Methode `wait()` auf und wartet so lange, bis ein gemeinsamer Wert zur Verfügung steht. Beachten Sie allerdings, dass `wait()` nur blockiert und Sie den Wert trotzdem mit `get()` abholen müssen. Wenn Threads keinen Wert zurückgeben (`void`), lässt sich `wait()` sehr schön zur Synchronisation von zwei Threads verwenden, ähnlich wie `Condition-Variablen`.

Methode	Beschreibung
<code>p.get_future()</code>	Gibt ein <code>std::future</code> zum versprochenen <code>std::promise</code> -Wert zurück.
<code>p.set_value(val)</code>	Setzt den versprochenen Wert auf <code>val</code> .
<code>p.set_exception(ex)</code>	Wenn ein Versprechen nicht eingehalten werden kann, wird der gemeinsame Zustand auf die Ausnahme <code>ex</code> gesetzt.
<code>p2.swap(p1)</code>	Tauscht den Zustand zweier <code>std::promise</code> -Objekte.

Tabelle 15.5 Methoden für »`std::promise`«

Zuweisung von `std::future` und `std::promise` nicht möglich

Hier soll noch angemerkt werden, dass weder `std::future` noch `std::promise` zugewiesen werden können. Ein `fu2=fu1` oder ein `p2=p1` sind nicht möglich. Wenn Sie eine Zuweisung machen wollen, dann müssen Sie auch hier wieder eine Verschiebung mit `std::move()` durchführen.

15.7 packaged_task

Mit `std::packaged_task` bekommen Sie noch mehr Kontrolle, wann die Berechnung ausgeführt werden soll. Dieses Template ist ein Wrapper um das eigentlich aufrufbare Objekt. Wie der Namen schon sagt, können Sie hiermit eine Arbeit (Task) in einem Paket vorbereiten und zu einem späteren Zeitpunkt in einem Thread aufrufen. Intern wird hiermit ein `std::future` mit einem `std::promise` verbunden, so dass ganz bequem der Wert des aufgerufenen Objekts abgeholt werden kann. Das folgende Beispiel soll eine einfache Verwendung von `std::packaged_task` demonstrieren:

```
00 // kapitel15/thread29.cpp
01 #include <future>
...
02 int fibonacci(int n) {
03     if(n<0) throw std::runtime_error("Negativer Wert");
04     int i=0;
05     int a=1, b=0, fibo=0;
06     while(n > i) {
07         b=fibo;
08         fibo=a+b;
09         a=b;
10         ++i;
11     }
12     return fibo;
13 }
...
14 std::packaged_task<int(int)> task(&fibonacci); // 1.Schritt
15 auto fu1 = task.get_future(); // 2.Schritt
16 // ... irgendwann später
17 task(20); // 3.Schritt
18 int val = fu1.get(); // 4.Schritt
19 std::cout << val << std::endl;
...
```

Hier verwenden wir erneut unsere bekannte `fibonacci`-Funktion. Wie Sie sehr schön erkennen können, taucht hier der Promise-Teil gar nicht erst auf. Dieser Teil wird jetzt vom Wrapper `std::packaged_task` übernommen. Sie müssen hier jetzt nur die Aufgabe verpacken, wie wir dies in Zeile 14 gemacht haben. Auf die genauere Syntax gehen wir noch ein. Anschließend müssen Sie das von `std::packaged_task` zur Verfügung gestellte Promise mit dem Future verbinden. Dies wurde wieder wie gehabt mit `get_future()` in Zeile 15 erledigt. Jetzt können Sie die Aufgabe starten, indem Sie die Argumente an das Promise übergeben, wie in Zeile 17 geschehen. Mithilfe des Futures und dessen Methode `get()` können Sie das Ergebnis abholen (siehe Zeile 18). Hier sind also lediglich vier Schritte nötig.

Jetzt zu der etwas eigenwilligen Syntax der Signaturen für `std::packaged_task`. Wenn Sie die Syntax

```
std::packaged_task<int(int)> task(&fibonacci);
```

genauer betrachten, erkennen Sie darin auch die Funktion wieder. An `int(int)` erkennen Sie, dass unsere Funktion einen Parameter vom Typ `int` erwartet und als Rückgabewert ebenfalls `int` zurückgibt. Also genau wie unser Funktionskopf:

```
int fibonacci(int n);
```

Als neuen Thread starten

Das Ganze lässt sich jetzt natürlich auch ganz komfortabel als neuer Thread mit `std::async` starten:

```
00 // kapitel15/thread30.cpp
...
01 std::packaged_task<int(int)> task(&fibonacci);
02 auto fu1 = task.get_future();
03 std::async(std::move(task), 20);
04 int val = fu1.get();
05 std::cout << val << std::endl;
...
```

Oder natürlich auch mit `std::thread`:

```
00 // kapitel15/thread31.cpp
...
01 std::packaged_task<int(int)> task(&fibonacci);
02 auto fu1 = task.get_future();
03 std::thread th01(std::move(task), 20);
04 int val = fu1.get();
05 std::cout << val << std::endl;
06 th01.join();
...
```

Da sich auch Tasks nicht kopieren lassen, müssen Sie hier auch wieder `std::move()` verwenden.

Argumente mit dem Task binden

Es ist durchaus möglich, mit `std::bind()` aus der Headerdatei `<functional>` die Argumente gleich mit in den Task zu packen. Dies ist nötig, weil bspw. Folgendes nicht funktioniert:

```
// Geht nicht:
std::packaged_task<int(int)> task(&fibonacci, 20);
```

Folgendermaßen können Sie Argumente mit einem Task binden:

```
00 // kapitel15/thread32.cpp
...
01 std::packaged_task<int()> task(std::bind(&fibonacci, 20));
02 auto fu1 = task.get_future();
03 std::async(std::move(task));
04 int val = fu1.get();
05 std::cout << val << std::endl;
...
```

Mithilfe von `std::bind()` wurde jetzt in Zeile 01 der Wert von `fibonacci` mit 20 vorbelegt. Dadurch ist es jetzt allerdings auch nicht mehr möglich, dass Sie später beim Aufruf diesem Task ein Argument mitgeben. Wenn Sie `task` aufrufen, wird dieser jetzt immer mit `fibonacci(20)` ausgeführt (wie dies in Zeile 03 der Fall ist). Damit dies auch funktioniert, musste auch die Signatur des Tasks mit `int()` angepasst werden. Der Task erwartet jetzt keinen Parameter mehr, weil wir diesen bereits vorbelegt haben.

Das Beispiel zeigt sehr schön, dass die Signatur beim Erstellen eines Tasks für `std::packaged_task` immer an den eigentlichen Aufruf des Tasks angepasst werden muss – also den Aufruf im Thread – und nicht an die Funktion, Lambda-Funktion oder das Funktionsobjekt angepasst wird. Sie haben die Möglichkeit, dem Task gleich beim Verpacken mit `std::packaged_task` über `std::bind` die Argumente zu übergeben, oder Sie können dies – ohne `std::bind` – beim Starten des Threads wie gehabt übergeben.

Als Lambda-Funktion und Funktionsobjekt

Es ist auch nicht unwesentlich schwerer, solche Task mithilfe von Lambda-Funktionen oder Funktionsobjekten zu realisieren. Damit Sie sich auch davon ein Bild machen können, finden Sie hierzu noch ein Beispiel, welches dies demonstriert:

```

00 // kapitel15/thread33.cpp
...
01 class Circle_area {
02 public:
03     double operator()(double r) {
04         return (r * r * 3.1415L);
05     }
06 };
...
07 std::packaged_task<int(int, int)> task1([](int a, int b) {
08     return a * b;
09 });
10 Circle_area c01;
11 std::packaged_task<double(double)> task2(c01);
12 auto fu1 = task1.get_future();
13 auto fu2 = task2.get_future();
14 std::async(std::move(task1), 20, 10);
15 std::thread th01(std::move(task2), 5.5);
16 int val = fu1.get();
17 double dval = fu2.get();
18 std::cout << val << std::endl;
19 std::cout << dval << std::endl;
20 th01.join();
...

```

In den Zeilen **07** bis **09** packen wir eine Lambda-Funktion, welche lediglich zwei `int`-Werte miteinander multipliziert und das Ergebnis ebenfalls als `int`-Wert zurückgibt, in einen `std::packaged_task`. Gleiches machen wir auch in Zeile **11** mit dem Funktionsobjekt, welches die Kreisfläche anhand eines übergebenen Radius berechnet. Ansonsten bietet das Listing nichts Neues mehr.

Ganzer Pool von Threads

In der Praxis eignen sich diese verpackten Tasks hervorragend dazu, um möglichst viele solcher arbeitenden Tasks im Hauptthread zu einem weiteren Paket zu versammeln (einem Pool von Threads), um so möglichst viele Arbeiter-Threads (Worker-Threads) auszuführen. Hierfür packen Sie zunächst die Tasks in einen `std::packaged_task`, verbinden die Promises mit den Futures und packen dann diese Tasks in einen Container von Tasks. Die einzelnen im Container verpackten Tasks können Sie jetzt bei Bedarf jeweils in einem eigenen Thread starten und ausführen. Das Ergebnis können Sie wieder wie gehabt mit `get()` aus den Futures abholen. Hier ein extrem einfaches Beispiel, welches wieder unsere `fibonacci`-Funktion verwendet:

```

00 // kapitel15/thread34.cpp
...
01 std::packaged_task<int()> task1(std::bind(&fibonacci, 10));
02 std::packaged_task<int()> task2(std::bind(&fibonacci, 20));
03 std::packaged_task<int()> task3(std::bind(&fibonacci, 30));

04 auto f1 = task1.get_future();
05 auto f2 = task2.get_future();
06 auto f3 = task3.get_future();

07 std::vector< std::packaged_task<int()> > vecTask;
08 vecTask.push_back(std::move(task1));
09 vecTask.push_back(std::move(task2));
10 vecTask.push_back(std::move(task3));

11 while(!vecTask.empty()) {
12     std::packaged_task<int()> task=std::move(vecTask.back());
13     vecTask.pop_back();
14     std::thread th(std::move(task));
15     th.detach();
16 }
17 int all = f1.get() + f2.get() + f3.get();
18 std::cout << all << std::endl;
...

```

In den Zeilen **01** bis **03** haben wir die Tasks jeweils in einen `std::packaged_task` verpackt und in den Zeilen **04** bis **06** die Promises mit dem Futures verbunden. In den Zeilen **08** bis **10** wurden dann die einzelnen Tasks in einen Container (hier `std::vector`) gepackt. Hier haben Sie also einen Container von Tasks für den Typ `std::packaged_task<int()>`. In den Zeilen **11** bis **16** holen wir immer jeweils den letzten Task aus dem Container und führen diesen in einem neuen Thread aus. Dies machen wir so lange, bis der Container leer (`empty()`) ist. Am Ende holen wir in Zeile **17** mithilfe der Futures die Ergebnisse mit `get()` ab. Im Beispiel wurden einfach alle Ergebnisse zusammenaddiert. Auf eine Ausnahmebehandlung haben wir der Übersichtlichkeit zuliebe verzichtet, die sich eigentlich immer bei den `get()`-Funktionen befinden sollte.

Natürlich können Sie hier auch verschiedene andere in den Container von Tasks packen. Allerdings müssen Sie immer auf die korrekte Signatur aufpassen. Alle Funktionen, Lambda-Funktionen oder Funktionsobjekte benötigen dieselbe Signatur.

Kapitel 16

Weitere Neuerungen in C++11

Viele Neuerungen von C++11 wurden zwar im Buch immer wieder verwendet oder erwähnt, aber nicht immer ausreichend behandelt. Und manche Neuerungen lassen sich auch nicht einfach so nebenbei erläutern und benötigen einen eigenen Abschnitt. In diesem Kapitel sollen die zu kurz gekommenen C++11-Themen nun nachgeholt werden.

Mit C++11 wurde die Sprache grundlegend überarbeitet und in vielen Bereichen vereinfacht und auch erweitert. In den vorangehenden Kapiteln haben Sie bereits viele Neuerungen von C++11 kennengelernt. In diesem Kapitel sollen noch ein paar Neuerungen zusammengefasst werden, welche im Buch zwar immer wieder verstreut angesprochen und verwendet wurden, aber nie so recht zum Thema passen wollten.

16.1 Move-Semantik und Perfect Forwarding

Eine der besten Neuerungen in C++11 dürften wohl die *RValue-Referenzen* sein. Sie haben diese Referenzen schon das eine oder andere Mal im Buch mit der `&&`-Syntax gesehen. Auf den ersten Blick wirkt diese vielleicht komplex, aber in Wirklichkeit erleichtert sie dem Programmierer das Leben doch erheblich, weil sich dieser doch an vielen Stellen um gar nichts mehr kümmern muss. Aber dazu erfahren Sie im Verlauf des Abschnitts noch mehr. Mit den neuen RValue-Referenzen wurde in C++11 Folgendes ermöglicht:

- *Move-Semantik*: Damit kann ein Objekt als Ganzes in ein anderes Objekt verschoben werden. Bisher war hier nur aufwendiges Kopieren möglich.
- *Perfect Forwarding*: Damit können Sie Bibliotheken mit weniger Überladungen als Argumente an eine Funktion weiterleiten.

Wenn Sie sich mit diesem Thema auseinandersetzen, ist es unerlässlich, wenigstens im Ansatz den Unterschied zwischen *LValue* und *RValue* zu kennen. Daher folgt hierzu ein kurz gehaltener Exkurs.

16.1.1 Exkurs: LValue und RValue

Wenn Sie mit diesen Begriffen bereits vertraut sind, können Sie diesen Exkurs auch überspringen und gleich mit Abschnitt 16.1.2 mit der Move-Semantik weitermachen. Viele Programmierer haben vielleicht schon jahrelang mit C++ entwickelt, ohne sich je Gedanken um die beiden Begriffe gemacht zu haben bzw. machen zu müssen. Aber da C++11 RValue-Referenz kann, ist das ein guter Grund, sich jetzt damit auseinanderzusetzen.

Hierzu schon mal eine verallgemeinerte Definition, was ein RValue und was ein LValue in C++ ist:

- LValue: Dies ist ein Objekt, welches anhand einer Adresse im **Speicher** lokalisiert werden kann.
- RValue: Dies ist ein Objekt, welches kein LValue ist, also nicht anhand einer Adresse im Speicher lokalisiert werden kann. Oder einfacher: RValues haben keinen Bezeichner. RValues sind häufig temporäre Objekte, die niemals mehr ein zweites Mal verwendet werden können.
- Jeder C++-Ausdruck ist entweder ein RValue oder ein LValue.

LValues

Dies alles hört sich komplexer an, als es ist. Daher ein einfacher Code dazu, der die LValues erklären soll:

```
01 int ival = 12345;
02 int* iptr = &ival;
03 class Data{};
04 Data dat;
```

In Zeile **01** sehen Sie mit `ival` einen LValue. Dies ist ein LValue, weil Sie ihn anhand der Adresse von `ival` lokalisieren können, wie Sie es in Zeile **02** mit `&ival` sehen. Dasselbe gilt natürlich auch für benutzerdefinierte Typen wie in Zeile **04** mit `dat`. Die Instanz `dat` ist ebenfalls ein LValue, weil diese eine erreichbare Adresse im Speicher hat. Daraus lässt sich schon mal schlussfolgern, dass die meisten Variablen in C++ LValues sind.

RValues

Jetzt im Gegensatz dazu ein grundlegender Blick auf die RValues:

```
01 int ival = 12345;    // 12345 ist RValue

02 int *iptr = &12345; // Fehler !!!
03 12345 = 10;         // Fehler !!!
```

```

04 12345 = val;           // Fehler !!!

05 class Data{};
06 Data dat;
07 dat = Data(); // Data() ist ein RValue

08 int function(int val) { return val*val; }
09 int val = function(10); // function(10) ist ein RValue

```

Der Wert 12345 von Zeile **01** ist ein RValue. Dass es ein RValue ist, beweisen die Zeilen **02** bis **04**. Diese Operationen schlagen alle fehl, weil der Wert 12345 keine ansprechbare Adresse hat. Sie können 12345 keinen Wert zuweisen oder die Adresse von 12345 verwenden. Dasselbe gilt jetzt auch für `Data()` in Zeile **07**. `Data()` können Sie weder einen Wert zuweisen noch die Adresse von `Data()` verwenden. Daher ist `Data()` ein RValue. Und Gleiches auch noch in Zeile **09** mit `function(10)`, wo es ebenfalls nicht möglich ist, bspw. `funktion(10) = 100` zu schreiben – also auch einen RValue.

(LValue-)Referenz

Sicherlich haben Sie schon das eine oder andere Mal gehört, dass eine Referenz wie `int&` eine LValue-Referenz ist. Spätestens nach den folgenden Codezeilen werden Sie auch genau wissen warum:

```

01 int ival = 12345;
02 int& ref1 = ival; // Ok
03 int& ref2 = 54321; // Fehler !!!

```

In Zeile **02** initialisieren wir die Referenz `ref1` mit `ival`. Da `ival` ein LValue ist, ist das auch so in Ordnung. Daher kommt auch der Name LValue-Referenz, weil `ref1` eine Referenz auf den LValue-Wert `ival` ist. Zeile **03** hingegen führt zu einem Fehler, weil 54321 ein RValue ist und keine ansprechbare Adresse im Speicher enthält.

Ausnahme const-Referenz

Es gibt hier noch eine wichtige Ausnahme, wenn Sie eine konstante LValue-Referenz verwenden. Dieser können Sie einen RValue zuweisen:

```
const int& ref = 5; // Hier OK!!!
```

Das Prinzip kennen Sie allerdings auch schon aus den Funktionen:

```

int function(const int& x) { return x*x; }
...
function(val); // OK
function(100); // Möglich, dank const im Funktionsparameter

```

LValue-Referenz oder Referenz

Im weiteren Verlauf des Kapitels werden wir die gewöhnlichen Referenzen wie `int&` als LValue-Referenz bezeichnen, um diese von den neuen RValue-Referenzen unterscheiden zu können, welche mit `int&&` dargestellt werden.

16.1.2 RValue-Referenz und Move-Semantik**Voraussetzungen**

Im Abschnitt »Verschiebekonstruktor (C++11)« in Abschnitt 7.4.2 und in Abschnitt 7.10, »Klassen für das Verschieben schreiben (C++11)«, haben Sie bereits erfahren, wie Sie die Move-Semantik (oder auch Verschiebesemantik) mithilfe von RValue-Referenzen für eigene Typen implementieren können. Allerdings wurden hier niemals die eigentliche Move-Semantik bzw. die RValue-Referenzen genauer erläutert. Dieser Abschnitt soll jetzt die Ergänzung dazu liefern. Blättern Sie gegebenenfalls nochmals zu den eben genannten Kapiteln zurück, und lesen Sie sie aufmerksam durch.

Hier gehe ich jetzt davon aus, dass Sie den Unterschied zwischen RValue und LValue kennen. Falls nicht, bitte ich Sie, den Exkurs vom Abschnitt zuvor durchzulesen, da es sonst schwierig ist, die RValue-Referenzen zu verstehen. Konkret geht es jetzt in diesem Abschnitt nicht mehr direkt um RValue und LValue, sondern um *RValue-Referenzen* und *LValue-Referenzen* (welche in C++03 auch einfach nur Referenzen hießen). Folgende Codezeilen demonstrieren beide Referenzen in deren Syntax:

```
01 int ival = 12345;
02 int& lref = ival; // lref ist eine LValue-Referenz
03 int&& rref;      // rref ist eine RValue-Referenz
```

In Zeile **02** haben wir eine LValue-Referenz mit `lref`. Verwenden Sie hingegen zwei `&&`, dann handelt es sich um ein RValue-Referenz, wie in Zeile **03** zu sehen ist. Allerdings ist diese RValue-Referenz hier noch recht sinnlos und lässt sich so auch gar nicht realisieren. Daher folgt jetzt ein einfaches Beispiel, wie Sie die RValue-Referenzen nutzen können:

```
// kapitel16/rvalue01.cpp
...
01 void function(int& x) { std::cout << x << std::endl; }
02 void function(int&& x) { std::cout << x << std::endl; }
...
03 int ival=10;
04 function(ival);           // function(int& x)
```

```

05 function(10);                // function(int&& x)
06 function(std::move(ival));    // function(int&& x)
...

```

Der erste Funktionsaufruf verwendet die Funktion mit der LValue-Referenz `int&`, weil hier mit `ival` eine Variable übergeben wird, welche eine Adresse im Speicher enthält. Anders sieht es hingegen mit dem Funktionsaufruf in Zeile 05 aus, wo wir einen RValue als Parameter an die Funktion übergeben. Der Wert 10 hat keine feste Adresse im Speicher. Daher wird jetzt auch die Funktion mit der RValue-Referenz `int&&` aufgerufen. Wollen Sie eine LValue- einer RValue-Referenz zuweisen, bietet die Standardbibliothek die Funktion `std::move()` dafür an. In Zeile 06 wird diese Funktion verwendet, um eine LValue- in eine RValue-Referenz umzuwandeln und somit die Funktion mit der RValue-Referenz `int&&` zu verwenden.

Ein Vorteil, den Sie vielleicht in diesem Beispiel gleich erkennen, ist, dass wir uns hiermit eine Funktion ersparen, welche unnötiges Kopieren erfordert. Wir ersparen uns auch Probleme beim Überladen der Funktionen, welche entstehen würden, wenn Sie Folgendes verwendet hätten:

```

void function(int x) { std::cout << x << std::endl; }
void function(int& x) { std::cout << x << std::endl; }

```

Der Compiler würde den Code nicht mehr übersetzen, weil er nicht entscheiden kann, welche Funktion (*by Value* oder *by Reference*) er für den Aufruf `function(ival)` verwenden soll.

Verschieben anstatt Kopieren

Einer der wichtigsten Aspekte für RValue-Referenzen ist somit, unnötiges Kopieren von Objekten zu vermeiden und die Objekte stattdessen zu verschieben. Das Prinzip der Kopiersemantik können Sie sich bei einem Texteditor wie beim Kopieren & Einfügen vorstellen. Die Move-Semantik stellt daher eher das Prinzip des Ausschneidens & Einfügens dar.

Die Standardcontainer und die Move-Semantik

Wie Sie eigene Klassen zum Verschieben schreiben können, haben Sie ja bereits in Abschnitt 7.10, »Klassen für das Verschieben schreiben (C++11)«, erfahren. Was aber noch mehr von Bedeutung ist, dass alle Standardcontainer (inklusive `std::string`) von C++ ab C++11 ebenfalls den neuen Move-Konstruktor und Move-Zuweisungsoperator implementiert haben. Und wenn Sie es in Abschnitt 7.10 nicht ganz verstanden haben, dann hilft es Ihnen vielleicht jetzt, die Semantik anhand eines Standardcontainers nachzuvollziehen. Im Beispiel wollen wir `std::string` dazu heranziehen.

Hierzu ein Ausschnitt der Standardklasse `std::string` mit entsprechenden Konstruktoren und Zuweisungsoperatoren:

```
01 class string {
...
02 public:
...
03     string(const string& rhs);
04     auto operator=(const string& rhs) -> string&;
05     string(string&& rhs) noexcept;
06     auto operator=(string&& rhs) noexcept -> string&;
...
};
```

In den Zeilen **03** und **04** finden Sie die Kopiersemantik mit dem Kopierkonstruktor und dem Kopier-Zuweisungsoperator, welche die klassische (LValue-)Referenz als Parameter enthalten. In den Zeilen **05** und **06** hingegen sehen Sie die neue Move-Semantik mit dem neuen Move-Kopierkonstruktor und dem Move-Zuweisungsoperator, welche sich durch ihre RValue-Referenz mit der `&&`-Syntax davon unterscheiden.

Somit sind quasi die Kopier- und Move-Operationen gewöhnliche Überladungen des Konstruktors oder des Zuweisungsoperators. Auch hier entscheidet der Compiler, welche Implementierung er bei dem Konstruktor bzw. Zuweisungsoperator verwendet, anhand des Parametertyps, der dazu passt. Einige Beispiele zur Demonstration:

```
00 // kapitel16/rvalue02.cpp
...
01 std::string line;
02 std::cout << "Bitte Text eingeben : ";
03 std::getline(std::cin, line);
04 std::cout << line << std::endl;

05 std::string cline1;
06 std::string mline1, mline2, mline3;

07 mline1 = cline1; // Kopierzweisung
08 mline2 = std::move(cline1); // Move-Zuweisung
09 mline3 = std::string("Temporärer String"); //Move-Zuweisung
10 std::string mline4(std::string("Auch temporär")); //Move-Konstr.
11 std::string cline2(mline4); // Kopierkonstruktor
```

In Zeile **07** ist `cline` ein LValue, und daher wird auch der Kopier-Zuweisungsoperator verwendet. In Zeile **08** wandeln wir `cline1` mithilfe der Funktion `std::move()` in einen RValue um, und daher wird hier jetzt der Move-Zuweisungsoperator verwendet. Die Abbildung 16.1 zeigt den Unterschied zwischen Zeile **07** und Zeile **08**.

In Zeile 09 wird in C++11 eine Move-Zuweisung verwendet, weil `std::string("Temporärer String")` ein RValue ist. In C++03 wurde hierfür erst `string(const char*)`, dann `operator=(const string&)` – also erst der normale Konstruktor, dann der Copy-Assign – aufgerufen. Dank der RValue-Referenz ist dies jetzt wesentlich effizienter. In Zeile 10 wird der Move-Konstruktor verwendet, weil auch hier nur ein temporäres Objekt (ein RValue) verwendet wird. In Zeile 11 hingegen wird wieder ein klassischer LValue als Argument und daher wird auch der Kopierkonstruktor verwendet.

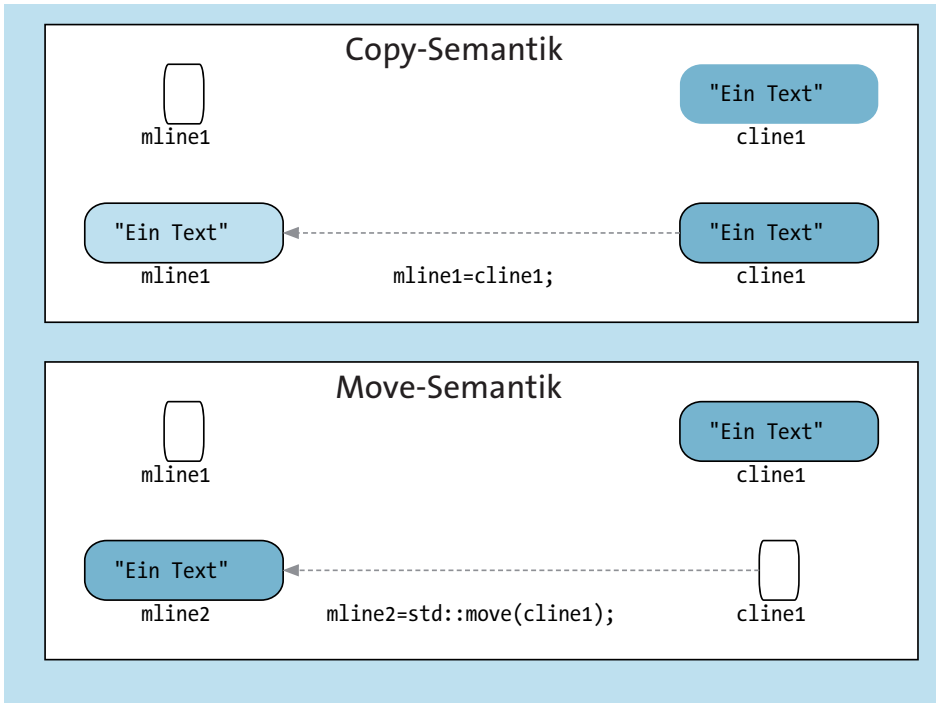


Abbildung 16.1 Diese Abbildung zeigt, glaube ich, recht deutlich den Unterschied zwischen dem Kopieren und Verschieben.

Wie Sie eigene Typen mit der Move-Semantik ausstatten können, haben Sie bereits in Abschnitt 7.10, »Klassen für das Verschieben schreiben (C++11)«, erfahren.

Was passiert mit der Quelle nach dem Verschieben?

Auch wenn es nicht so vorgeschrieben ist, stellen alle Standardcontainer sicher, dass nach einer Move-Operation das Quellobjekt noch weiterhin benutzt werden kann. Wie Sie in Abbildung 16.1 sehen, ist der Inhalt des Quellobjekts dann meistens »leer«. Aus diesem Grund würde ich es auch empfehlen, dies für die eigenen Typen ebenfalls so zu implementieren. In Abschnitt 7.10 haben Sie erfahren, wie Sie bei eigenen Typen die Move-Semantik implementieren können. Hier nochmals die drei Schritte zusammengefasst, um das Verschieben zu implementieren:

1. Ressource des Ziels freigeben
2. Elemente der Quelle nach Ziel verschieben (oder kopieren)
3. Quelle auf einen definierten Zustand setzen

Automatische Optimierung von altem Code dank neuer Move-Semantik

Da die Move-Semantik in allen Standardcontainern implementiert wurde, haben Sie schon mal den Vorteil, dass Ihr Programm in vielen Situationen einen automatischen Performance-Schub erhält, ohne dass Sie den Code ändern müssen. Ein einfaches Beispiel:

```
00 // kapitel16/rvalue03.cpp
...
01 std::vector<int> createVec(int n) {
02     std::vector<int> ivec(n);
03     for(int i=0; i<ivec.size(); ++i) {
04         ivec[i] = i;
05     }
06     return ivec;
07 }
...
08 std::vector<int> data = createVec(10);
```

Sie wissen ja noch von der Einführung in RValues und LValues, dass der Funktionsaufruf in Zeile **08** ein RValue ist. Aber in C++03 gab es noch keine RValue-Unterstützung, und somit wurde von der Funktion `createVec()` in Zeile **06** eine ganze Kopie von `std::vector<int>` zurückgegeben. Zugegeben, solche Dinge in C++03 zurückzugeben, war schon sehr »schwergewichtig«, und man hatte auch hier schon spezielle Tricks mit RVO (*Return Value Optimization*) verwendet.

Trotzdem muss man sich jetzt mit C++11 und dank integrierter RValue-Technik keine Gedanken mehr darum machen. Der Rückgabewert ist temporär, und es wird keine Kopie mehr zurückgegeben, sondern es kommt die Move-Semantik zum Einsatz. Der Compiler übernimmt dies automatisch für uns. Sie müssen lediglich Ihren alten Code mit einem C++11-Compiler erneut übersetzen, um den neuen Performance-Schub in Ihrem Programm zu integrieren.

Wahl zwischen Verschieben oder Kopieren dank `std::move`

Dank der neuen Standardfunktion `std::move()` haben Sie jetzt endlich auch die Wahl, ob Sie nichttemporäre Objekte verschieben oder kopieren wollen. Folgender Codeausschnitt soll den Sachverhalt näher demonstrieren:

```

00 // kapitel16/rvalue04.cpp
...
01 std::vector<std::string> text;
02 std::string line;
03 while( std::getline(std::cin, line)) {
04     if( line == "quit") break; // Abbruchbedingung
05     text.emplace_back(std::move(line));
06 }

```

In diesem Beispiel lesen wir zeilenweise von der Standardeingabe in den String `line` ein und fügen diese Zeile am Ende eines Containers an. Mit `std::move()` in Zeile **05** sorgen Sie dafür, dass die eingelesene Zeile in den Container verschoben statt kopiert wird. Ohne `std::move` würde hier jedes Mal eine Kopie von `line` angefertigt und dann am Ende des Containers angefügt. Hier ist das total unnötig. Wenn die Daten statt von der Standardeingabe von einer Datei kommen, dann macht sich der Performance-Schub dank Verschiebens statt Kopierens erheblich bemerkbar.

16.1.3 Perfect Forwarding

Ebenfalls neu in C++11 wurde das *Perfect Forwarding* implementiert, das mithilfe von RValue-Referenzen realisiert werden kann. Die Idee dahinter ist folgende: Eine Funktion (oder ein Konstruktor) bekommt die Daten als LValue oder RValue und verwendet sie, um eine weitere Funktion passend zu diesen Daten aufzurufen, so dass die Eigenschaften von RValue bzw. LValue erhalten bleiben. Damit die Eigenschaften erhalten bleiben, wurde ein sogenanntes *Reference-Collapsing* eingeführt.

Das Konzept ist ideal, um spezielle Abstraktionen von Klassen- bzw. Funktions-Templates zu erzeugen, welche einfacher und sicherer in deren Anwendung werden. In der Praxis wird dieses Feature wohl vorwiegend in Bibliotheken eingesetzt. Auch die Standardbibliothek macht regen Gebrauch davon. Um beim Thema zu bleiben, soll hier folgendes Beispiel verwendet werden:

```

00 // kapitel16/forward01.cpp
...
01 class Data {
02     std::vector<std::string> text;
03     std::string label;
04 public:
05     Data(std::vector<std::string> &t, std::string& l):
06         text(t), label(l){}
07     Data(std::vector<std::string> &t, std::string&& l):
08         text(t), label(std::move(l)){ }

```

```

07  Data(std::vector<std::string> &&t, std::string& l):
    text(std::move(t)), label(l){}
08  Data(std::vector<std::string> &&t, std::string&& l):
    text(std::move(t)), label(std::move(l)){}
09  };
...
10  std::vector<std::string> text{
    "Die erste Zeile",
    "Die zweite Zeile",
    "Noch eine Zeile",
    "Die letzte Zeile"
    };
11  std::string label("Ein Textlabel");

12  Data(text, label); // Copy, Copy
13  Data(text, std::move(label)); // Copy, Move
14  Data(std::move(text), label); // Move, Copy
15  Data(std::move(text), std::move(label)); // Move, Move
...

```

In diesem Beispiel wurde eine Klasse `Data` erstellt, die auf viele erdenkliche Möglichkeiten der Konstruktoren-Verwendung hin implementiert wurde. Obwohl unsere Klasse nur zwei Eigenschaften beinhaltet, sind hierzu vier verschiedene Konstruktoren nötig, um auch auf eine Kombination von Move- und Copy-Semantiken vorbereitet zu sein. In den Zeilen 12 bis 15 werden verschiedene Aufrufmöglichkeiten demonstriert. Der Aufwand ist enorm, und wollen Sie jetzt noch eine dritte Eigenschaft der Klasse hinzufügen, dann müssen Sie noch mehr Konstruktoren dafür schreiben.

An dieser Stelle können Sie sich jetzt mithilfe von Perfect Forwarding das Leben erheblich leichter machen. Hier zunächst das Beispiel der Klasse `Data`, umgeschrieben für Perfect Forwarding:

```

00  // kapitel16/forward02.cpp
...
01  class Data {
02      std::vector<std::string> text;
03      std::string label;
04  public:
05      template<typename T1, typename T2>
06      Data(T1&& t, T2&& l) :
07          text(std::forward<T1>(t)), label(std::forward<T2>(l)){}
08  };
...

```

Die so geschriebene Klasse erfüllt denselben Zweck wie im Beispiel von Listing *forward01.cpp*. Um zu verstehen, was hier passiert, müssen Sie etwas mehr in die Tiefe gehen.

Reference Collapsing

Damit überhaupt eine entsprechende Instanziierung erzeugt wird, ist eine Technik nötig, die weiß, wann ein LValue mit `&` und wann ein RValue mit `&&` verwendet wird. Hierfür würde mit *Reference Collapsing* ein Regelwerk eingeführt, welches vorschreibt, als was ein Typ betrachtet werden muss. Im Grunde sind es vier Regeln, aber Sie können sich auch einfach nur merken: Wenn mindestens eine der beteiligten Referenzen ein `&` ist, dann ist das Ergebnis immer `&`. Nur wenn beide beteiligten Referenzen `&&` sind, dann bleibt auch ein `&&` bestehen.

Ein einfacher Konstruktor-Aufruf hierzu:

```
Data(text, std::move(label)); // Copy, Move
```

Mit `text` haben Sie einen LValue des Typs `std::string`, daher wird `T1` in `typename` als `T1&` betrachtet. Nach dem Reference Collapsing wird daher daraus ein `std::vector<std::string>&`. Beim Aufruf von `std::forward<T1>(t)` wird also `std::vector<std::string>&` verwendet.

Durch `std::move()` haben Sie `label` in einen RValue konvertiert. Aus `T2` wird also `T2&&`, und gemäß dem Reference Collapsing wird insgesamt ein `std::string&&` daraus. Beim Aufruf von `std::forward<T2>(l)` wird also `std::string&&` verwendet.

Insgesamt wurde das Thema Reference Collapsing hier relativ knapp gehalten, aber Sie müssen einfach nur wissen: Wenn mindestens eine der beteiligten Referenz ein `&` ist, dann ist das Ergebnis immer `&`. Nur wenn beide beteiligten Referenzen `&&` sind, dann bleibt auch ein `&&` bestehen.

In der Praxis kommt das Reference Collapsing eigentlich vorwiegend bei einer Funktions-Template-Anwendung zur Geltung, um den Referenztyp der Argumente zu ermitteln.

std::forward()

Die Funktion `std::forward()` muss jetzt nur noch die Template-Argumente entsprechend als Copy- oder Move-Semantik weiterleiten. Besonders wichtig ist bei `std::forward()` auch, dass Sie immer das Typargument `<T>` mit angeben, weil wir nicht wollen, dass der Compiler den Typ selber herausfinden kann, was nur dazu führen würden, dass in der RValue-Referenz ein LValue gemacht würde.

16.2 Benutzerdefinierte Literale (C++11)

Mit C++11 können Sie jetzt auch benutzerdefinierte Literale erstellen. Sie wissen ja noch aus der Einführung, dass Literale auch Konstanten sind. In C++ können Sie vier verschiedene Literale verwenden:

- ▶ Ganzzahlen wie 100, 255, 65000 usw.
- ▶ Gleitkommazahlen wie 3.1415, 0.03245 usw.
- ▶ Zeichen wie 'A', 'B', 'Z'
- ▶ Strings wie "Das ist ein String-Literal"

Ebenso können Sie für einige Literale ein Suffix verwenden, um den Typ des Literals genauer zu spezifizieren. Verwenden Sie eine Ganzzahl wie 100, dann handelt es sich um ein `int`-Literal. Verwenden Sie hingegen 100u, dann wird das Literal als `unsigned int` interpretiert, und mit 100L wird das Literal als `long`-Wert angesehen. Aber das alles kennen Sie ja bereits aus Ihren ersten Schritten in C++ vom Anfang des Buches.

Mit benutzerdefinierten Literalen der Version C++11 haben wir jetzt die Möglichkeit, unser eigenes Suffix zu erstellen. Aber warum sollten wir das wollen? Betrachten Sie folgende Codezeile:

```
long double weight = 79.5;
```

Was an dieser Codezeile stört, ist, dass man hieran nicht erkennt, um welche Gewichtseinheit es sich hier handelt. Sind es Tonnen, Kilogramm, Gramm oder gar Milligramm – oder eine ganz andere Gewichtseinheit wie Pfund? Nur der Entwickler selbst weiß, was hier für eine Einheit hingehört. Vielleicht wird der Code irgendwo dokumentiert, und Sie können ermitteln, welche Einheit der Entwickler hier festgelegt hat.

Aber wäre es nicht schön, wenn man gleich hier etwas vorfinden würde, wie z.B.:

```
long double weight = 79.5_kg;
```

So etwas ist mit C++11 jetzt dank benutzerdefinierter Literale möglich. Hier müssen Sie nur bestimmen, wie ein Literal interpretiert werden soll, indem Sie ein Mitglied des neuen `operator""` überschreiben. Die Realisierung ist denkbar einfach zu implementieren:

```
00 // kapitel16/userliterals01.cpp
...
01 long double operator "" _kg(long double w) {return w;}
...
02 long double weight = 79.5_kg + 10.6_kg;
03 std::cout << "Gesamtgewicht: " << weight << " kg\n";
```

Das Beispiel können wir noch erweitern. Nehmen wir an, Sie wollen als Maßeinheit Kilogramm verwenden, und Werte mit dem Suffix `_t` für Tonne oder `_g` für Gramm sollen automatisch in die Maßeinheit Kilogramm umgerechnet werden. Auch das ist kein Problem mehr und lässt sich wie folgt realisieren:

```
00 // kapitel16/userliterals02.cpp
...
01 long double operator "" _t(long double w) {return w*1000;}
02 long double operator "" _kg(long double w) {return w;}
03 long double operator "" _g(long double w)
    {return w/1000;};
...
04 long double weight1 = 79.5_kg + 8100.0_g;
05 std::cout << "Gesamtgewicht: " << weight1 << " kg\n";
06 long double weight2 = 1.3_t;
07 std::cout << weight2 << " kg\n"; // = 1300
...
```

In Zeile **04** werden bspw. 79.5 Kilogramm mit 8100 Gramm addiert. Dank des Suffixes `_g`, bei dem wir in Zeile **03** den übergebenen `long double`-Wert durch 1000 teilen (da 1 kg = 1000 g), wird aus 8100.0_g der Wert 8.1 (für Kilogramm). Ähnliches passiert auch in Zeile **06**, wo wir 1.3 Tonnen an `weight2` zuweisen. Dank des Suffixes `_t` wird der Wert 1.3 in Zeile **01** als Parameter übergeben und mit 1000 multipliziert (da 1T = 1000 kg), und zurückgegeben wird dann der `long double`-Wert 1300 (für Kilogramm).

Performance-Schub mit constexpr

Leider hat unser Beispiel noch eine unschöne Seite. Die benutzerdefinierten Literale werden erst zur Laufzeit übersetzt, und das kostet unnötig Rechenzeit. Hier können wir es uns einfach machen, indem wir vor den Funktionen das Schlüsselwort `constexpr` setzen und somit sicherstellen, dass alle Berechnungen in die richtige Einheit bereits zu Kompilierzeit gemacht werden:

```
...
constexpr long double operator "" _t(long double w) {
    return w*1000;
}
constexpr long double operator "" _kg(long double w) {
    return w;
}
constexpr long double operator "" _g(long double w) {
    return w/1000;
}
...
```

Regeln für die Namen

Natürlich sind hier auch Einschränkungen vorhanden, und Sie können nicht auf Teufel komm heraus verwenden, was Sie wollen. Der Bezeichner sollte immer mit einem Unterstrichzeichen `_` beginnen, weil alle anderen Bezeichner für Suffixe künftigen Spracherweiterungen vorbehalten sind. Auch sollten Sie keine zwei Unterstriche am Anfang verwenden, weil die für den Compiler und die Standardbibliothek reserviert sind. Als zweites Zeichen nach dem Unterstrich wird ein Kleinbuchstabe oder eine Ziffer empfohlen.

Stecken Sie benutzerdefinierte Literale in einen Namensraum

Es wird außerdem empfohlen, benutzerdefinierte Literale in einen eigenen Namensraum zu stecken, um künftige Namenskollisionen zu vermeiden, weil diese benutzerdefinierten Literale in der Praxis recht kurze Bezeichner haben.

Erlaubte Argumente für den operator "" (Raw und Cooked)

Als Rückgabewert für den operator "" sind alle Typen möglich. Sie können alle eingebauten Typen von C++ verwenden wie auch benutzerdefinierte Typen. Es ist allerdings nicht möglich, beliebige Typen als Parameter für operator "" zu verwenden. Folgende Typen können Sie als Parameter von benutzerdefinierten Literalen verwenden:

```
// Cooked-Form:
Typ operator "" _suffix (unsigned long long)
Typ operator "" _suffix (long double)
// Raw-Form:
Typ operator "" _suffix (const char*)
Typ operator "" _suffix (const char*, size_t)
Typ operator "" _suffix (const wchar_t*, size_t)
Typ operator "" _suffix (const char16_t*, size_t)
Typ operator "" _suffix (const char32_t*, size_t)
```

Im Beispiel zuvor haben Sie bereits gesehen, wie Sie Zahlenparameter in einer *Cooked*-Form übergeben. Bei einer Cooked-Form stehen die Zahlen gleich in der Form zu Verfügung, um damit weiterarbeiten zu können. Hierfür steht Ihnen mit `unsigned long long` eine Cooked-Version für Ganzzahlen und mit `long double` eine Version für Gleitkommazahlen zur Verfügung. Da jeweils die größten Typen in der Gruppe verwendet wurden, sind Sie damit für alle Ganzzahl- und Gleitkommawerte vorbereitet.

Die anderen Versionen sind in einer *Raw*-Form und somit noch unbearbeitet. Während Ganzzahlen und Gleitkommazahlen sowohl in Cooked- als auch Raw-Form verwendet werden können, steht Ihnen für Strings und Zeichen nur die Raw-Form zur

Verfügung. Bei der Raw-Form müssen Sie sich um das Parsen des Strings kümmern. Zum besseren Verständnis hierzu unser Beispiel Kilogramm nochmals in Raw-Form:

```
00 // kapitel16/userliterals03.cpp
...
01 int operator "" _kg(const char* s, size_t l) {
02     std::string str(s);
03     std::cout << str << std::endl;
04     int val = 0;
05     for(int i=0; i<l; ++i) {
06         if( !(str.at(i) >= '0') && !(str.at(i) <='9') ) {
07             return val;
08         }
09     }
10     std::stringstream conv;
11     conv << str;
12     conv >> val;
13     if(!conv.eof())
14         return 0;
15     return val;
16 }
...
17 int weight1 = "8123"_kg + "1245"_kg;
18 std::cout << "Gesamtgewicht: " << weight1 << " kg\n";
```

Der Aufruf erfolgt als String, gefolgt von dem Suffix, wie Sie in Zeile 17 sehen können. Die Strings "8123" und "1245" müssen Sie dann in den Zeilen 01 bis 16 selbst auswerten. Wir wollen daraus ein `int` machen und prüfen den Inhalt auf Zeichen von '0' bis '9'. Sind alle Zeichen auch Zahlen, dann konvertieren wir den String in ein Integer mithilfe von `std::stringstream` und geben diesen zurück.

Tipp: Verwenden Sie `atoi` oder `stoi` (C++11)

Die Konvertierung von einem String in ein Integer können Sie sich mit den Funktionen `std::atoi()` oder `std::stoi()` (C++11) erheblich einfacher und kürzer machen. Statt der Zeilen 01 bis 16 können Sie auch Folgendes schreiben:

```
int operator "" _kg(const char* s, size_t l) {
    return std::stoi(s);
}
```

Zugeben, für solch ein Beispiel werden Sie eher selten eine Raw-Form verwenden, weil es mit der Cooked-Form leichter ist. Aber Sie könnten hiermit bspw. eine binäre Darstellung wie "10011"_b in eine Ganzzahl konvertieren.

Hierzu soll noch ein letztes Beispiel erstellt werden, welches mithilfe des Suffixes `_b` eine binäre Zahl wie 1011001 in einen dezimalen Wert umwandeln soll. Hierfür benötigen wir ein Raw-Format, weil wir einen String parsen müssen. Des Weiteren soll noch eine Version erstellt werden, welche einen dezimalen Wert wie 254 in einen String mit der binären Nummer des Wertes umwandeln soll. Hierfür erstellen wir einen benutzerdefinierten String mit dem Suffix `_dec`. Dafür können wir ebenfalls eine Cooked-Form für das Argument verwenden. Hier der Quellcode dazu:

```
00 // kapitel16/userliterals04.cpp
...
01 int operator "" _b(const char* s, size_t l) {
02     int decimal{0};
03     for(int i=0; i<l; ++i) {
04         decimal <<= 1;
05         if(s[i] == '1') {decimal+=1;}
06     }
07     return decimal;
08 }

09 std::string operator "" _dec(unsigned long long val) {
10     std::string binary("");
11     do {
12         if( (val & 1) == 0)
13             binary.push_back('0');
14         else
15             binary.push_back('1');
16         val >>= 1;
17     }while( val );
18     std::reverse(binary.begin(), binary.end());
19     return binary;
20 }
...
21 int bin2dez01 = "110011"_b;
22 std::cout << bin2dez01 << std::endl; // 51

23 std::string dez2bin01 = 254_dec;
24 std::cout << dez2bin01 << std::endl; // = 11111110
...
```

Im Beispiel können Sie sehr schön sehen, wie Sie aus dem binären Raw-Format "110011"`_b` den dezimalen Wert 51 mithilfe eines benutzerdefinierten Strings errechnen lassen. Gleiches wurde auch für 254`_dec` realisiert, wo Sie von einer Ganzzahl, an die Sie das Suffix `_dec` anhängen, ein `std::string` in binärem Format zurückbekommen. Hierfür konnten wir auch mit dem Cooked-Format arbeiten.

16.3 Smart Pointer (C++11)

Neben den RValue-Referenzen dürften wohl die neuen *Smart Pointer* zu den bedeutendsten Neuerungen von C++11 gehören. Hierzu gehören drei neue Pointer mit `std::shared_ptr`, `std::weak_ptr` und `std::unique_ptr` aus der Headerdatei `<memory>` (und natürlich dem Namensbereich `std`). Der Vorteil der neuen klugen Zeiger ist, dass sie die Lebenszeit der Ressourcen überwachen, auf die diese verweisen. Damit kann quasi eine automatische Speicherverwaltung verwendet werden (`new` brauchen wir zwar immer noch, aber `delete` fällt damit oft weg). Die Zeiger werden wie normale Zeiger verwendet, nur wird das referenzierte Objekt automatisch freigegeben, wenn Sie dieses nicht mehr benötigen. Darauf gehen wir anschließend noch wesentlich detaillierter ein. In Tabelle 16.1 finden Sie eine erste Gegenüberstellung der neuen Smart Pointer.

Name	Beschreibung
<code>std::shared_ptr</code>	Dieser Zeiger hat einen Referenzzähler auf gemeinsam genutzte Ressourcen und weiß daher selbst, wann er die Ressource freigeben kann.
<code>std::weak_ptr</code>	Wird häufig mit <code>std::shared_ptr</code> verwendet, um mögliche zyklische Referenzen auszulösen, die in Verbindung mit <code>std::shared_ptr</code> auftreten können.
<code>std::unique_ptr</code>	Dieser Zeiger kann nur für eine exklusive Ressource verwendet und kann nicht mit einem anderen Zeiger geteilt werden. Auch kopieren ist nicht möglich. Dieser Zeiger garantiert die Freigabe seiner exklusiven Ressource, wenn der Gültigkeitsbereich verlassen wird. Da dieser Zeiger keine heimliche Move-Semantik enthält, ist er die bessere Alternative für den veralteten Zeiger <code>std::auto_ptr</code> .

Tabelle 16.1 Übersicht zu den neuen Smart Pointern von C++11

RAII (Resource Acquisition Is Initialization)

Die automatische Speicherverwaltung wird mithilfe der RAII-Technik realisiert, welche im Grunde denkbar einfach und trotzdem genial ist. Bei dieser Technik wird die Ressource in einem lokalen Objekt gekapselt und so zur Lebenszeit des Objekts überwacht. Vereinfacht heißt dies, in einem Konstruktor wird die Ressource erzeugt und in einem Destruktor wieder zerstört. Wo die Ressource erzeugt und zerstört wird, hängt dann natürlich vom Gültigkeitsbereich ab. Erzeugen Sie ein Objekt innerhalb eines Anweisungsblocks mit dem Konstruktor, wird die Ressource automatisch wie-

der freigegeben, sobald Sie diesen Anweisungsblock verlassen, weil dann der Destruktor aufgerufen wird.

std::auto_ptr (deprecated)

Bereits C++98 hatte einen »klugen« Zeiger mit `std::auto_ptr`. Dieser Zeiger kann mit dem neuen `std::unique_ptr` verglichen werden. Allerdings hatte `std::auto_ptr` einen Schwachpunkt, wenn ein `std::auto_ptr` kopiert wurde, weil hiermit der komplette Inhalt der Quelle in den Zielzeiger in einer Art Move-Semantik verschoben wurde. Damit konnte man mit diesem Zeiger nicht mit den Standardcontainern arbeiten und man konnte durch falsche Anwendung ein undefiniertes Verhalten erzeugen. Aus diesem Grund erhalten Sie bei neueren Compilern auch eine Warnmeldung, wenn Sie den veralteten `std::auto_ptr` im Programm bei der Übersetzung verwenden. Das Standardkomitee hat `std::auto_ptr` zu *deprecated* erklärt, und daher sollten Sie in künftigen Projekten darauf verzichten und auf `std::unique_ptr` oder `std::shared_ptr` zurückgreifen (abhängig vom Anwendungsfall).

Das folgende Beispiel soll das Problem mit einem `std::auto_ptr` demonstrieren:

```
01 std::auto_ptr<int> ap01(new int(12345));
02 std::auto_ptr<int> ap02(ap01);
03 int val = *ap01; // undefinierter Zugriff
```

In Zeile 01 reservieren wir dynamisch zur Laufzeit Speicher für ein Integer und übergeben dieser Adresse auch gleich den Wert 12345. In Zeile 02 wollten wir eigentlich einen zweiten Zeiger auf diese Adresse verweisen lassen. Aber da `std::auto_ptr` anders arbeitet als ein gewöhnliches `int*` und hier mit einer heimlichen Move-Semantik alles von `ap01` nach `ap02` verschiebt, wird in Zeile 03 ein undefinierter Zugriff gemacht. Der Fehler wurde gerne gemacht, weil man ja irgendwie intuitiv denkt, es funktioniert alles wie bei gewöhnlichen Zeigern:

```
01 int* ip01 = new int(12345);
02 int* ip02(ip01);
03 int val = *ip01; // hier in Ordnung !!!
```

Bei dieser Version mit den rohen Zeigern haben Sie allerdings dann wieder das Problem, sich um die Freigabe des reservierten Speichers kümmern zu müssen. Auch müssen Sie dann aufpassen, weil zwei Zeiger auf denselben Speicherbereich verweisen. Außerdem sind rohe Zeiger in C++ recht spartanisch.

In C++11 haben Sie hierfür jetzt zwei Möglichkeiten. Haben Sie Ressourcen, die Sie mit mehreren Zeigern teilen wollen, wie im eben gesehenen Beispiel, dann können Sie hierfür `std::shared_ptr` verwenden. Wollen Sie vermeiden, dass mehrere Zeiger dieselbe Adresse teilen, dann verwenden Sie einfach `std::unique_ptr`.

16.3.1 Shared Pointer

Werden (bzw. wurden) mehrere rohe Zeiger für eine gemeinsame Ressource verwendet, muss (musste) man sich selbst vor Problemen wie wilden Zeigern (engl. *dangling pointer*, hängende Zeiger) und Speicherlecks (*Memory Leaks*) schützen. Der folgende Codeausschnitt soll diese beiden möglichen Probleme kurz demonstrieren:

```
00 // kapitel16/smartpointer01.cpp
...
01 class Data {
02     std::string label;
03 public:
04     Data(std::string l):label(l) {
05         std::cout << l << " erzeugt\n";
06     }
07     ~Data() { std::cout << label << " zerstört\n";}
08     void printLabel(){std::cout << "Data: " << label
                                << std::endl;}
09     void setLabel(const std::string& s) {label=s;}
10 };
...
11 std::cout << "Programmanfang\n";
12 {
13     Data* d01 = new Data("Buchhaltung");
14     Data* d02(d01);
15     d01->printLabel();
16     delete d01;
17     //...
18     d02->printLabel(); // Hängender Zeiger
19     Data *d03 =new Data("Monatsabrechnung"); //Memory Leak
20 }
21 std::cout << "Programmende\n";
...
```

In Zeile 13 wird Speicher für ein neues Data-Objekt reserviert, welchen der Zeiger d01 verwaltet. In der nächsten Zeile erhält d02 dieselbe Adresse wie d01. Hiermit zeigen beide Zeiger auf dasselbe Speicherobjekt. Die Ausgabe in Zeile 15 ist noch in Ordnung. Mit delete in Zeile 16 wird der Speicherbereich, auf den d01 verweist, freigegeben. Damit ist das Speicherobjekt nicht mehr gültig, und die Ausgabe aus Zeile 18 verwendet einen hängenden Zeiger, der jetzt auch nicht mehr gültig ist, weil das Speicherobjekt ja offiziell in Zeile 16 zerstört wurde. In Zeile 19 erzeugen wir außerdem auch gleich noch ein Speicherleck. Wir fordern einen Speicher an und geben diesen am Ende des Gültigkeitsbereichs (hinter Zeile 20) nicht mehr zurück. Der Speicher wird

zur Laufzeit des Programms nicht mehr an das System zurückgegeben. Das Beispiel demonstriert sehr schön den gefährlichen und ressourcenfressenden Umgang mit rohen Zeigern und warum man in C++ darauf verzichten sollte.

Hängender Zeiger (Dangling Pointer)

Das Schlimme an hängenden Zeigern, wie wir sie in diesem Beispiel absichtlich provoziert haben, ist, dass ein weiterer Zugriff auf den bereits freigegebenen Speicher nicht sofort zu einem Laufzeitfehler führt, da die Rückgabe des Speichers an den Heap nicht immer sofort geschieht. Eine kleine Abhilfe würden Sie in unserem Beispiel schaffen, wenn Sie gleich hinter `delete` beiden Zeigern den `nullptr` zuweisen würden. Dann würde zumindest das Programm mit einer Schutzverletzung des Speichers abstürzen. Aber glücklicherweise hat C++11 jetzt etwas viel Besseres an Bord.

Die Ausgabe des Programms:

```

Programmanfang
Buchhaltung erzeugt
Data: Buchhaltung
Buchhaltung zerstört    // Data "Buchhaltung" wurde zerstört
Data: Buchhaltung      // !!! Hängender Zeiger !!!
Monatsabrechnung erzeugt // Memory Leak
Programmende

```

Dasselbe Beispiel soll jetzt mit `std::shared_ptr` erstellt werden. Hierbei müssen Sie sich keine Sorgen mehr um hängende Zeiger oder Memory Leaks machen. Aber sehen Sie am besten selbst:

```

00 // kapitel16/smartpointer02.cpp
...
01 std::cout << "Programmanfang\n";
02 {
03     std::shared_ptr<Data> d01(new Data("Buchhaltung"));
04     std::shared_ptr<Data> d02(d01);
05     d02->printLabel();
06     d01.reset();
07     //...
08     d02->printLabel();
09     std::shared_ptr<Data> d03(
        new Data("Monatsabrechnung"));
10 }
11 std::cout << "Programmende\n";
...

```

Auch hier reservieren wir in Zeile **03** einen Speicher für ein neues Data-Objekt. Hier wird der Speicherbereich jetzt aber von einem `std::shared_ptr` mit dem Bezeichner `d01` verwaltet. In Zeile **04** bekommt `d02` dieselbe Adresse wie `d01` – nur diesmal ebenfalls mithilfe eines `std::shared_ptr` für die Verwaltung. Der `std::shared_ptr` ist auch sehr angenehm in der Verwendung. In Zeile **05** können Sie sehen, dass Sie damit genauso wie mit einem gewöhnlichen Zeiger auf die `public`-Mitglieder einer Klasse zugreifen können. In Zeile **06** entfernen wir `d01` vom referenzierten Speicherbereich und setzen diesen auf `nullptr`.

Bei rohen Zeigern hätten Sie jetzt mit dem Zugriff in Zeile **08** wieder einen hängenden Zeiger. Aber nicht so bei `std::shared_ptr`. Ein `std::shared_ptr` hat einen internen Referenzzähler, der mitzählt, wie viele Zeiger auf einen bestimmten Bereich verweisen. Solange dieser Referenzzähler nicht auf 0 steht, wird der Speicherbereich auch nicht freigegeben. In diesem Beispiel wird der Speicher automatisch außerhalb des Gültigkeitsbereichs hinter dem Anweisungsblock in Zeile **10** (dank RAII) freigegeben.

Auch wird in Zeile **09** kein Speicherleck mehr erzeugt. Dank des RAII-Idioms wird der Speicher automatisch außerhalb des Gültigkeitsbereichs (hier hinter der Zeile **10**) dem System zurückgegeben.

Das Programm bei der Ausführung:

```

Programmanfang
Buchhaltung erzeugt
Data: Buchhaltung
Data: Buchhaltung
Monatsabrechnung erzeugt
Monatsabrechnung zerstört
Buchhaltung zerstört
Programmende

```

Referenzzähler

Der Hauptzweck von `std::shared_ptr` liegt in der Verwendung von gemeinsamen Ressourcen. Dass hier keine hängenden Zeiger mehr produziert werden können, liegt daran, dass `std::shared_ptr` neben einem Zeiger auf das Objekt auch einen Referenzzähler enthält. Der Referenzzähler wird beim Erzeugen eines `std::shared_ptr` um 1 erhöht und beim Löschen um 1 reduziert. Auch beim Kopieren eines `std::shared_ptr` wird der Referenzzähler um 1 erhöht. Erreicht der Referenzzähler den Wert 0, wird der Speicher automatisch freigegeben. Mithilfe der Methode `use_count()` können Sie ermitteln, wie viele `std::shared_ptr` dieselbe Ressource teilen. Mit der Methode `reset()` hingegen entfernen Sie einen `std::shared_ptr` von der Ressource und weisen diesem intern den `nullptr` zu. Hiermit wird der Referenzzähler ebenfalls um 1 reduziert.

Hierzu ein weiteres Listing, welches diesen Referenzzähler etwas deutlicher in der Praxis demonstrieren soll:

```

00 // kapitel16/smartpointer03.cpp
...
01 class DataPool {
02     std::shared_ptr<Data> dPtr;
03 public:
04     void setData(std::shared_ptr<Data> ptr) {
05         std::cout << "Neue Daten im Datenpool\n";
06         dPtr = ptr;
07     }
08 };
...
09 std::cout << "Programmanfang\n\n";
10 {
11     std::shared_ptr<Data> d01(new Data("Buchhaltung"));
12     std::shared_ptr<Data> d02(d01);
13     std::cout << "d01:" << d01.use_count() << std::endl; //2
14     std::cout << "d02:" << d02.use_count() << "\n\n"; //2
15     DataPool *poolPtr = new DataPool();
16     poolPtr->setData(d01);
17     std::cout << "d01:" << d01.use_count() << std::endl; //0
18     std::cout << "d02:" << d02.use_count() << "\n\n"; //3
19     d01.reset();
20     std::cout << "d01:" << d01.use_count() << std::endl; //0
21     std::cout << "d02:" << d02.use_count() << "\n\n"; //2
22     std::shared_ptr<Data> d03(std::move(d02));
23     std::cout << "d01:" << d01.use_count() << std::endl; //0
24     std::cout << "d02:" << d02.use_count() << std::endl; //0
25     std::cout << "d03:" << d03.use_count() << "\n\n"; //2
26     delete poolPtr;
27     std::cout << "d01:" << d01.use_count() << std::endl; //0
28     std::cout << "d02:" << d02.use_count() << std::endl; //0
29     std::cout << "d03:" << d03.use_count() << "\n\n"; //1
30     d03.reset();
31     std::cout << "d01:" << d01.use_count() << std::endl; //0
32     std::cout << "d02:" << d02.use_count() << std::endl; //0
33     std::cout << "d03:" << d03.use_count() << "\n\n"; //0
34 }
...

```

Durch die Erzeugung eines neuen Objekts in Zeile 11 und das Kopieren der Adresse in Zeile 12 haben beide `std::shared_ptr` `d01` und `d02` jeweils den Referenzzähler auf 2

stehen. In Zeile 15 erzeugen wir ein neues `DataPool`-Objekt und weisen diesem die Adresse von `d01` in Zeile 16 zu, was beide Referenzzähler von `d01` und `d02` jeweils um den Wert 1 inkrementieren lässt. An dieser Stelle verweisen jetzt drei Zeiger (`d01`, `d02` und der Zeiger in der Klasse `DataPool dPtr`) auf denselben Speicherbereich.

In Zeile 19 entbinden wir `d01` vom referenzierten Speicherbereich, womit dieser automatisch auf den `nullptr` gesetzt wird. Der `std::shared_ptr d01` verweist somit auf keine Ressource mehr, aber die anderen beiden Zeiger verweisen nach wie vor auf die Ressource. Durch die Freigabe von `d01` wurde allerdings der Referenzzähler um 1 dekrementiert, und die Ausgabe von `d02` zeigt, dass nur noch zwei `std::shared_ptr` auf denselben Speicherbereich verweisen.

In Zeile 22 verschieben wir jetzt mit `std::move` den kompletten Inhalt von `d02` nach `d03`, wodurch `d02` anschließend ebenfalls frei ist und der Referenzzähler auf 0 gestellt wurde. Dafür verwaltet jetzt allerdings der neue `std::shared_ptr d03` die Ressourcen, und es bleibt dabei, dass immer noch zwei `std::shared_ptr` auf die gemeinsame Ressource zeigen. Jetzt sind es eben `d03` und der `std::shared_ptr` von `DataPool`.

In Zeile 26 wird das Objekt von `poolPtr` mit `delete` zerstört und der Referenzzähler um 1 dekrementiert. Jetzt ist es nur noch `d03`, der alleine auf die Ressource zeigt. In Zeile 30 entbinden wir auch `d03` von der Adresse, womit der Referenzzähler auf 0 gestellt und somit die Ressource gelöscht wird. Auch ohne Zeile 30 wäre die Ressource spätestens am Ende des Anweisungsblocks nach Zeile 34 dank des RAII-Idioms gelöscht worden.

make_shared vs. std::shared_ptr-Konstruktor

Es gibt noch einen zweiten Weg, um einen `std::shared_ptr` zu erzeugen. Anstatt einen `std::shared_ptr`-Konstruktor zu verwenden, können (sollten) Sie `std::make_shared<>` verwenden. Hierzu das Beispiel:

```
00 // kapitel16/smartpointer04.cpp
...
01 std::shared_ptr<Data> d01(new Data("Buchhaltung1"));
02 std::shared_ptr<Data> d02 =
    std::make_shared<Data>("Buchhaltung2");
03 auto d03 = std::make_shared<Data>("Buchhaltung3");
04 Data mydat("Buchhaltung4");
05 auto d04 = std::make_shared<Data>(mydat);
...
```

Die Version in Zeile 01 mit dem gewöhnlichen Konstruktor kennen Sie ja bereits. Die Version erwartet einen Zeiger als Parameter, den wir hier ja mit `new` zurückgeben. Die Version mit `std::make_shared` erwartet Konstruktorargumente zum *Forwarden* als

Parameter, wie Sie hier in den Zeilen 02, 03 und 05 sehen können. Damit ersparen Sie sich auch die etwas umständlichen Konstruktoraufrufe, wie bpsw. in Zeile 01.

Ob Sie nun die Version mit dem Konstruktor wie in Zeile 01 oder der Funktion `std::make_shared` wie in den Zeilen 02, 03 und 05 verwenden, bleibt in der anschließenden Anwendung des `std::shared_ptr` alles beim Alten. Was allerdings die Sicherheit betrifft, ist die Version mit `std::make_shared` wesentlich besser geeignet.

Die Gefahr bei der Version mit dem Konstruktor `std::shared_ptr` in Zeile 01 liegt in der Auswertung von Ausdrücken, weil diese in Schritte getrennt werden – durch Schritte, die Ausnahmen werfen könnten.

Bei der Version mit dem Konstruktor werden drei Arbeiten ausgeführt. Bei der Version in Zeile 01 muss zuerst ein Objekt `Data` erzeugt werden. Dann muss noch ein Objekt `std::shared_ptr` erzeugt werden und zum Schluss muss das Objekt `Data` an den `std::shared_ptr` zugewiesen werden. Die Gefahr hierbei ist, wenn bspw. die Erzeugung des `Data`-Objekts erfolgreich war, aber die Erzeugung des Objekts `std::shared_ptr` fehlschlägt, haben Sie ein Speicherleck, welches nicht mehr behoben werden kann. Verwenden Sie hingegen die Version mit `std::make_shared`, dann werden alle drei Aktionen gleichzeitig ausgeführt, was diese Funktion wesentlich sicherer macht als den Konstruktor.

std::shared_ptr weiter bzw. erneut verwenden

Mit der Methode `reset()` ohne Parameter haben Sie gesehen, wie Sie einen `std::shared_ptr` von einer Ressource entbinden und auf den `nullptr` setzen konnten. Mit dieser Methode können Sie aber auch gleich den `std::shared_ptr` für eine neue Ressource verwenden.

```
01 std::shared_ptr<Data> d01(new Data("Buchhaltung"));
02 d01.reset(new Data("Buchhaltung NEU"));
03 d01->printLabel();
04 d01->reset(); // entspricht d01=nullptr;
```

In Zeile 01 erzeugen wir einen `std::shared_ptr` auf ein `Data`-Objekt, welches wir in Zeile 02 zurücksetzen, um es auf ein neues Objekt `Data` zu setzen. Das alte Objekt `Data` wird dabei zerstört. In Zeile 04 mit einem leeren `reset()` wird dann auch das neue Objekt `Data` aus Zeile 02 zerstört.

Kontrolle von rohen Zeigern an std::shared_ptr übergeben

Es ist durchaus möglich, einen rohen Zeiger einem `std::shared_ptr` zuzuweisen. Allerdings sollten Sie dann Vorsicht bei der Freigabe von Objekten walten lassen. Wenn Sie einen rohen Zeiger in die Obhut eines `std::shared_ptr` geben, dann sollten Sie diesen künftig auch damit verwalten. Betrachten Sie folgendes Beispiel:

```

01 Data* dptr = new Data("Buchhaltung");
02 std::shared_ptr<Data> d01(dptr);
03 d01->printLabel();
04 d01.reset();
05 delete dptr; // zum 2ten mal gelöscht !!!

```

In Zeile **02** geben wir den rohen Zeiger auf ein Data-Objekt in die Obhut eines `std::shared_ptr`. Beachten Sie hierbei allerdings, dass der rohe Zeiger kein `std::shared_ptr` ist und der Referenzzähler erst ab der Erzeugung des `std::shared_ptr` auf 1 gesetzt wird. In Zeile **04** wird daher mit `reset()` der Speicher zerstört und freigegeben, weil hiermit auch der Referenzzähler 0 erreicht hat. Folglich wird mit dem Aufruf von `delete` in Zeile **05** ein bereits freigegebener Speicher erneut freigegeben, was normalerweise das Programm abstürzen lässt.

Das Programm würde aber auch abstürzen, wenn Sie den Speicher wie folgt freigeben:

```

01 Data* dptr = new Data("Buchhaltung");
02 std::shared_ptr<Data> d01(dptr);
03 d01->printLabel();
04 delete dptr;
...

```

Auch hier stürzt das Programm ab, weil der Speicher wieder zweimal freigegeben wird. Einmal zerstören Sie den Speicher manuell mit `delete` in Zeile **04**. Da aber der interne Referenzzähler immer noch auf 1 steht, wird nach dem Verlassen des Gültigkeitsbereichs vom `shared_ptr` `d01` dank des RAII-Idioms der Speicher erneut automatisch freigegeben.

Hier gilt also das, was eingangs zu diesem Abschnitt gesagt wurde: Wenn Sie mit einem `std::shared_ptr` die Kontrolle über einen rohen Zeiger übernehmen, dann sollten Sie die künftige Verwaltung auch mit dem `std::shared_ptr` durchführen.

Rohen Zeiger mit `get()`

Der Abschnitt zuvor wirft dann gleich die Frage auf, was ist, wenn Sie jetzt einen rohen Zeiger von einem `std::shared_ptr` benötigen? Dies kann bspw. nötig sein, wenn bei einer Schnittstelle ein roher Zeiger als Parameter übergeben werden muss? Für solche Zwecke liefert `std::shared_ptr` die Methode `get()`. Das folgende Beispiel soll diese Methode demonstrieren:

```

00 // kapitel16/smartpointer05.cpp
...
01 std::string extractLabel(Data* d) {
02     return d->getLabel();

```

```

03  }
...
04  std::shared_ptr<Data> d01(new Data("Buchhaltung"));
05  std::string lab = extractLabel(d01.get());
06  std::cout << lab << std::endl;

```

Die globale Funktion `extractLabel()` hat einen rohen Zeiger als Parameter, um mithilfe der Methode `getLabel()` (welche jetzt neu unserer Klassen hinzugefügt wurde) den String (hier "Buchhaltung") zu extrahieren. Im Programm wurde aber ein `std::shared_ptr` zur Verwaltung der Daten verwendet, welcher so nicht an die Funktion übergeben werden kann. Hier können Sie jetzt entweder hergehen und im Funktionskopf ebenfalls `std::shared_ptr` als Argument verwenden, aber wenn Sie keinen Einfluss auf diese Schnittstelle haben, dann können Sie auch die Methode `get()` verwenden, wie in Zeile 05 zu sehen ist.

Auch mit Standardcontainer anwendbar

Mit den neuen Smart Pointer ist endlich auch die Verwendung von Standardcontainern (und auch Algorithmen) mit dynamischen Elementen komfortabel nutzbar. Zwar konnten Sie auch mit dem klassischen `new` dynamisch Elemente in einen Container packen, aber hierbei hätten Sie auch dann wieder das Problem, sich um die Freigabe des Speichers und eventuelle hängende Zeiger selbst kümmern zu müssen.

Auch hierbei machen uns die neuen Smart Pointer das Leben erheblich einfacher. Im folgenden Code werden dem Standardcontainer `std::vector` dynamisch zur Laufzeit Elemente zum Vektor mithilfe von `std::make_shared` hinzugefügt:

```

00  // kapitel16/smartpointer06.cpp
...
01  {
02      std::vector<std::shared_ptr<Data>> vecData;
03      vecData.push_back(std::make_shared<Data>(
          "Buchhaltung1"));
04      vecData.push_back(std::make_shared<Data>(
          "Buchhaltung2"));
05      vecData.push_back(std::make_shared<Data>(
          "Buchhaltung3"));

06      std::shared_ptr<Data> sharData(vecData.at(1));

07      for(size_t i=0; i<vecData.size(); ++i) {
08          std::cout << "\n\nReferenzzähler: " ;
09          std::cout << vecData.at(i).use_count()
              << std::endl;

```

```

10     vecData.at(i)->printLabel();
11 }
12 std::cout << std::endl;

13 vecData.at(2).reset();

14 vecData.clear();
15 std::cout << vecData.size() << std::endl; // = 0
...
16 }
...
```

In den Zeilen **03** bis **05** packen wir jeweils dynamisch zur Laufzeit mithilfe von `std::make_shared` neue Data-Objekte in den Vektor. Dank der Semantik eines `std::shared_ptr` müssen wir uns keine Gedanken um die Freigabe des Speichers mehr machen. In Zeile **06** lassen wir einen `std::shared_ptr` auf das zweite Element im Vektor verweisen. Hierdurch erhöht sich der Referenzzähler auf 2, wie die Ausgabe aller Elemente mit Referenzzähler in den Zeilen **07** bis **11** demonstrieren soll.

In Zeile **13** löschen wir das dritte Element im Vektor manuell und setzen den `std::shared_ptr` damit auf den `nullptr`. Auch der Referenzzähler vom dritten Element ist damit bei 0. Mit Zeile **14** löschen wir alle Elemente im Standardcontainer, abgesehen vom zweiten Element, weil hier noch von Zeile **06** ein `std::shared_ptr` verweist. Neben einem Speicherleck sind wir hiermit also auch vor hängenden Zeigern geschützt. Das Element des zweiten Elements im »leeren« (!) Container wird erst außerhalb des Gültigkeitsbereichs hinter Zeile **16** automatisch freigegeben – genauso wie auch der gemeinsame Zeiger von Zeile **06**.

Beachten Sie, dass mit `clear()` in Zeile **14** der Vektor tatsächlich leer ist, wie die Ausgabe von `size()` in Zeile **15** beweisen soll. Aber da die Gefahr hier noch vorhanden ist, dass jemand den Zeiger `sharData` aus Zeile **06** verwenden könnte, hebt `std::shared_ptr` die Ressource zur Sicherheit bis zum Ende des Gültigkeitsbereichs auf, wo dieser Zeiger dann offiziell keinen Schaden mehr anrichten kann. Ermöglicht wird dies natürlich wieder durch den Referenzzähler.

Eigene Löschfunktion

In manchen Fällen kann eine eigene Löschfunktion (*Custom Deleter*) benötigt werden. In diesem Fall können Sie einem `std::shared_ptr` (und auch `std::unique_ptr`) mit einer solchen Löschfunktion »ausstatten«. Hierzu müssen Sie eine aufrufbare Einheit definieren, was eine globale Funktion, ein Funktionsobjekt oder eine Lambda-Funktion sein kann.

Wenn Sie `std::shared_ptr` (oder auch `std::weak_ptr`) mit einer Löschfunktion ausgestattet haben, wird diese aufrufbare Einheit (wenn der Destruktor an der Reihe ist) mit dem Zeiger der Ressource als Parameter aufgerufen – anstatt wie gewöhnlich `delete`. Es gibt sogar Fälle, in denen Sie mit einer benutzerdefinierten Löschfunktion ein Speicherleck vermeiden.

Als nicht ganz ernst gemeintes Beispiel können Sie folgendes Codebeispiel testen:

```
std::shared_ptr<Data> shptr(new Data[5]);
```

Wenn Sie dieses Beispiel aufrufen, werden fünf `Data`-Objekte erzeugt und am Ende des Gültigkeitsbereichs nur eines wieder freigegeben – allerdings wird hierbei beim Zerstören nur `delete` anstatt `delete[]` aufgerufen, wodurch eben nur das erste Objekt gelöscht würde und die restlichen zwei im Speicher erhalten blieben – auf manchen System stürzt das Beispiel auch ab, weil versucht wird, mehrmals `delete` aufzurufen (also mehrmals dasselbe Objekt). Dem Problem könnten Sie hier Abhilfe schaffen, indem Sie eine benutzerdefinierte (Lambda-)Funktion einbauen, welche das jeweilige Objekte mit `delete []` zerstört. In der Praxis sieht dies wie folgt aus:

```
std::shared_ptr<Data> shptr(new Data[5],
    [](Data *d){delete[]d; } );
```

Führen Sie das Beispiel erneut aus, werden alle fünf Objekte zerstört, sobald der Gültigkeitsbereich verlassen wird.

Vielleicht noch ein etwas sinnvollerer Beispiel, um unser Listing *smartpointer06.cpp* um eine benutzerdefinierte Löschfunktion zu erweitern. In diesem Beispiel wird wieder eine Lambda-Funktion dafür verwendet:

```
00 // kapitel16/smartpointer07.cpp
...
01 std::ofstream logfile("data.log");
02 std::vector<std::shared_ptr<Data>> vecData;
03 vecData.push_back(std::shared_ptr<Data>(
    new Data("Buchhaltung1"),
    [&](Data *t) {
        logfile << t->getLabel() << " gelöscht\n";
        delete t; }));
04 vecData.push_back(std::shared_ptr<Data>(
    new Data("Buchhaltung2"),
    [&](Data *t) {
        logfile << t->getLabel() << " gelöscht\n";
        delete t; }));
05 vecData.push_back(std::shared_ptr<Data>(
    new Data("Buchhaltung3"),
```

```
[&](Data *t) {
    logfile << t->getLabel() << " gelöscht\n";
    delete t;});
...

```

Bei allen drei benutzerdefinierten Löschfunktionen übernehmen wir jetzt das Löschen mit `delete` manuell und schreiben den Löschvorgang auch noch in ein extra Logfile mit.

Auch mit `std::unique_ptr` realisierbar

Natürlich lassen sich die Standardcontainer und benutzerdefinierten Löschfunktionen auch auf `std::unique_ptr` verwenden. Ohnehin sind sich die beiden Smart Pointer `std::unique_ptr` und `std::shared_ptr` extrem ähnlich, außer dass der eine Smart Pointer eben seine Ressourcen teilt und der andere nicht. Aber auf die Unterschiede der beiden Pointer soll noch in Abschnitt 16.3.3, »Unique Pointer«, zu `std::unique_ptr` genauer eingegangen werden.

16.3.2 Weak Pointer

Bei `std::shared_ptr` wurde behauptet, dass wir uns niemals mehr Gedanken um Speicherlecks (engl. *Memory Leaks*) machen müssten. Die Aussage war aber nicht 100 %ig korrekt. Es gibt bei den `std::shared_ptr` ein spezielles Problem, dass als zyklische Referenzen (engl. *cyclic reference*) bezeichnet wird. Das folgende Listing soll Ihnen eine solche zyklische Referenz in der Praxis demonstrieren:

```
00 // kapitel16/smartpointer08.cpp
...
01 class Data {
02     std::string label;
03     std::shared_ptr<Data> attachPtr;
04 public:
05     Data(std::string l):label(l) {
06         std::cout << l << " erzeugt\n";
07     }
08     ~Data() { std::cout << label << " zerstört\n";}
09     std::string getLabel() {return label;}
10     void attachData(std::shared_ptr<Data> a) {
11         attachPtr = a;
12     }
13     void getattachedData() {
14         if(attachPtr != nullptr)
15             std::cout << label << "->"

```

```

        << attachPtr->getLabel() << std::endl;
16     }
17 };
...
18 std::vector<std::shared_ptr<Data>> vecData;
19 vecData.push_back(std::shared_ptr<Data>(
    new Data("Buchhaltung1")));
20 vecData.push_back(std::shared_ptr<Data>(
    new Data("Buchhaltung2")));

21 vecData.at(0)->attachData(vecData.at(1));
22 vecData.at(1)->attachData(vecData.at(0));

23 vecData.at(0)->getattachedData();
24 vecData.at(1)->getattachedData();

```

Hier verwenden wir erneut unsere Klasse `Data`. Jetzt enthält die Klasse noch eine zusätzliche Eigenschaft als `std::shared_ptr` in Zeile 03. Diese Eigenschaft wollen wir dazu verwenden, um verschiedene `Data`-Objekte miteinander »verknüpfen« zu können – als Datenanhang, wenn Sie so wollen. Die Funktion zum virtuellen Anhängen eines `Data`-Objekts wurde in den Zeilen 10 bis 12 definiert. Mit welchem `Data`-Objekt das aktuelle Objekt verknüpft wurde, können Sie mit der Methode `getattachedData()` in den Zeilen 13 bis 16 ausgeben lassen.

Jetzt zu unserem Problem mit den zyklischen Referenzen. In den Zeilen 19 und 20 erzeugen wir zwei neue `Data`-Objekte vom Heap und fügen diese jeweils am Ende des Vektors `vecData` hinzu. In Zeile 21 hängen wir an das erste `Data`-Objekt im Container (hier "Buchhaltung1") das zweite Objekt an (hier "Buchhaltung2") – also die Eigenschaft `std::shared_ptr` vom ersten Objekt im Container verweist auf das zweite Objekt. So weit, so gut und immer noch kein Problem. Erst wenn wir jetzt in Zeile 22 dasselbe in die umgekehrte Richtung machen, wird es knifflig, weil jetzt das zweite Objekt im Container einen Verweis auf das erste Objekt im Container enthält. Die Ausgabe mit den Zeilen 23 und 24 bestätigt dies auch.

Das Programm bei der Ausführung:

```

Buchhaltung1 erzeugt
Buchhaltung2 erzeugt
Buchhaltung1->Buchhaltung2
Buchhaltung2->Buchhaltung1
Programmende

```

Das Erste, was hierbei gleich auffällt, ist, dass hier kein Destruktor aufgerufen wird. Wir haben somit ein Speicherleck geschaffen, und der Grund ist die zyklische Referenz.

renz zwischen den beiden Objekten. Beim Versuch, das erste Data-Objekt freizugeben, kommt `std::shared_ptr` nicht weiter, weil noch ein zweiter `std::shared_ptr` vorhanden ist, der auf das zweite Data-Objekt verweist. Andersherum dasselbe Spiel. Das zweite Data-Objekt kann auch nicht gelöscht werden, weil ein `std::shared_ptr` auf das erste Objekt verweist. Die `std::shared_ptr` stecken in einer zyklischen Referenz – zwei Objekte, die sich gegenseitig referenzieren –, aus der sie allein nicht mehr herauskommen.

Einfaches Beispiel

In der Praxis ist es häufig nicht immer so eindeutig, zyklische Referenzen zu lokalisieren. Bei einem komplexeren Code sollten Sie diesen immer etwas genauer unter die Lupe nehmen. Ganz besonders dann, wenn Sie `std::shared_ptr` verwenden.

Für diesen Zweck wurde `std::weak_ptr` eingeführt. Dieser schwache Zeiger ist eine Erweiterung für `std::shared_ptr`, um die eben gezeigten zyklischen Referenzen aufzubrechen. Mit `std::weak_ptr` können Sie quasi einen unsichtbaren Zugriff auf einen `std::shared_ptr` durchführen. Intern besitzt ein `std::weak_ptr` **keinen** Referenzzähler, der inkrementiert wird. `std::weak_ptr` ist allein für sich ohnehin recht nutzlos und bekommt erst zusammen mit `std::shared_ptr` seinen Sinn und Zweck. Ein `std::weak_ptr` besitzt nicht mal eine Ressource, sondern leiht sich diese von `std::shared_ptr` aus. So ist es mit einem `std::weak_ptr` nicht möglich, die Ressource zu löschen.

Ebenso ist es nicht einfach so möglich, eine `std::weak_ptr` wie einen gewöhnlichen Zeiger auf Methoden der Ressource zu verwenden. Wenn Sie auf die Ressourcen des `std::weak_ptr` zugreifen wollen, müssen Sie diesen zuvor mit `lock()` sperren. Die `lock()`-Funktion macht aus einem `std::shared_ptr` einen `std::weak_ptr`. Damit wird sichergestellt, dass der `std::weak_ptr` auf ein gültiges Objekt verweist.

In unserem Beispiel reichen zwei Codezeilen aus, um das Problem mit der zyklischen Referenz zu beheben:

```
00 // kapitel16/smartpointer09.cpp
...
01 class Data {
02     std::string label;
03     std::weak_ptr<Data> attachPtr;
04 public:
05     Data(std::string l):label(l) {
06         std::cout << l << " erzeugt\n";
07     }
08     ~Data() { std::cout << label << " zerstört\n";}
09     std::string getLabel() {return label;}
```



```

10 void attachData(std::shared_ptr<Data> a) {
11     attachPtr = a;
12 }
13 void getattachedData() {
14     std::cout << label << "->"
        << attachPtr.lock()->getLabel() << std::endl;
15 }
16 };

```

Das Programm jetzt bei der Ausführung mit `std::weak_ptr`:

```

Buchhaltung1 erzeugt
Buchhaltung2 erzeugt
Buchhaltung1->Buchhaltung2
Buchhaltung2->Buchhaltung1
Programmende
Buchhaltung2 zerstört
Buchhaltung1 zerstört

```

Jetzt wird am Ende auch ordentlich wieder der Speicher freigegeben.

Ein kleines Problem könnte noch bei der Methode `getattachedData()` auftreten, wenn der `std::weak_ptr` ein leerer Zeiger ist. In dem Fall würde die Methode `lock()` in Zeile 14 eine Ausnahme werfen. Wollen Sie dies vermeiden, können Sie den `std::weak_ptr` mit der Methode `expired()` testen. Wenn der `std::weak_ptr` nicht »zerfallen« (= *expired*) ist, dann handelt es sich um keinen leeren Zeiger:

```

...
void getattachedData() {
    if( !attachPtr.expired())
        std::cout << label << "->"
            << attachPtr.lock()->getLabel() << std::endl;
}
...

```

use_count()

Auch `std::weak_ptr` unterstützt die Methode `use_count()`, um auszugeben, wie viele `std::shared_ptr` auf das Objekt verweisen.

16.3.3 Unique Pointer

Der `std::unique_ptr` ist dem `std::shared_ptr` in seiner Verwendung und auch seinen Methoden sehr ähnlich. Allerdings besitzt ein `std::unique_ptr` seine Ressource

exklusiv und teilt diese mit keinem anderen Zeiger mehr. Die Ressource wird auch hier freigegeben, wenn der `std::unique_ptr` seinen Gültigkeitsbereich verlässt. Neben der Ressource, die dann freigegeben wird, wird selbstverständlich auch der `std::unique_ptr` zerstört.

Ein `std::unique_ptr` kann auch nicht kopiert werden. Die einzige Möglichkeit, einen `std::unique_ptr` weiterzugeben, ist das Verschieben mit `std::move`. Außerdem ist `std::unique_ptr` mehr als ein würdiger Nachfolger für den zu *deprecated* erklärten `std::auto_ptr`.

Leichtgewichtiger Smart Pointer

Ein `std::unique_ptr` muss erheblich weniger Ballast mit sich herumtragen als ein `std::shared_ptr` und wird daher gerne auch als »Light Weight Smart Pointer« bezeichnet. Wenn Sie also viele Smart Pointer in einem Programm benötigen und diese nicht mit anderen Zeigern geteilt werden müssen, dann sollte `std::unique_ptr` immer die erste Wahl sein.

Das folgende Listing demonstriert den alten und beschwerlichen Weg mithilfe von `new` und `delete`, um zur Laufzeit einen Speicher anzufordern und diesen am Ende wieder freizugeben:

```
00 // kapitel16/smartpointer10.cpp
...
01 class Data {
02     std::string label;
03 public:
04     Data(std::string l=""):label(l) {
05         std::cout << l << " erzeugt\n";
06     }
07     ~Data() {
08         std::cout << label << " zerstört\n";
09     }
10     std::string getLabel() {return label;}
11 };

12 void workData() {
13     Data* dPtr = new Data("Buchhaltung");
14     std::cout << "Daten in Data: " << dPtr->getLabel()
15         << std::endl;
16     // Weitere Arbeiten ...
17     delete dPtr;
18 }
```

```
...
18 workData(); // Funktionsaufruf
...
```

In diesem Beispiel verwenden wir eine Funktion `workData()`, um in Zeile 13 einen Speicher für ein `Data`-Objekt vom Heap anzufordern. Dann arbeiten wir mit dem `Data`-Objekt, und am Ende in Zeile 16 wird der Speicher mit `delete` wieder ordentlich freigegeben – ein klassischer alter C++-Code eben. Allerdings hat die Vergangenheit gezeigt, dass dort gerne mal `delete` vergessen wurde und so Speicherlecks entstanden sind. Bedenken Sie, dass ein Code in der Regel nicht immer so einfach und übersichtlich ist, wie hier abgedruckt.

Neben dem menschlichen Versagen (»Vergessen«), ein `delete` zu verwenden, hat dieser Code auch noch eine zweite Schwachstelle, und zwar wenn eine Ausnahme geworfen wird zwischen dem Bereich, an dem Sie einen Speicher vom Heap mit `new` anfordern, und dem, wenn Sie ihn mit `delete` wieder freigeben. Wenn also beim Kommentar »Weitere Arbeiten« eine Ausnahme geworfen würde, ist es ebenfalls nicht mehr möglich, diese Speicher freizugeben – auch nicht mehr, wenn Sie eine Ausnahmebehandlung eingerichtet haben.

Für beide Fälle können Sie sich jetzt mit `std::unique_ptr` rüsten. Hierzu benötigen Sie erneut die Funktion `workData()`, jetzt aber mit `std::unique_ptr`:

```
00 // kapitel16/smartpointer11.cpp
...
01 void workData() {
02     std::unique_ptr<Data>dPtr(new Data("Buchhaltung"));
03     std::cout << "Daten in Data: " << dPtr->getLabel()
               << std::endl;
04     // Weitere Arbeiten
05 }
```

Der Code macht dasselbe wie der Code in Listing *smartpointer10.cpp* zuvor. Nur müssen Sie sich hier nicht mehr um die Freigabe des Speichers mit `delete` kümmern. Das übernimmt `std::unique_ptr` automatisch für uns, sobald der Gültigkeitsbereich verlassen wird. Das gilt jetzt auch, wenn eine Ausnahme geworfen wird. Auch dann kümmert sich `std::unique_ptr` darum, dass der Destruktor aufgerufen wird. Natürlich hätten Sie für dieses Beispiel auch `std::shared_ptr` verwenden können. Dieser Pointer hätte denselben Effekt gehabt, nur ist ein `std::shared_ptr` für solch einen einfachen Zweck zu *schwergewichtig*.

Weitergabe von `std::unique_ptr`

Es wurde bereits eingangs erwähnt, dass ein `std::unique_ptr` nicht kopiert werden kann und exklusiv seine Ressource für sich verwaltet. Wenn Sie jetzt einen

`std::unique_ptr` weitergeben wollen/müssen, können Sie entweder die Ressource mit der Methode `release()` aufgeben und einem rohen Zeiger zuweisen, oder Sie verschieben den `std::unique_ptr` mit `std::move()`.

Zuerst das Beispiel mit `release()`:

```
00 // kapitel16/smartpointer12.cpp
...
01 void processData(Data* ptr) {
02     std::cout << "Label: " << ptr->getLabel() << std::endl;
03 }

04 void workData() {
05     std::unique_ptr<Data>dPtr(new Data("Buchhaltung"));
06     processData(dPtr.release());
07 }
...
08 workData();
...
```

In Zeile **06** geben wir die Ressource mit `release()` auf und übergeben einen rohen `Data`-Zeiger an die Funktion `processData`, die eben einen solchen Zeiger als Argument erwartet. Wenn Sie das Beispiel ausführen, werden Sie allerdings feststellen, dass die Ressource nicht mehr automatisch freigegeben wird und wir wieder ein Speicherleck erzeugt haben. Denn sobald Sie mit `release()` den Zeiger von `std::unique_ptr` aufgeben, sind Sie wieder selbst für die Freigabe des Speichers verantwortlich. Im Beispiel können Sie den Speicher manuell hinter Zeile **02** mit `delete` freigeben oder am Ende der Funktion `workData()`, was wie folgt aussehen könnte:

```
00 // kapitel16/smartpointer12.cpp
...
01 void workData() {
02     std::unique_ptr<Data>dPtr(new Data("Buchhaltung"));
03     Data *ptr = dPtr.release();
04     processData(ptr);
05     delete ptr;
06 }
...
```

Hier holen wir in Zeile **03** mit einem rohen Zeiger die aufzugebende Ressource ab und übergeben die Adresse noch an `processData()` zur Arbeit. Wenn `processData()` fertig ist mit der Ausführung, geben wir mit `delete` in Zeile **05** die Ressource frei und haben das Speicherleck wieder behoben.

Das Aufgeben einer Ressource von `std::unique_ptr` ist natürlich nicht schön, weil man dann auch gleich auf einen rohen Zeiger oder einen `std::shared_ptr` hätte

zurückgreifen können. Eine zweite Möglichkeit, einen `std::unique_ptr` weiterzugeben, ist es, diesen mit `std::move()` zu verschieben. Aber auch hier müssen Sie aufpassen, weil Sie damit den ursprünglichen Eigentümer aufgeben und die Ressource einen neuen exklusiven Eigentümer erhält. Betrachten Sie folgendes Beispiel:

```
00 // kapitel16/smartpointer13.cpp
...
01 void processData(std::unique_ptr<Data> ptr) {
02     std::cout << "Label: " << ptr->getLabel() << std::endl;
03 } // <- Freigabe der Ressource

04 void workData() {
05     std::unique_ptr<Data>dPtr(new Data("Buchhaltung"));
06     processData(std::move(dPtr));
07 }
...
```

Die einzige Möglichkeit, den `std::unique_ptr` auch als solchen weiterzugeben, funktioniert nur über `std::move()`, wie dies in Zeile **06** zu sehen ist, wo wir die Ressource, auf die der `std::unique_ptr` `dPtr` verweist, an das Funktionsargument `ptr` verschieben, welches ebenfalls ein `std::unique_ptr` ist. Durch diese Verschiebeaktion ist jetzt `ptr` in der Funktion `processData()` der exklusive Besitzer der Ressource. `dPtr` ist jetzt keine Ressource mehr. Dass `ptr` von Zeile **01** jetzt der exklusive Besitzer der Ressource ist, bedeutet auch, dass diese Ressource mit dem Ende der Funktion `processData()` hinter Zeile **03** automatisch freigegeben wird – nicht, wie vielleicht vermutet, hinter Zeile **07** nach der Funktion `workData()`.

std::unique_ptr zurückgeben

Ähnlich ist dies bei der Rückgabe von einem `std::unique_ptr`, was folgendes Beispiel zeigen soll:

```
00 // kapitel16/smartpointer14.cpp
...
01 std::unique_ptr<Data> newData(const std::string& lab) {
02     std::unique_ptr<Data> ptr(new Data(lab));
03     return ptr;
04 }

05 void workData() {
06     std::unique_ptr<Data>dPtr = newData("Buchhaltung");
...
07 }
...
```

Die Funktion `newData` erzeugt ein neues `Data`-Objekt vom Heap und gibt dieses zurück. Da der `std::unique_ptr` *by Value* zurückgegeben wird, wird automatisch in Zeile 03 die neue Move-Semantik verwendet. `ptr` wird hier dann quasi nach `dPtr` verschoben.

std::unique_ptr manuell zerstören

Wie schon mit `std::shared_ptr` können Sie `std::unique_ptr` mit `reset()` manuell zerstören oder eben zerstören und gleich wieder mit einer neuen Ressource versehen. Ein einfaches Beispiel dazu:

```
00 // kapitel16/smartpointer15.cpp
...
01 void workData() {
02     std::unique_ptr<Data>dPtr(new Data("Buchhaltung"));
...
03     dPtr.reset(new Data("Neue Buchhaltung"));
...
04     dPtr.reset(); // entspricht: dPtr=nullptr;
05     if(dPtr==nullptr) {
06         std::cout << "dPtr ist leer\n";
07     }
08 }
...
```

In Zeile 02 erzeugen wir ein neues `Data`-Objekt vom Heap, welches unser `std::unique_ptr` exklusiv verwaltet. In Zeile 03 geben wir den Speicher zurück und erzeugen ein neues `Data`-Objekt vom Heap, welches wiederum von `dPtr` verwaltet wird. In Zeile 04 hingegen geben wir den Speicher wieder frei, und `dPtr` erhält den `nullptr`.

Spezialisierung von Arrays

Sicherlich erinnern Sie sich noch an das folgende Beispiel:

```
std::shared_ptr<Data> shptr(new Data[5]);
```

Hier wurden mit einem `std::shared_ptr` fünf `Data`-Objekte erzeugt, aber nur das erste wieder freigegeben. Daher mussten wir hier eine benutzerdefinierte Löschfunktion schreiben.

Bei `std::unique_ptr` können Sie auf eine benutzerdefinierte Löschfunktion verzichten, weil dieser Smart Pointer eine Spezialisierung von Arrays mit `class unique_ptr<T[]>` enthält. Dies kann wie folgt aussehen:

```
std::unique_ptr<Data[]>dPtr(new Data[5]);
```

Dank dieser Spezialisierung werden fünf Objekte erzeugt und auch wieder fünf Objekte zerstört.

std::unique_ptr als Eigenschaft einer Klasse

Der Vorteil von `std::unique_ptr` macht auch nicht Halt vor Eigenschaften von Klassen. Auch hier haben Sie den Vorteil, dass Sie sich nicht mehr um die Freigabe des Speichers kümmern müssen. Auch im Falle einer Ausnahme kümmert sich `std::unique_ptr` um unsere Ressource.

std::make_unique erst mit C++14

An dieser Stelle vermisst man dann doch eine Komfortfunktion wie `std::make_shared()` bei `std::shared_ptr`. Bei `std::unique_ptr` wurde diese Funktion in Version C++11 noch nicht implementiert und ist erst für C++14 geplant.

Natürlich auch für Standardcontainer

`std::unique_ptr` lässt sich genauso für Standardcontainer und Algorithmen verwenden wie `std::shared_ptr`, allerdings immer mit der Einschränkung, dass bei `std::unique_ptr` nur mit der Move-Semantik verschoben werden kann. Es wird keine Copy-Semantik unterstützt. Es wird gerne mal vergessen, dass ein `std::unique_ptr` sehr exklusiv auf eine Ressource achtet und es dann eben bei einfachen Standardcontainer-Operationen zu einer Fehlermeldung kommt. Folgendes ist bspw. nicht möglich:

```
01 std::vector<std::unique_ptr<Data>> vecData;
02 vecData.push_back(std::unique_ptr<Data>(
    new Data("Buchhaltung1")));
03 std::unique_ptr<Data> ptr(new Data("Buchhaltung2"));
04 vecData.push_back(ptr); // Kopieren nicht möglich
```

In Zeile **04** wird versucht, mit `push_back()` ein in Zeile **03** erzeugtes `Data`-Objekt in den Vektor zu kopieren. Hier meldet sich gleich der Compiler mit einer Fehlermeldung. Damit dies in diesem Beispiel auch klappt, müssen Sie die Ressource mit `std::move()` verschieben:

```
04 vecData.push_back(std::move(ptr)); // Jetzt OK
```

16.3.4 Weiteres zu den Smart Pointern

Zwei Dinge sollen hier noch angemerkt werden, welche in Verbindung mit Erzeugung und Zerstörung von Ressourcen von Bedeutung sind.

Erzeugung der Ressource

Es wird empfohlen, die Ressource gleich beim Konstruktoraufwurf von `std::shared_ptr` bzw. `std::unique_ptr` zu erzeugen. Hierbei haben Sie den Vorteil, dass der Konstruktor die Ressource auch wieder freigibt, wenn eine Ausnahme geworfen wird. Ein einfaches Beispiel, wie Sie es nicht machen sollten:

```
01 Data* d = new Data("Buchhaltung");
02 // Weitere Arbeiten
03 std::unique_ptr<Data> uptr(d);
```

Wenn jetzt nach der Speicheranforderung (Zeile 01) und der Übergabe der Ressource an den Smart Pointer in Zeile 03 eine Ausnahme geworfen wird, kann der Speicher nicht mehr an das System zurückgegeben werden. Auch die Speicheranforderung in Zeile 01 selbst könnte eine Ausnahme werfen (`std::bad_alloc`).

Bei solchen Beispielen wird daher empfohlen, die Ressource direkt im Konstruktor zu erzeugen:

```
std::unique_ptr<Data> uptr(new Data("Buchhaltung"));
```

Jetzt wird auch die Freigabe des Speichers von `std::unique_ptr` sichergestellt.

Alternativ hätten Sie auch eine Ausnahmenbehandlung einrichten können, die sich dann mit diesem Problem herumschlägt:

```
...
01 Data* d=nullptr;
02 try{
03     d = new Data("Buchhaltung");
04     std::unique_ptr<Data> ptr(d);
05 }
06 catch(...){
07     delete d; // Speicher im Notfall freigeben
08 }
...
```

Eigene Löschfunktion bei `std::unique_ptr`

Das Thema eigene Löschfunktion wurde ja bereits bei `std::shared_ptr` behandelt. Allerdings unterscheiden sich hier die beiden Smart Pointer ein wenig. Bei `std::unique_ptr` muss zusätzlich noch der Typ zwischen den spitzen Klammern geschrieben werden. Das folgende Listing soll den kleinen Unterschied von eigenen Löschfunktionen zwischen `std::shared_ptr` und `std::unique_ptr` demonstrieren:


```

00 // kapitel16/smartpointer15.cpp
...
01 class DeleteData {
02 public:
03     void operator()(Data* d) {
04         if(d) delete d;
05         std::cout << "DeleteData war aktiv\n";
06     }
07 };
...
08 std::shared_ptr<Data> sptr(
    new Data("Buchhaltung1"), DeleteData());
09 std::unique_ptr<Data, DeleteData> uptr(
    new Data("Buchhaltung2"), DeleteData());
...

```

Hier sehen Sie sehr schön zum Vergleich die unterschiedliche Verwendung von eigenen Löschfunktionen zwischen `std::shared_ptr` und `std::unique_ptr` in den Zeilen 08 und 09.

16.4 Zeitbibliothek

Neu in C++11 wurde die Bibliothek `chrono` eingeführt, mit der Sie u.a. mit Zeit und Datum arbeiten können. Es gab (und gibt) schon andere Bibliotheken, mit denen mit Zeit gearbeitet werden konnte, aber hierbei gab es schon immer Probleme mit der Präzision der Zeiteinheiten. Meistens hatte man gerade mal Sekunden (oder Ticks) als Zeiteinheit, aber in der Computer-Praxis werden hierfür auch Milli-, Micro- oder gar Nanosekunden benötigt. Mit der Bibliothek `chrono` sind solche kleinen Zeiteinheiten jetzt endlich möglich, und zwar portierbar.

16.4.1 Zeitgeber (Clock)

Jeder kennt einen Zeitgeber wie eine Uhr, welche Ihnen den aktuellen Zeitpunkt zurückgibt. Dies gilt auch für den Zeitgeber in C++11. Dieser enthält ebenfalls eine Zeitdauer mit einem Zeitpunkt, um den aktuellen Zeitpunkt zurückzugeben, mit dem Sie Zeitmessungen durchführen können. Der C++11-Standard unterscheidet hierbei zwischen drei verschiedenen Zeitgebern:

► `std::chrono::system_clock`

Dieser Zeitgeber enthält die aktuelle Zeit des Systems. Das ist gewöhnlich der Zeitgeber, den auch Ihr Betriebssystem verwendet, um die Uhrzeit anzuzeigen.

► `std::chrono::steady_clock`

Dieser Zeitgeber läuft immer in einer konstanten Geschwindigkeit und kann nicht explizit verändert werden. Diese Zeit beginnt in der Regel mit dem Starten des Rechners. Wenn Ihnen dieser Zeitgeber anzeigt, dass 30 Minuten vergangen sind, dann können Sie sich auch darauf verlassen, dass es wirklich 30 Minuten waren. Die Systemzeit von `std::chrono::system_clock` hingegen kann verändert werden, und daher ist es nicht garantiert, dass 30 Minuten vergangen sind, weil jemand jederzeit die Zeit ändern kann. Der Wert von `std::chrono::steady_clock` ist garantiert immer fortlaufend (*steady*) und kann niemals weniger werden.

► `std::chrono::high_resolution_clock`

Dieser Zeitgeber enthält die höchste auf dem System verfügbare Genauigkeit. Auf manchen Systemen wird dieser Zeitgeber nur als Synonym für `std::chrono::system_clock` oder `std::chrono::steady_clock` angeboten.

Den aktuellen Zeitpunkt der drei Uhren können Sie mit `now()` zurückgeben.

16.4.2 Zeitpunkt (`time_point`) und Systemzeit (`system_clock`)

Für den Zeitpunkt der Epoche wird in `chrono` der Typ `std::chrono::time_point` definiert. Dies ist der Beginn (Epoche) der Zeitrechnung ab dem 1.1.1970. Dieser Typ wird von der Systemzeit verwendet – also in unserem Fall von `std::chrono::system_clock`. Über die Methode `time_since_epoch()` der Klasse `std::chrono::time_point` kann die Zeitdauer seit dieser Epoche zurückgegeben werden.

Boot-Zeit vs. Systemzeit

Die Systemzeit wird nur von `std::chrono::system_clock` verwendet. Die anderen Zeitgeber `std::chrono::steady_clock` und `std::chrono::high_resolution_clock` verwenden den Startzeitpunkt (Bootvorgang) des Rechners.

Da nur die Systemzeit den Beginn der Epoche verwendet, macht es auch Sinn, dass diese Klasse (`std::chrono::system_clock`) auch gleich noch mit dem C-Style-Datum und der -Uhrzeit aus der Headerdatei `<ctime>` umgehen kann. Die Klasse `std::chrono::system_clock` bietet daher mit `to_time_t()` zwei weitere Methoden an, um einen Zeitpunkt (`time_point`) in einen klassischen `std::time_t`-Typ zu konvertieren und so eine echte Uhrzeit mit bspw. `std::ctime()` ausgeben zu können. Andererseits gibt es natürlich auch noch mit `from_time_t()` eine Methode, um aus einem klassischen `std::time_t` einen Zeitpunkt von Typ `time_point` zu machen.

Hierzu ein einfaches Beispiel, welches zeigen soll, wie Sie die Systemzeit, die sogenannte Epoche, verwenden können:

```

00 // kapitel16/time01.cpp
01 #include <chrono>
02 #include <ctime>
...
03 std::chrono::time_point<std::chrono::system_clock> time,
    time2;
04 time=std::chrono::system_clock::now();
05 auto epoche = time.time_since_epoch();
06 std::time_t cur_time = std::chrono::system_clock::to_time_t(time);
07 std::time_t epo_time=
    std::chrono::system_clock::to_time_t(time2);

08 std::cout << "Epoche läuft seit ";
09 std::cout << std::ctime(&epo_time);
10 std::cout << "Seit dem sind vergangen:\n";
11 std::cout << epoche.count() << " Nanosekunden"
    << std::endl;
12 std::chrono::hours hs =
    std::chrono::duration_cast<std::chrono::hours>(epoche);
13 std::cout << hs.count() << " Stunden" << std::endl;
14 std::cout << "Heute ist " << std::ctime(&cur_time);

```

Der Code sieht zunächst schlimm aus, und hier würde man wirklich am liebsten alles in den Namensraum setzen, um nicht immer wieder `std::chrono` verwenden zu müssen. Aber dass es auch einfacher geht, können Sie in Zeile 05 mit `auto` sehen.

In Zeile 04 holen wir die aktuelle Systemzeit (`time_point`) mit der Methode `now()` ab. Die Differenz seit dieser Epoche gibt uns die Methode `time_since_epoch()` in Zeile 05 zurück. Der Rückgabewert wurde hier noch mit `auto` versteckt und lautet `std::chrono::duration`, was für die Zeitdauer steht (siehe Abschnitt 16.4.3).

In Zeile 06 konvertieren wir den aktuellen Zeitpunkt (`time_point`) in ein klassisches C-Style `std::time_t`. Das Gleiche machen wir in Zeile 07 mit einem leeren `time_point`, wodurch wir den Zeitpunkt des Anfangs der Epoche in `std::time_t` konvertieren. In Zeile 09 verwenden wir dann die C-Style-Funktion `std::ctime()`, um den konvertierten Zeitpunkt in Klartext ausgeben zu lassen (1.1.1970). In den Zeilen 11 und 12 verwenden wir zwei Methoden, welche wieder die Zeitdauer mit dem Typ `std::chrono::duration` verwenden, um einmal die Nanosekunden (Standardwert) und einmal die Stunden auszugeben, die seit Anfang der Epoche vergangen sind. Am Ende geben wir nochmals mithilfe der C-Funktion `std::time` das aktuelle Datum und die Uhrzeit auf dem Bildschirm aus.

Das Programm bei der Ausführung:

Epoche läuft seit Thu Jan 1 01:00:00 1970
 Seit dem sind vergangen:
 1381960551344019 Nanosekunden
 383877 Stunden
 Heute ist Wed Oct 16 23:55:51 2013

Jeder Zeitgeber hat seinen eigenen `time_point`

In diesem Beispiel wurde vorwiegend der Zeiger `system_clock` verwendet. Hierbei sollte aber nicht der Eindruck entstehen, dass die anderen beiden Zeiger denselben `time_point` verwenden. Alle drei Zeitgeber haben eine eigene `time_point`-Definition.

16.4.3 Zeitdauer (`duration`)

Die Zeitdauer wird von `chrono` mit dem Typ `std::chrono::duration` dargestellt. Eine Zeitdauer ist eine Differenz zwischen zwei Zeitpunkten. `std::chrono::duration` besteht aus einem arithmetischen Typ und einer Zeiteinheit (*Ticks; tick period*).

`std::ratio`

Dieses Tick wird in C++ vom Typ `std::ratio` repräsentiert. Bevor wir also mit der Zeitdauer (`duration`) fortfahren können, müssen wir ein wenig die Einheit `std::ratio` aus der Headerdatei `<ratio>` behandeln. Wenn Sie bereits einen Blick in die Dokumentation von `<ratio>` oder der Headerdatei geworfen haben, werden Sie feststellen, dass Sie dort bereits viele vordefinierte Namen vorfinden, welche Ihnen bei der Umrechnung von Einheiten behilflich sein können.

Neben den vordefinierten Einheiten ist es allerdings auch problemlos möglich, eigene Einheiten zu erstellen. Betrachten wir hierzu folgenden Code:

```
00 // kapitel16/time02.cpp
01 #include <ratio>
...
02 std::ratio<1, 100> r1;    // 1/10
03 std::ratio<20, 100> r2;   // 20/100 == 1/5
04 std::ratio<1609, 1000> r3; // 1609/1000
05 std::cout << r1.num << ":" << r1.den << std::endl; //1:100
06 std::cout << r2.num << ":" << r2.den << std::endl; //1:5
07 std::cout << r3.num << ":" << r3.den; //1609:1000
...
```

In den Zeilen **02** bis **04** definieren wir drei verschiedene Einheiten aus Zähler und Nenner. Den Zähler können Sie mit `num` (engl. für *numerator*) und den Nenner mit den

(engl. für *denominator*) ausgeben, wie Sie es in den Zeilen 05 bis 07 sehen können. Eine Einheit wie 20/100 wird automatisch gleich zur kleinsten Fraktion 1/5 berechnet.

Mithilfe von `std::ratio` können Sie jetzt auch ermitteln, wie die Zeiteinheiten für die einzelnen Zeitgeber vorgegeben sind:

```
00 // kapitel16/time03.cpp
...
01 std::cout << "system_clock: "
    << std::chrono::system_clock::period::num << ":"
    << std::chrono::system_clock::period::den << std::endl;
02 std::cout << "steady_clock: "
    << std::chrono::steady_clock::period::num << ":"
    << std::chrono::steady_clock::period::den << std::endl;
03 std::cout << "high_resolution_clock: "
    << std::chrono::high_resolution_clock::period::num << ":"
    << std::chrono::high_resolution_clock::period::den
    << std::endl;
```

Auf meinem System werden hier folgende Werte ausgegeben:

```
system_clock: 1:1000000
steady_clock: 1:1000000000
high_resolution_clock: 1:1000000000
```

Die Systemzeit (`system_clock`) läuft hier mit 1/1000000 was eine Mikrosekunde darstellt. Die beiden anderen Einheiten sind mit 1/1000000000 vorhanden, was somit einer Nanosekunde entspricht. Hier wird auch gleich bestätigt, dass `high_resolution_clock` auf diesem System gleich `steady_clock` entspricht.

Wie bereits eingangs erwähnt, benötigt eine Zeitdauer `std::chrono::duration` einen arithmetischen Typ und eine Zeiteinheit. Bei »3 Stunden« ist »3« bspw. der arithmetische Typ und »Stunden« die Einheit. Damit können Sie auch selbst eine solche Zeitdauer für `std::chrono::duration` erstellen. `std::chrono::duration<int, std::ratio<1,1>>` bspw. soll bewirken, dass wir hier die Anzahl der Sekunden vom Typ `int` speichern. Das wäre somit der Standardtyp. Hierbei können Sie noch einen weiteren Typ definieren. Mit `std::chrono::duration<double, std::ratio<3600,1>>` würden wir bspw. die Stunden darstellen, welche in einem `double` gespeichert werden.

Dies lässt sich natürlich auch für andere Dinge wie Gewicht, Maßeinheiten usw. »missbrauchen« und nicht nur für Zeiteinheiten nutzen. Folgender schlichte Codeausschnitt zeigt, wie einfach es ist, mithilfe von `std::ratio` und `std::chrono::duration` einfache Maßeinheiten umzurechnen:

```

00 // kapitel16/time04.cpp
...
01 using Kilometer =
    std::chrono::duration<double, std::ratio<1>>;
02 using Meilen =
    std::chrono::duration<double, std::ratio<1609, 1000>>;

03 Kilometer km{100};
04 Meilen m = km;
05 std::cout << m.count() << std::endl;

06 Meilen m02{10};
07 Kilometer km02 = m02;
08 std::cout << km02.count() << std::endl;

09 Kilometer km03 = km + m02;
10 std::cout << km03.count() << std::endl;

```

Zunächst legen wir für `Kilometer` in Zeile **01** mit `std::ratio<1>` den Standardwert fest. Mit `Meilen` in Zeile **02** legen wir einen zweiten Typ mit der Einheit 1609/1000 fest. Diese beiden Einheiten reichen aus, dass wir jetzt beliebige Werte von `Kilometer` nach `Meilen` und umgekehrt konvertieren können. Wir können sogar bei der Berechnung beide Einheiten mischen, wie Zeile **09** demonstriert. Die Methode `count()` übernimmt dann die Berechnung für uns.

std::chrono::duration

Wem es zu komplex erscheint, Einheiten mithilfe von `std::ratio` zu erstellen, dem sei gesagt, dass in `chrono` glücklicherweise schon solche Zeiteinheiten vordefiniert sind (siehe Tabelle 16.2) und Sie sich nicht mehr um die Einheiten kümmern müssen.

Zeiteinheit	Synonym für
nanoseconds	Nanosekunden
microseconds	Mikrosekunden
milliseconds	Millisekunden
seconds	Sekunden
minutes	Minuten
hours	Stunden

Tabelle 16.2 Vordefinierte Zeiteinheiten in »std::chrono::duration«

Auch hier wird die Verwendung dank der vordefinierten Zeiteinheiten zum Kinderspiel, wie das folgende Beispiel zeigen soll:

```
00 // kapitel16/time05.cpp
...
01 std::chrono::minutes min = std::chrono::hours(3) +
    std::chrono::minutes(33);
02 std::chrono::seconds sec = min + std::chrono::seconds(10);
03 std::chrono::milliseconds mil = std::chrono::minutes(12);

04 std::cout << min.count() << " min" << std::endl;
05 std::cout << sec.count() << " sec" << std::endl;
06 std::cout << mil.count() << " ms" << std::endl;
...
```

An diesem Beispiel können sehr schön sehen, wie Sie komfortabel verschiedene Zeiteinheiten mischen und damit arbeiten können. Dies ist alles dank eines `std::ratio` möglich. Die Ausgabe und Berechnung können Sie auch hier ganz bequem über die Elementfunktion `count()` durchführen.

duration_cast

Wenn Sie aber eine Einheit von einer höheren Auflösung zu einer niedrigeren konvertieren, dann funktioniert dies nicht mehr, weil hierbei Informationen verloren gehen. Dies ist bspw. der Fall, wenn Sie Mikrosekunden an Millisekunden zuweisen wollen, wie im folgenden Beispiel zu sehen ist:

```
std::chrono::microseconds mic(12500);
std::chrono::milliseconds mil = mic; // Fehler !!!
```

In solch einem Fall müssen Sie dann eine explizite Typumwandlung mit `duration_cast<>` vornehmen, damit der Compiler dies auch mitmacht. In unserem Beispiel sieht diese Umwandlung mit `duration_cast<>` wie folgt aus:

```
00 // kapitel16/time06.cpp
...
01 std::chrono::microseconds mic(12500);
02 std::chrono::milliseconds mil =
    std::chrono::duration_cast<std::chrono::milliseconds>(mic);
03 std::cout << mil.count() << std::endl; // 12 Millisekunden
```

Wenn Sie das Beispiel ausführen, werden Sie feststellen, dass der Wert beschnitten wird. Da die Typen als Integer integriert sind, werden aus 12500 Mikrosekunden 12 Millisekunden statt der tatsächlichen 12,5 Millisekunden. Die Stellen nach dem Komma werden einfach abgeschnitten.

Wollen Sie vermeiden, dass die Werte, wie hier, abgeschnitten werden, dann kommen Sie nicht darum herum, selbst die Einheiten mit `std::ratio` vom Typ `float` oder `double` zu implementieren. Bei `std::ratio` haben Sie bereits kurz gesehen, wie Sie das machen können. Ich will Ihnen die Lösung dazu nicht vorenthalten:

```
00 // kapitel16/time07.cpp
...
01 using myseconds =
    std::chrono::duration<double, std::ratio<1>>;
02 using mymilliseconds =
    std::chrono::duration<double, std::ratio<1,1000>>;
03 using mymicroseconds =
    std::chrono::duration<double, std::ratio<1,1000000>>;
    // ...
04 mymicroseconds mic(12500);
05 mymilliseconds mil = mic;
06 std::cout << mil.count() << std::endl; //12,5 Millisek.
...
```

Da wir jetzt benutzerdefinierte Einheiten zur Umrechnung von kleinen Zeiteinheiten für Sekunden, Millisekunden und Mikrosekunden als `double` implementiert haben, benötigen wir zum einen kein `duration_cast` mehr, und unser Ergebnis wird zum anderen auch nicht mehr beschnitten (bis zu einem gewissen Grad, da `double` ja auch nicht unendlich ist).

Alle zusammen: Zeitdauer für Zeitmessungen verwenden

Nachdem Sie die Zeitgeber (`system_clock`, `steady_clock` und `high_resolution_clock`), den Zeitpunkt (`time_point`) und die Zeitdauer (`duration`) kennengelernt haben, kennen Sie alle nötigen Komponenten der neuen Zeitbibliothek `chrono`. Zum Abschluss sollen daher nochmals alle drei Komponenten im Einsatz demonstriert werden, um eine Zeitmessung vorzunehmen. Im Beispiel wird einfach nur gemessen, wie lange eine Schleife von 0 bis 5000000 zum Zählen benötigt:

```
00 // kapitel16/time08.cpp
...
01 std::chrono::time_point<std::chrono::steady_clock> start, stop;
02 start = std::chrono::steady_clock::now();
03 for(int i=0; i<5000000; ++i); // Zeit verschwenden
04 stop = std::chrono::steady_clock::now();
05 auto dur = stop-start;
06 std::cout << dur.count() << " ns" << std::endl; //Nanosek.
07 std::chrono::milliseconds mil =
    std::chrono::duration_cast<std::chrono::milliseconds>
```



```

        (dur);
08  std::cout << mil.count() << " ms" << std::endl;
...

```

In Zeile 02 holen wir den aktuellen Zeitpunkt mithilfe von `now()`. Als Zeitgeber haben wir in Zeile 01 `steady_clock` (Bootzeit) verwendet, weil diese auf meinem System etwas genauer ist als `system_clock` und weil `high_resolution_clock` der `steady_clock` entspricht. In Zeile 03 lassen wir die CPU ein wenig arbeiten – was in diesem trivialen Beispiel auch unsere Zeitmessung sein soll. Nach der Arbeit holen wir erneut den Zeitpunkt in Zeile 04 ab. Jetzt benötigen Sie nur noch die Zeitdauer (*duration*) dieser beiden Zeitpunkte, subtrahieren `stop-start` und erhalten so die Zeitdauer, welche hier in `dur` gespeichert wird. Mit `dur.count()` können Sie ausgeben (Zeile 06), wie lange die Schleifenausführung benötigt hat. In Listing *time03.cpp* haben Sie gesehen, dass auf meinem System Nanosekunden für Ticks von `steady_clock` verwendet werden. Somit ist die Ausgabe hier in Nanosekunden. In Zeile 07 wollen wir noch das Ergebnis in Millisekunden umwandeln und müssen hierbei in Kauf nehmen, dass der Wert beschnitten wird. In Zeile 08 wird dann die Zeitdauer in Millisekunden ausgegeben.

Das Programm bei der Ausführung:

```

41401496 ns
41 ms

```

16.5 Type-Traits

Ebenfalls neu in C++11 sind die *Type Traits* hinzugekommen. Mit den Type Traits haben Sie die Möglichkeit, bereits zur Übersetzungszeit Typabfragen, Typvergleiche und Typtransformationen durchzuführen. Da dies alles zur Übersetzungszeit geschieht, wird keinerlei Laufzeit für das Programm verschwendet. Die Type Traits sind schon ein etwas fortgeschrittenes Thema und finden ihren Einsatz in der Template-Metaprogrammierung von C++.

Typeeigenschaften abfragen

Die Typeeigenschaften lassen sich mit sogenannten Prädikaten abfragen. Hierbei wird zwischen der primären Typkategorie und der zusammengesetzten Typkategorie unterschieden.

Bei einer **primären Typkategorie** kann ein Datentyp nur einer Kategorie angehören. So kann bspw. ein Zahlentyp nur entweder ein Integer (`is_integral<T>`) oder eine Gleitpunktzahl (`is_floating_point<T>`) sein. Insgesamt finden Sie hier dreizehn verschiedene primäre Typkategorien vor: `is_void<T>`, `is_integral<T>`, `is_floating_point<T>`, `is_array<T>`, `is_pointer<T>`, `is_lvalue_reference<T>`, `is_rvalue_refe-`

rence<T>, is_member_object_pointer<T>, is_member_function_pointer<T>, is_enum<T>, is_union<T>, is_class<T> und is_function<T>.

Die **zusammengesetzten Typkategorien** hingegen basieren auf den primären Typkategorien. So kann man sagen, dass Zahlentypen von Integer und Gleitkommazahlen arithmetische Typen sind. Hierfür kann dann bspw. das Prädikat `is_arithmetic<T>` zum Überprüfen verwendet werden. Die zusammengesetzte Typkategorie `is_arithmetic<T>` gibt `true` zurück, wenn `is_floating_point<T>::value` oder `is_integral<T>::value` gleich `true` zurückgeben. Neben `is_arithmetic<T>` gibt es hier noch `is_fundamental<T>`, `is_object<T>`, `is_scalar<T>`, `is_reference<T>`, `is_compound<T>` und `is_member_pointer<T>`.

Da sich das Thema recht komplex anhört, wurde versucht, hier ein möglichst primitives Beispiel zu erstellen, um den Sachverhalt von Type Traits und hier jetzt dem Abfragen einer Typeigenschaft möglichst einfach verständlich zu machen. Verwenden wir hierzu ein altes, klassisches Beispiel, welches den größeren von zwei Werten aus einer Template-Funktion zurückgibt:

```
01 template <typename T>
02 T BigNum(T val1, T val2) {
03     if( val1 >= val2 ) return val1;
04     return val2;
05 }
```

Die Funktion sollte klar sein, und die Verwendung hierbei ist wie folgt angedacht:

```
01 int val = BigNum(100, 200);
02 std::cout << val << std::endl;
03 double dval = BigNum(3.14, 3.1415);
04 std::cout << dval << std::endl;
```

Trotzdem ist es in diesem Beispiel auch möglich, die Template-Funktion wie folgt zu missbrauchen:

```
01 std::string str = BigNum("Hallo", "Type");
02 std::cout << str << std::endl;
03 const char* cstr = BigNum("Zebra", "Auto");
04 std::cout << cstr << std::endl;
```

Bei `std::string` als Argument kommt noch etwas Vernünftiges heraus, weil hier ein lexikografischer Vergleich durchgeführt wird. Aber bei `const char*` wird nur noch die größere von zwei Adressen (!) zurückgegeben.

Wir wollen jetzt aber sicherstellen, dass bei der Template-Funktion `BigNum()` nur arithmetische Typen verwendet werden. Hier könnten Sie jetzt hergehen und folgende Spezialisierungen schreiben:

```

...
template<>
std::string BigNum<std::string>(
    std::string val1, std::string val2) {
    std::cerr << "Ungültige Verwendung von BigNum\n";
    return val1;
}

template<>
const char* BigNum<const char*>
    (const char* val1, const char* val2) {
    std::cerr << "Ungültige Verwendung von BigNum\n";
    return val1;
}

```

Jetzt haben Sie zumindest diese Typen »ausgeschlossen«. Trotzdem können Sie jetzt noch verschiedene andere Typen missbrauchen, um dieses Funktions-Template zu verwenden, und es kann ein ziemlicher Aufwand sein, für alle eine Spezialisierung zu schreiben. Des Weiteren wird durch die Spezialisierung unnötig viel Ballast produziert, den das Programm mit sich herumschleppen muss, was sich dann auch letztendlich in der Laufzeit widerspiegelt. Es ist außerdem irgendwie sinnlos, Spezialisierungen zu schreiben für Dinge, die man gar nicht im Programm haben will. Und genau an dieser Stelle kommen jetzt die Type Traits ins Spiel.

In unserem Beispiel wollen wir eben nur arithmetische Typen erlauben, ohne zur Laufzeit irgendwelche aufwendigen Typüberprüfungen durchzuführen. Mehr als eine Zeile Code ist mit den Type Traits hierfür nicht nötig:

```

00 // kapitel16/type_traits01.cpp
01 #include <type_traits> // benötigte Headerdatei
...
02 template <typename T>
03 T BigNum(T val1, T val2) {
04     static_assert(std::is_arithmetic<T>::value,
05                 "T sollte ein arithmetischer Typ sein");
06     if( val1 >= val2 ) return val1;
07     return val2;
08 }

```

Wenn Sie jetzt versuchen, eine Instanz mit `std::string` oder `const char*` als Argumente und Rückgabewerte erzeugen, wird der Compiler die Übersetzung aufgrund von Zeile 03 abbrechen und die Fehlermeldung von `static_assert()` ausgeben. Das Programm wird nur kompilieren, wenn `std::is_arithmetic<T>::value` gleich `true` ist. Wollen Sie hier nur Integer erlauben, können Sie stattdessen auch `std::is_inte-`

`gral<T>::value` hinschreiben. Für »nur« Gleitkommazahlen müssten Sie `std::is_floating_point<T>::value` verwenden. Ich denke, das dürfte den Sachverhalt recht deutlich verständlich gemacht haben, wie sich Type Traits einsetzen lassen.

Jetzt haben Sie eine Menge Typeigenschaften, die Sie abfragen können. Was ist aber, wenn Sie es gerne noch spezieller hätte, ein eigenes `::value`, wenn Sie so wollen? So etwas lässt sich, ebenfalls wieder mit Templates, einfacher erstellen, als Sie denken. Hier das Beispiel dazu:

```
00 // kapitel16/type_traits02.cpp
...
01 template <typename T>
02 struct is_num {
03     static const bool value=false;
04 };

05 template <>
06 struct is_num<unsigned short> {
07     static const unsigned short value=false;
08 };

09 template <>
10 struct is_num<short> {
11     static const short value=false;
12 };

13 template <>
14 struct is_num<unsigned int> {
15     static const unsigned int value=true;
16 };

17 template <>
18 struct is_num<int> {
19     static const int value=true;
20 };

21 template <>
22 struct is_num<float> {
23     constexpr static const float value=false;
24 };

25 template <>
26 struct is_num<double> {
27     constexpr static const double value=true;
28 };
...
```

In den Zeilen **01** bis **04** erzeugen wir ein Template für eine Struktur mit einem einzigen öffentlichen Mitglied mit dem Bezeichner `value`. In Zeile **03** weisen wir `value` vom Typ `bool` den Wert `false` zu, und Sie ahnen schon, was wir hier bewirken wollen. Dasselbe machen wir jetzt noch als Spezialisierung für andere Basisdatentypen. Dieses Template und dessen Spezialisierungen können Sie jetzt ganz einfach wie folgt verwenden:

```
00 // kapitel16/type_traits02.cpp
01 #include <type_traits> // benötigte Headerdatei
...
02 template <typename T>
03 T BigNum(T val1, T val2) {
04     static_assert(std::is_arithmetic<T>::value,
                    "T sollte ein arithmetischer Typ sein");
05     static_assert(is_num<T>::value,
                    "T darf nur unsigned int, int oder double sein");
06     if( val1 >= val2 ) return val1;
07     return val2;
08 }
...
09 short val1=100, val2=200;
10 short val = BigNum(val1, val2); // Fehler !!!
11 std::cout << val << std::endl;
12 float fval1=3.123, fval2=3.234;
13 float dval = BigNum(fval1, fval2); // Fehler !!!
14 std::cout << dval << std::endl;
...
```

Zeile **04** können wir drinlassen, da der Typ ein arithmetischer ist. Neu ist jetzt Zeile **05** mit unserem eigenen `is_num<T>::value`. Dieser Wert muss `true` zurückgeben, damit unser Quellcode kompiliert werden kann. In diesem Beispiel erlauben wir jetzt nur noch `unsigned int`, `int` und `double` als Typ um eine Instanziierung aus `BigNum` zu machen. Der Compiler wird daher die Zeilen **10** mit `short` und **13** mit `float` nicht übersetzen, weil wir zuvor `is_num<short>::value` und `is_num<float>::value` auf `false` gesetzt haben.

Das Ganze können Sie natürlich dank `static_assert()` noch etwas erweitern. Wollen Sie bspw. nur integrale Typen zulassen, die 4 Bytes haben, dann brauchen Sie keine weiteren Templates oder Spezialisierungen für `is_num<T>::value` zu schreiben, sondern Sie schreiben einfach:

```
static_assert(
    std::is_arithmetic<T>::value && sizeof(T) == 4,
    "T sollte ein arithmetischer Typ sein");
```

Typeeigenschaften überprüfen

Neben den Typkategorien und zusammengesetzten Typkategorien bietet Type Traits auch die Möglichkeit an, die Typeigenschaften abzufragen, also Dinge wie, ob ein Typ mit `const` qualifiziert oder `unsigned` ist. Auch hierfür gibt es wieder eine Menge vorgefertigter Prädikate: `is_const<T>`, `is_volatile<T>`, `is_polymorphic<T>`, `is_abstract<T>`, `is_signed<T>`, `is_unsigned<T>`, `is_trivial<T>`, `is_trivial_coyable<T>`, `is_standard_layout<T>`, `is_pod<T>`, `is_literal_type<T>` und `is_empty<T>`.

Die Verwendung entspricht exakt der der Typkategorien mit `::value`. Ein Beispiel mit `is_unsigned<T>`:

```
00 // kapitel16/type_traits03.cpp
...
01 template <typename T>
02 T BigNum(T val1, T val2) {
03     static_assert(std::is_integral<T>::value &&
04                   std::is_unsigned<T>::value,
05                   "T sollte ein integraler unsigned Typ sein");
04     if( val1 >= val2 ) return val1;
05     return val2;
06 }
```

Jetzt wird nur eine Instanziierung des Funktions-Templates gemacht, wenn `T` ein integraler Typ und `unsigned` ist.

Überprüfung benutzerdefinierter Typen

Die Type-Traits-Bibliothek bietet auch Operationen für benutzerdefinierte Typen wie Klassen an. So kann bspw. mit `is_constructible` überprüft werden, ob der Typ einen Konstruktor für Argumente hat. Mit `is_default_constructible` kann ermittelt werden, ob ein Default-Konstruktor vorhanden ist. Hiervon gibt es noch weitere Versionen für Überprüfungen von Konstruktoren und Zuweisungsoperatoren.

Typvergleiche

Neben der Möglichkeit der Typabfragen, gibt es auch einige Prädikate, um zur Übersetzungszeit Typen zu vergleichen. Die Type-Traits-Bibliothek bietet hierzu vier Vergleiche an:

- ▶ `is_same<T, U>`: Prüft, ob die Typen `U` und `T` gleich sind.
- ▶ `is_base_of<Base, Derived>`: Prüft, ob ein Typ von einem anderen abgeleitet wurde.
- ▶ `is_convertible<From, To>`: Prüft, ob ein Typ in den anderen konvertiert werden kann.

Das folgende Listing zeigt `is_same<T>` im Einsatz:

```
00 // kapitel16/type_traits04.cpp
...
01 template <typename T, typename U>
02 T BigNum(T val1, U val2) {
03     static_assert(std::is_same<T, U>::value,
                    "T und U müssen gleich sein");
04     if( val1 >= val2 ) return val1;
05     return val2;
06 }
...
```

Jetzt wird nur eine Instanz vom Funktions-Template erzeugt, wenn die Typen `T` und `U` vom selben Typ sind. In diesem Beispiel eigentlich ziemlich Unsinn, aber es demonstriert zumindest einen Typvergleich zu Übersetzungszeit.

Typtransformation

Typen vor der Übersetzungszeit zu überprüfen, ist schon nicht schlecht. Aber es ist auch noch möglich, Typen zur Übersetzungszeit zu transformieren. Sie können bspw. einfach `const` mit `remove_const<T>` entfernen, oder auch andersherum können Sie mit `add_const<T>` für den Typ `T` ein `const` hinzufügen.

Fazit

Zugegeben, der Abschnitt hier war nur ein kleiner Überblick über den Type Traits, aber ich denke, Sie konnten schon die Mächtigkeit dieser neuen Technik erkennen. Gerade wenn Sie irgendwann einmal eigene Bibliotheken schreiben, werden Sie dieses Werkzeug zur Template-Metaprogrammierung zu schätzen wissen.

Dokumentation

Auch hier macht es wenig Sinn, alle Typvergleiche, Typabfragen und Typtransformationen haarklein aufzuzählen. Hierfür würden enorm viele Seiten benötigt. Es ist auch gar nicht nötig, da Sie jetzt wissen, was Type-Traits sind und wie Sie diese verwenden können, reicht ein Blick in eine Referenz, wie bspw. <http://de.cppreference.com/w/cpp/types>, um das passende Prädikat für sich zu finden.

16.6 POD (Plain Old Data) (C++11)

Als POD (*Plain Old Data*) werden Datenstrukturen bezeichnet, die ein C-Layout besitzen. C++11 hat diese Typen um Strukturen und Klassen erweitert. Damit Strukturen

oder Klassen auch als POD gelten, müssen einige Bedingungen erfüllt werden. Es kann nicht jede Struktur oder Klasse als POD verwendet werden. Als POD gilt eine Klasse oder Struktur, wenn diese trivial ist, ein Standardlayout besitzt und die nicht-statischen Mitglieder wiederum PODs sind.

Trivial ist eine Klasse bzw. Struktur, wenn diese einen gewöhnlichen Konstruktor, Kopierkonstruktor, Zuweisungsoperator und/oder einfachen (keinen virtuellen) Destruktor enthält.

Ein Standardlayout ist gegeben, wenn keine virtuellen Methoden, virtuellen Basis-klassen, keine Referenzen zum Einsatz kommen und/oder alle nichtstatischen Mitglieder denselben Zugriffsspezifizierer haben.

Wozu ist POD gut?

Wozu überhaupt C-Style-Dinge wie POD für Klassen und Strukturen, werden Sie sich fragen? Ganz einfach, um an der Performance-Schraube zu drehen, indem Sie auf wesentlich effizientere Funktionen wie `memcpy`, `memmove` zum Kopieren oder `memset` zum Initialisieren zurückgreifen können. Letztendlich machen auch viele Algorithmen der Standardbibliothek nichts anderes, als versteckt auf diese alten C-Funktionen zurückzugreifen.

Wenn Sie sich jetzt nicht sicher sind, ob ein Datentyp ein POD ist oder nicht, können Sie hierzu ganz einfach die Template-Metaprogrammierung mit der neuen Type-Traits-Bibliothek und das darin enthaltene Prädikat `std::is_pod<T>` verwenden. Folgendes Beispiel demonstriert, wie Sie dies überprüfen können:

```
00 // kapitel16/is_pod.cpp
01 #include <type_traits> // für is_pod<T>
...
02 struct Data01 {
03     int ival;
04 }; // == POD

05 struct Data02 {
06     int ival;
07     Data02() : ival(0) {}
08 }; // kein POD
...
09 Data01 d01;
10 d01.ival = 1234;
11 Data02 d02;
12 d02.ival = 4321;
...
```



```

13 static_assert( std::is_pod<Data01>::value,
                  "Data01 is kein POD");
14 static_assert( std::is_pod<Data02>::value,
                  "Data02 is kein POD");

```

Die Struktur `Data01` ist noch ein POD. `Data02` ist kein POD mehr, da kein trivialer Konstruktor mehr vorhanden ist. Daher wird `static_assert()` in Zeile 14 die Übersetzung verweigern, weil `std::is_pod<Data02>::value` hier nicht mehr `true` zurückgibt.

Natürlich will ich Ihnen nicht vorenthalten, wie Sie eine eigene, extrem schnelle Kopierfunktion mit `std::memcpy()` und den PODs schreiben können. Hier das einfache Beispiel:

```

00 // kapitel16/pod_copy.cpp
01 #include <type_traits>

02 struct Data {
03     int ival;
04 }; // == POD

05 static const size_t chunk = 1000000;

06 void myCopy(Data* dest, Data* src, const size_t ch) {
07     std::memcpy(dest, src, ch * sizeof(Data));
08 }
...
09 static_assert( std::is_pod<Data>::value,
                  "Data is kein POD" );

10 Data BigData01[chunk], BigData02[chunk];
11 for(int i=0; i<chunk; ++i) BigData01[i].ival = i;
12 myCopy(BigData02, BigData01, chunk);

```

In Zeile 10 legen wir zwei sehr große Arrays an. In Zeile 11 wird eines der Arrays mit Werten initialisiert, und in Zeile 12 kopieren wir dann dieses große Array mit der Funktion `myCopy()`, wobei wir das extrem schnelle `std::memcpy()` zum Kopieren verwenden. Die Überprüfung, dass es sich hierbei auch wirklich um ein POD handelt, haben wir in Zeile 09 ebenfalls mit eingebaut.

16.7 std::initializer_list

Die neue Möglichkeit, die verschiedensten Datenstrukturen in C++ mit einer Initialisierungsliste zu initialisieren, ist wirklich sehr hilfreich:

```

01 int carr[] = {1, 2, 3, 5, 8, 13};
02 std::string cppstr[] = {"String1", "String2", "usw."};
03 std::vector<int> vec = {1, 2, 3, 5, 8, 13};

04 for(auto n : carr) std::cout << n << ' ';
05 for(auto n : cppstr) std::cout << n << ' ';
06 for(auto n : vec) std::cout << n << ' ';

```

In den Zeilen **01** bis **03** können Sie sehen, dass die Syntax immer dieselbe ist, egal, welche Datenstruktur Sie hier verwenden. Ebenso einheitlich ist dann auch der Zugriff, wie die Zeilen **04** bis **06** zeigen sollen. Aber das Thema wurde bereits behandelt.

Was uns hier vielmehr interessiert, ist, wie wir so etwas in benutzerdefinierte Typen einbauen können, um dort ebenfalls eine einheitliche Initialisierungsliste verwenden zu können. Mit C++11 können Sie dies jetzt realisieren, indem Sie einen Konstruktor mit `std::initializer_list<T>` definieren. Für das Template benötigen Sie zusätzlich den Header `<initializer_list>`. In der Praxis sieht das dann wie folgt aus:

```

00 // kapitel16/initializer_list01.cpp
01 #include <initializer_list>
...
02 template <typename T>
03 struct Data {
04     std::vector<T> dat;
05     Data(std::initializer_list<T> list) : dat(list) {
06         std::cout << list.size() << " Elemente erzeugt\n";
07     }
08 };
...
09 Data<int> mydat = { 1, 2, 3, 5, 8, 13 };
10 Data<double> mydat2 {1.5, 2.5, 3.5, 4.5 };
...
11 for(auto n : mydat.dat) std::cout << n << ' ';
12 for(auto n : mydat2.dat) std::cout << n << ' ';

```

Eine solche `std::initializer_list<T>` kann nur Elemente vom selben Typ enthalten. Ein `std::initializer_list<int>` kann also nur `int`-Typen in die Liste aufnehmen. Benötigen Sie hingegen weitere Parameter, müssen Sie dies beim Aufruf explizit angeben und in einem weiteren Klammerpaar mit angeben:

```

00 // kapitel16/initializer_list02.cpp
01 #include <initializer_list>
...

```

```

02 template <typename T>
03 struct Data {
04     std::vector<T> dat;
05     std::string key;
06     Data(std::initializer_list<T> list) : dat(list) {
07         std::cout << list.size() << " Elemente erzeugt\n";
08     }
09     Data(std::initializer_list<T> list, std::string s) :
10         dat(list), key(s) {
11         std::cout << list.size() << " Elemente erzeugt";
12         std::cout << " mit dem Schlüssel: " << s
13             << std::endl;
14     }
15 };
16 ...
17 Data<int> mydat3{ {8, 2, 3, 3}, "Mering" };
18 for(auto n : mydat3.dat) std::cout << n << ' ';
19 std::cout << " Schlüssel: " << mydat3.key << std::endl;

```

In Zeile 09 sehen Sie den Konstruktor, der neben einer `std::initializer_list` noch einen `std::string` enthält. Daher müssen Sie bei der direkten Initialisierung in Zeile 14 auch ein weiteres Klammerpaar dafür verwenden.

Sie können solch eine Liste übrigens auch »herumtragen« wie einen normalen Typ und an Funktionen oder Methoden übergeben. Hierzu noch ein letztes Beispiel:

```

00 // kapitel16/initializer_list03.cpp
01 #include <initializer_list>
02 ...
03 template <typename T>
04 struct Data {
05     std::vector<T> dat;
06     std::string key;
07     Data(std::initializer_list<T> list) : dat(list) {
08         std::cout << list.size() << " Elemente erzeugt\n";
09     }
10     Data(std::initializer_list<T> list, std::string s) :
11         dat(list), key(s) {
12         std::cout << list.size() << " Elemente erzeugt";
13         std::cout << " mit dem Schlüssel: " << s
14             << std::endl;
15     }
16 }
17 void add_Datas(std::initializer_list<T> list) {

```

```

14     dat.insert(dat.end(), list.begin(), list.end());
15 }
16 size_t Data_size() const {return dat.size();}
17 };
...
18 Data<int> mydat3{ {8, 2, 3, 3}, "Mering" };
19 mydat3.add_Datas({6, 3, 7, 1});
20 std::cout << mydat3.Data_size() << " Elemente in ";
21 std::cout << mydat3.key << std::endl;

22 auto listVals = {4, 5, 2, 5, 9};
23 mydat3.add_Datas(listVals);
24 std::cout << mydat3.Data_size() << " Elemente in ";
25 std::cout << mydat3.key << std::endl;

26 for(auto n : mydat3.dat) std::cout << n << ' ';

```

In Zeile 19 übergeben wir eine ganze Liste von `std::initializer_list<int>` an die Methode `add_Datas()`, die wir in den Zeilen 13 bis 15 definiert haben und wo wir in Zeile 14 die Daten von `std::initializer_list` von `begin()` bis `end()` an das Ende von `dat.end()` hängen. Dies wird ermöglicht, weil die `std::initializer_list` ebenfalls die Iteratoren `begin()` und `end()` enthalten.

Zeile 22 zeigt auch, dass wir den Typ von `std::initializer_list` auch einfach wie einen gewöhnlichen Typ verwenden können. In Zeile 23 übergeben wir diese Liste wieder an `add_Datas()`. Trotzdem könnten Sie mit `listVals` auch unabhängig davon weiterarbeiten.

Kapitel 17

GUI-Programmierung mit Qt

Die häufigste Frage, die jeder Einsteiger früher oder später stellt, ist: »Wie kann ich Anwendungen mit einer grafischen Oberfläche in C++ schreiben?« Die Grundvoraussetzung, um sogenannte GUI-Toolkits (GUI = Graphical User Interface) zu benutzen, sind fundierte C++-Kenntnisse. Wenn Sie das Buch bis zu diesem Kapitel durchgearbeitet haben, dürfte dies kein Problem mehr darstellen. Als Nächstes stellt sich dann die Frage, welches GUI-Toolkit man dafür verwenden sollte. Hierfür gibt es eine Menge interessanter GUI-Toolkits. In diesem Buch werden wir das Qt-Framework hierfür verwenden.

Sicherlich stellen Sie sich zunächst die Frage, warum es ausgerechnet Qt sein soll bzw. warum ich mich in diesem Buch für Qt entschieden habe? Warum nicht MFC von Microsoft? Einen Vergleich mit anderen GUI-Frameworks anzustellen, macht meistens wenig Sinn. Zunächst habe ich mich für Qt entschieden, weil sich dieses Framework mittlerweile zu einem der Platzhirsche gemausert hat und ziemlich beliebt ist. Topsoftware wie u.a. Adobe Photoshop Elements, Google Earth, Skype, Spotify, Mathematica wurden mit Qt erstellt, um einige bekannte Anwendungen zu nennen.

Die Liste der Firmen, die Qt verwenden, ist enorm und auch recht beeindruckend. Natürlich bedeutet das noch lange nicht, dass es das Nonplusultra ist, nur weil Firmen wie Adobe, Google, Samsung, Walt Disney Feature Animation, NASA usw. ein Framework verwenden, aber es hat schon was. Viele Anwendungen, die mit Qt erstellt werden, bekommt man sowieso nie zu Gesicht, weil es sich hierbei meistens um speziell für Firmen erstellte Programme handelt.

Aber (oder besser vor allem) auch von der technischen Seite hat Qt eine Menge zu bieten. Das Framework ist sehr flexibel und kann auf vielen gängigen Systemen eingesetzt werden. Hiermit ist es egal, wo Sie Ihren Quellcode schreiben, die Anwendung kann dann sowohl unter MS-Windows, Mac OS X, Linux oder anderen Unix-artigen Systemen mit X11 übersetzt und auch dort zur Verfügung gestellt werden.

iOS und Android

Künftige Versionen sollen außerdem auch iOS unterstützen, um Anwendungen (Apps) für das iPhone oder iPad zu entwickeln. Auch für die Android-Plattform lassen sich dann Apps entwickeln. Zur Drucklegung dieses Buches wurde bereits das Qt-Framework zur Android-App-Entwicklung zur Verfügung gestellt. Als Beispiel finden Sie mit »Qt Everywhere« im Google Store eine Demonstrations-App. Für iOS steht die App »Sub Attack« auf iTunes zur Verfügung. Beide Apps wurden mit dem Qt-Framework entwickelt.

Neben dem portablen Quellcode ist auch der Reichtum an Funktionalität ein gewaltiger Grund, das Qt-Framework zu verwenden. Einige mir bekannte Entwickler verwenden Qt allein deshalb, um die Netzwerkfunktionalität davon in Ihrem Programm zu verwenden. Auch hier wiegt der Vorteil der Portabilität sehr stark.

Bei der gewaltigen Vielfalt, die Qt bietet, wurde trotzdem beachtet, dass sich das Framework auch relativ einfach anwenden lässt. Auch die Dokumentation ist vom Feinsten und lässt keine Fragen mehr offen.

Was können Sie von diesem Kapitel erwarten

Ich will Ihnen in diesem Kapitel das Qt-Framework etwas näherbringen und auch auf die grundlegenden Konzepte eingehen. Wenn Sie von diesem Framework überzeugt sind, kommen Sie allerdings nicht darum herum, sich eingehender mit dem Framework zu befassen, was allerdings angesichts der exzellenten Dokumentation und den vielen Beispielen, die dem Framework beiliegen, keine allzu großes Problem darstellen sollte. Trotzdem ist das Kapitel hier nicht mehr als nur ein kleiner Leitfaden zum Qt-Framework, der Sie befähigen soll, kleinere GUI-Anwendungen mit Qt zu schreiben. Viele Listings wurden auf die Buch-CD verlagert, weil es unzumutbar ist, hier mehrere Seiten mit Code abzudrucken.

Hier noch ein paar Checkpunkte, was Sie in diesem Kapitel erfahren werden:

- ▶ Umgang mit dem Qt Creator, Qt Designer und der Hilfe (Qt Assistant)
- ▶ Signal-Slot-Verbindung (Kommunikation zwischen Widgets)
- ▶ eigene Widgets erstellen (manuell und mit dem Qt Designer) und verwenden
- ▶ Erstellen von Dialogen, daneben Übersicht der vorgefertigten Dialoge von Qt
- ▶ Übersicht der grundlegenden Steuerelemente von Qt für eine grafische Anwendung
- ▶ Anwendung mit einem Hauptfenster erstellen

Bibliotheksaufbau (Übersicht)

Die Bibliothek ist extrem umfangreich und in mehrere Module aufgeteilt. Mehrere Module bringen den Vorteil, dass die Programme schlanker und schneller sind, weil nur die verwendeten Module zum Programm hinzugelinkt werden. Des Weiteren hilft diese Modularisierung enorm, um sich einen Überblick zu verschaffen und gegebenenfalls das herauszupicken, was man für sein Programm benötigt. In Tabelle 17.1 finden Sie einen Überblick zu den verschiedenen Modulen des Qt-Frameworks.

Modul	Beschreibung
Qt Core	Kernklassen ohne grafische Oberfläche, welche von allen anderen Modulen verwendet werden
Qt GUI	Enthält die Komponenten, welche zur Erstellung von grafischen Oberflächen benötigt werden (inklusive OpenGL).
Qt Multimedia	Klassen für Audio-, Video, Radio- und Kamera-Funktionalitäten auf einer niedrigeren Ebene
Qt Multimedia Widgets	Widget-basierte Klassen für die Multimedia-Funktionalitäten
Qt Network	Klassen für die Netzwerkprogrammierung
Qt SQL	Klassen für die Datenbankintegration mit SQL
Qt Test	Werkzeuge zum Testen der eigenen Anwendungen
Qt WebKit	Klassen für WebKit2-basierte Anwendungen (inklusive der neuen QML API)
Qt WebKit Widgets	WebKit1- und QWidget-basierte Klassen von Qt4
Qt Widgets	Klassen, welche Qt GUI um weitere C++-Widgets erweitern

Tabelle 17.1 Übersicht zu einigen grundlegenden Qt-Modulen

Qt Quick

Nicht aufgelistet wurden hier die neueren Qt Quick-Module (mit Qt Quick, Qt Quick Controls und Qt Quick Layouts), welche quasi Frameworks innerhalb des Qt-Frameworks sind und womit sich benutzerdefinierte UIs mit fließenden Übergängen und Effekten erzeugen lassen, was gerade bei mobilen Geräten sehr gängig ist.

Qt Quick enthält auch die Skriptsprache QML. Zwar unterstützt Qt auch mobile Systeme wie Symbian und Maemo/MeeGo, aber richtig interessant dürften diese Module werden, wenn Qt mit iOS und Android verwendet werden kann (was wohl beim Lesen dieser Zeilen schon der Fall sein dürfte).

Auf einigen Systemen finden Sie noch viele weitere Module in Form eines Add-ons vor. Dazu gehören u.a. Dinge wie OpenGL, Format für Bilder, Druckerunterstützung, SVG, XML und noch einige mehr.

Was brauche ich?

Mittlerweile wird für Windows, Mac OS X und Linux ein komplettes SDK (Software Development Kit) zum Download angeboten. Dieses Kit beinhaltet alle nötigen Qt-Bibliotheken, gegebenenfalls den passenden Compiler, eine umfangreiche Qt-Dokumentation und den Qt Creator. Der Qt Creator ist eine komplette und mittlerweile mächtige Entwicklungsumgebung, mit der Sie auch gewöhnliche Standard-C++-Projekte programmieren können. Bei Mac OS X wird der *clang* als Compiler verwendet (und auf Wunsch mit installiert). Das komplette und neueste SDK (zur Drucklegung war 5.1.1 aktuell und die Beta mit 5.2) können Sie sich von <http://qt-project.org/downloads> herunterladen und dann den Installationsanweisungen auf dem Bildschirm folgen.

Was kostet Qt?

Zunächst müssen Sie sich über die Kosten keine Gedanken machen, sofern Sie Qt einfach nur kennenlernen wollen. Auch wenn Sie Qt-Anwendungen für private Zwecke oder den Open-Source-Bereich entwickeln, kostet Qt Sie nichts. Erst wenn Sie kommerzielle Anwendungen (bspw. für Ihr eigenes Unternehmen) erstellen, wird eine Lizenzierung nötig. Qt verwendet ein duales Lizenzsystem und steht somit unter einer kommerziellen und einer Open-Source-Lizenz. Mehr zu diesem dualen Lizenzsystem entnehmen Sie bitte folgender Seite: <http://de.wikipedia.org/wiki/Mehrfachlizenzierung>. Wenn Sie eine kommerzielle Lizenz erwerben wollen, sollten Sie Ihre Anfrage an Digia selbst stellen. Die Kontaktadresse lautet: <http://qt.digia.com/About-us/Contact-us>.

17.1 Ein erstes Programmbeispiel schreiben

Um ein erstes Programm zu schreiben, werden wir hier auch gleich die im SDK mitgelieferte Entwicklungsumgebung *Qt Creator* verwenden. Zwar gibt es durchaus auch die Möglichkeit, eine andere Entwicklungsumgebung zu verwenden oder gar die Listings in der Kommandozeile zu übersetzen, aber der Vorteil von Qt Creator ist der, dass es auf allen Systemen mit dem gleichen, beschriebenen Weg funktioniert, was Ihnen den Nachvollzug erleichtert. Ich gehe daher davon aus, dass Sie das komplette SDK (mitsamt Qt Creator) heruntergeladen und installiert haben. Jetzt müssen Sie nur noch den Qt Creator starten, indem Sie das entsprechende Icon anklicken. Wo Sie das Icon finden, variiert von System zu System und hängt auch davon ab, wo Sie das SDK installiert haben.



Abbildung 17.1 Der Qt Creator nach dem Start (hier unter Mac OS X)

17.1.1 Kurze Übersicht zur Oberfläche von Qt Creator

Hier soll kurz die Oberfläche von Qt Creator etwas beschrieben werden. Eine umfassende Beschreibung macht in einem solchem Buch wenig Sinn und ist dank der tollen mitgelieferten Dokumentation auch gar nicht nötig.

Nach dem Start von QT CREATOR finden Sie auf der linken Seite sieben verschiedene Arbeitsmodi ❶ (Abbildung 17.1):

- **WILLKOMMEN:** Auf dem Willkommens-Bildschirm finden Sie einen tollen Einstieg zur Benutzeroberfläche und viele Anleitungen und Dokumentationen zu Qt. Wenn Sie in Qt einsteigen wollen, finden Sie hier eigentlich alles, was Sie über Qt und die Entwicklung von Software damit wissen wollen. Hier können Sie auch gleichzeitig die unzähligen Beispiele von Qt öffnen und anschließend übersetzen und ausführen. Der Clou an den Projekten ist außerdem, dass zusätzlich noch ein Fenster geöffnet wird, in dem die wichtigen Codestellen des Projekts genauer beschrieben werden.
- **EDITIEREN:** In diesem Modus können Sie das Projekt und die Codedateien editieren. In einer weiteren Seitenleiste werden die einzelnen Dateien aufgelistet. Der Editor bietet im Grunde alles, was das Entwicklerherz begehrt, um einen Code zu entwerfen, zu editieren und darin zu navigieren. Neben der Codehervorhebung

bietet der Editor auch Autovervollständigung, Kontext-Hilfe und eine Fehlerüberprüfung während der Eingabe an.

- **DESIGN:** Hier stehen Ihnen gleich zwei integrierte grafische Editoren zur Erstellung von Benutzerschnittstellen zu Verfügung: der *Qt Designer*, welcher die Benutzerschnittstellen mit Qt Widgets erstellt, und dann wäre da noch der neuere *Qt Quick Designer*, welcher animierte Benutzeroberflächen mithilfe der Sprache QML erstellen kann. Bei der Projektübersicht können Sie durch Anklicken der entsprechenden Datei den dazugehörigen Designer öffnen. Die Dateierendung *.ui* wird von Qt Designer und *.qml* eben von Qt Quick Designer verwendet. In diesem Buch wird nur auf den Qt Designer eingegangen.
- **DEBUGGEN:** Auf der Suche nach Fehlern finden Sie auch hier, wie es sich für eine Entwicklungsumgebung gehört, einen Debug-Modus.
- **PROJEKTE:** Hier können Sie alle Einstellungen zu einem Projekt vornehmen, und zwar vom Bauen einer Debug- oder Release-Version bis hin zur Ausführung. Wenn mal etwas nicht klappt oder fehlt, können/sollten Sie hier nachsehen.
- **ANALYSE:** Hier ist eine Analyse des QML-Codes möglich.
- **HILFE:** Hier finden Sie die komplette Dokumentation, die vom *Qt Assistant* registriert ist. Natürlich finden Sie hierbei auch gleich die Dokumentationen zu verschiedenen Qt-Tools (inklusive des Qt Creators). Die umfangreiche Dokumentation hat sehr viel dazu beigetragen, dass Qt mittlerweile eines der beliebtesten Frameworks überhaupt ist.

Unter den Arbeitsmodi finden Sie weitere Schaltflächen ❷, mit denen Sie Ihr Projekt ausführen, debuggen und erstellen können.

Die Ausgaben, die beim Bauen, Kompilieren und Linken gemacht werden, finden Sie unterhalb des Editors. Diese Informationen können Sie auch über die fünf Schaltflächen ❸ (BUILD-PROBLEME, SUCHERGEBNISSE, AUSGABE DER ANWENDUNG, KOMPIlierung und QML/JS-Konsole) am unteren Rand des Bildschirms aufrufen.

17.1.2 Ein einfaches Projekt erstellen

Programme mit einer grafischen Oberfläche können Sie mit Qt entweder explizit Zeile für Zeile schreiben oder mit dem Designer gestalten. In der Praxis verwendet man gerne eine Mischung aus beidem. Zur Einführung dürfte ein manuell erstellter Code zunächst noch hilfreicher und verständlicher sein. Der erste Schritt soll hier daher sein, ein leeres Qt-Projekt zu erstellen und zur Ausführung zu bringen.

Auch wenn die Bildschirmfotos hier auf einem Mac-System gemacht wurden, so funktioniert der Vorgang unter Windows und Linux ganz genauso, mit denselben Schritten.

1. Nach dem Sie Qt Creator gestartet haben, starten Sie ein neues Projekt über das Menü DATEI • NEU.
2. Wählen Sie im nächsten Dialog unter den Vorlagen ANDERES PROJEKT aus, so dass Sie in der Mitte LEERES QT-Projekt auswählen können. Klicken Sie anschließend auf die Schaltfläche AUSWÄHLEN (Abbildung 17.2).

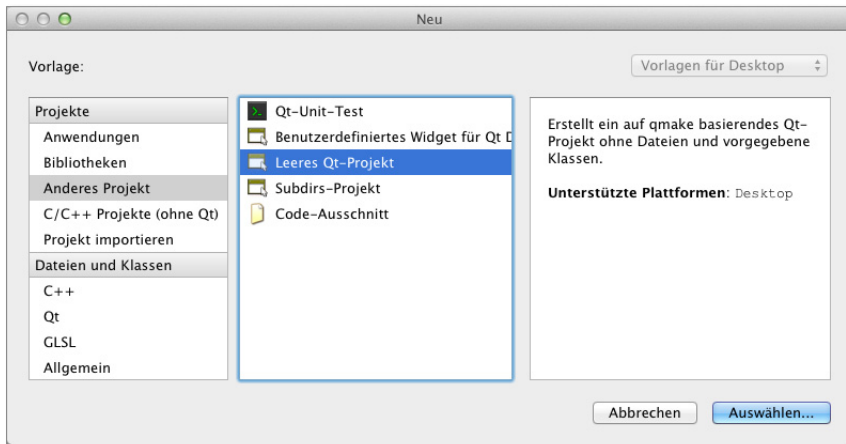


Abbildung 17.2 Ein leeres Qt-Projekt erstellen

3. Anschließend geben Sie einen Namen ein und wählen den Pfad aus, unter dem das Projekt gespeichert werden soll (Abbildung 17.3). Klicken Sie dann auf WEITER.

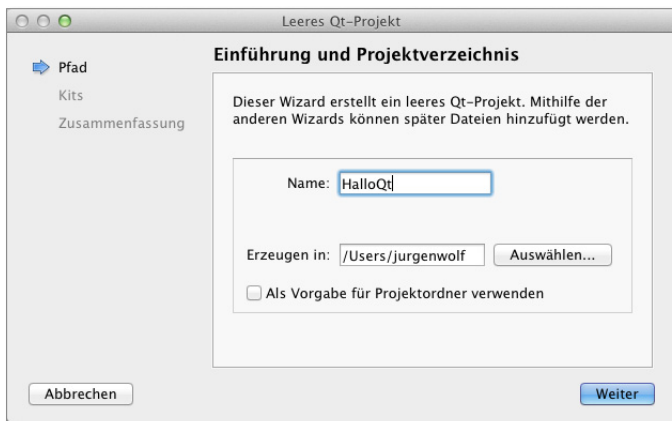


Abbildung 17.3 Verzeichnis und Namen für das Projekt auswählen

4. Im nächsten Fenster können Sie jetzt die Kits für das Projekt auswählen (Abbildung 17.4). Ein Kit wird für die Erstellung und Ausführung des Projekts benötigt. Darunter fallen verschiedene Qt-Versionen, verschiedene Compiler und weitere Einstellungen. In der Regel können Sie die Voreinstellungen auch stehen lassen.



Abbildung 17.4 Auswahl des Kits

5. Am Schluss können Sie Ihr Projekt noch einer Projektverwaltung unterstellen, und es wird alles nochmals in einer Zusammenfassung aufgelistet. Mit einem Klick auf die Schaltfläche **FERTIG** schließen Sie den Assistenten ab.
6. Jetzt erscheint der Editor im Modus **EDITIEREN**, wo Sie ein leeres Projekt mit der Projektdatei mit Endung `.pro` vorfinden. Diese Projektdatei wurde von `qmake` erstellt (und weiter verwaltet) und wird dazu benötigt, um auf dem jeweiligen System ein `Makefile` zu bauen. Als Nächstes wollen wir eine C++-Quelldatei mit unserem Qt-Code hinzufügen. Klicken Sie hierzu mit der rechten Maustaste auf den Projektordner, und wählen Sie im Kontextmenü den Befehl **HINZUFÜGEN** aus (Abbildung 17.5).

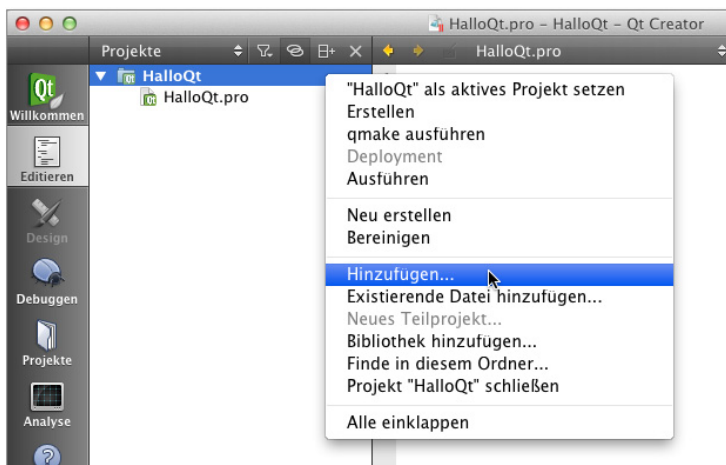


Abbildung 17.5 Eine C++-Quelldatei dem Projekt hinzufügen

7. Jetzt öffnet sich ein weiterer Assistent, wo Sie bei den Vorlagen C++ und dann C++-QUELLDATEI auswählen (Abbildung 17.6). Mit einem Klick auf AUSWÄHLEN geht es weiter.



Abbildung 17.6 Neue C++-Quelldatei auswählen

8. Neben dem Pfad, unter dem Sie die Datei speichern, müssen Sie hier auch den Namen für die C++-Datei angeben (Abbildung 17.7).



Abbildung 17.7 Namen und Pfad für die C++-Datei auswählen

9. Zum Schluss müssen Sie nur noch bei der PROJEKTVERWALTUNG die Quelldatei dem entsprechenden Projekt hinzufügen (Abbildung 17.8), damit *qmake* auch weiß, welche Datei es mit übersetzen muss. Zusätzlich können Sie auch hier eine Versionskontrolle für diese Datei der Verwaltung hinzufügen. Mit einem Klick auf die Schaltfläche FERTIG wird die Quelldatei dem Projekt hinzugefügt.

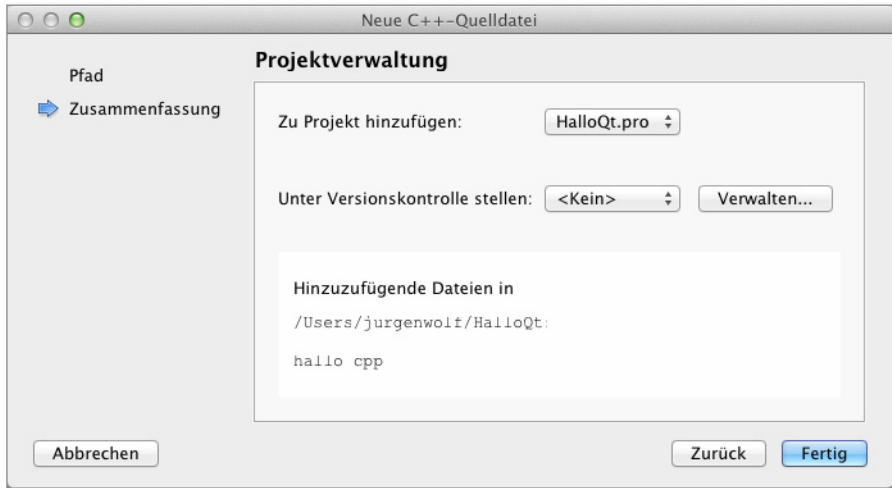


Abbildung 17.8 Dem Projekt die C++-Quelldatei hinzufügen

10. Jetzt finden Sie in der PROJEKTVERWALTUNG bei PROJEKTE die Quelldatei (hier *hallo.cpp*) wieder. Wenn Sie diese Datei doppelklicken, können Sie den Quelltext eingeben (Abbildung 17.9).

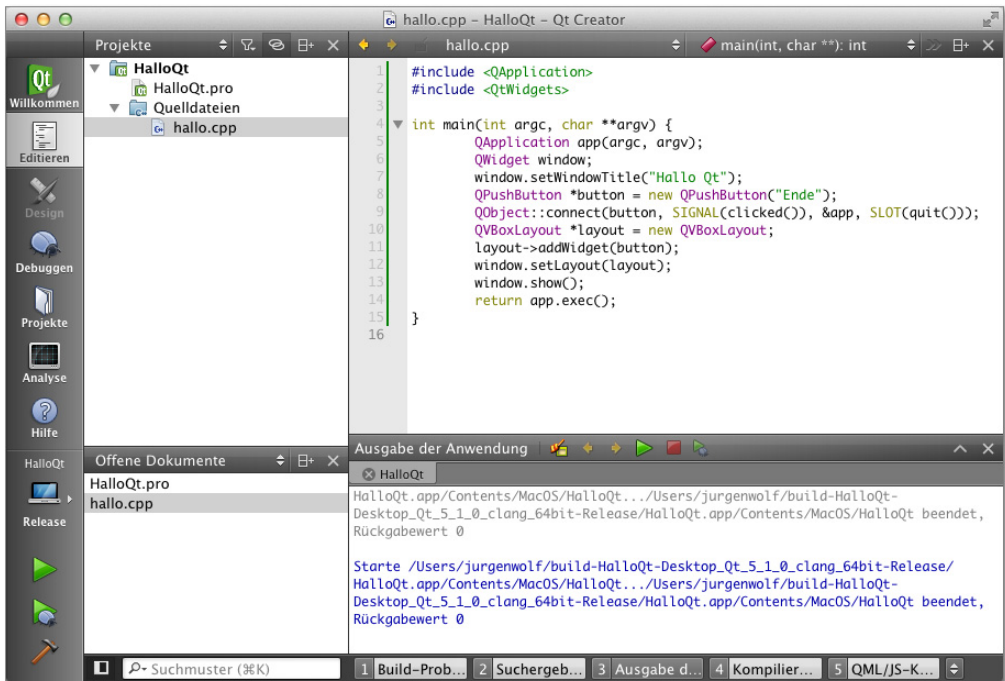


Abbildung 17.9 Bereit für die Eingabe des Quelltextes

Für unser erstes Beispiel wurde folgender Quelltext verwendet:

```

00 // kapitel17/qt_example01/hallo.cpp
01 #include <QApplication>
02 #include <QtWidgets>

03 int main(int argc, char **argv) {
04     QApplication app(argc, argv);
05     QWidget window;
06     window.setWindowTitle("Hallo Qt");
07     QPushButton *button = new QPushButton("Ende");
08     QObject::connect( button, SIGNAL(clicked()),
                       &app, SLOT(quit()));
09     QVBoxLayout *layout = new QVBoxLayout;
10     layout->addWidget(button);
11     window.setLayout(layout);
12     window.show();
13     return app.exec();
14 }
```

Hierzu eine grobe Beschreibung des Quellcodes: Die Basis eines jeden Qt-Programms startet mit `QApplication` in Zeile 04 und endet mit Zeile 13. Diese beiden Zeilen sind die Mindestvoraussetzung für eine Qt-Anwendung, weil damit die Ereignisschleife (engl. *event loop*) gestartet und auf die Eingabe vom Anwender gewartet wird.

In Zeile 05 erzeugen wir ein neues Fenster, welches in Zeile 06 einen Titel bekommt. In Zeile 07 erzeugen wir eine neue Schaltfläche mit dem Text »Ende«. Mit Zeile 08 richten wir eine Signal-Slot-Verbindung für die Schaltfläche ein. Und zwar reagiert diese Verbindung, sobald der Button angeklickt (`SIGNAL(clicked())`) wird, mit dem die Beendigung der Qt-Anwendung eingeleitet werden soll, indem die Methode `quit()` an `QApplication` gesendet wird. In den Zeilen 09 bis 11 stecken wir die Schaltfläche in ein horizontales Layout und packen es in das Fenster, welches mit Zeile 12 angezeigt wird.

Fehlende Bibliotheken in der .pro-Datei (No such file or directory)

Beim Erstellen eines leeren Qt-Projekts kann es passieren, dass `qmake` die nötigen Qt-Module nicht hinzulinkt. In unserem Beispiel würden die Module *Qt Core*, *Qt GUI* und *Qt Widgets* benötigt. Sollten Sie Probleme mit dem Übersetzen haben, brauchen Sie lediglich in der *.pro*-Datei folgende Zeile einzufügen:

```
QT      += core gui widgets
```


Fehlendes OpenGL unter Linux

Bei Linux kann sich der Compiler auch beschweren, dass dieser kein *GL/gl.h* oder *-lGL* finden kann, also die OpenGL-Headerdatei bzw. -Bibliothek. In dem Fall müssen Sie das OpenGL-Paket zur Entwicklung nachinstallieren. Bei Linux finden Sie dieses als Mesa-Paket vor (bspw. *libglu1-mesa-dev*).

Das Programm können Sie jetzt mit einem Klick auf die Schaltfläche AUSFÜHREN übersetzen und gleich ausführen lassen. In Abbildung 17.10 bis Abbildung 17.12 sehen Sie das Programm bei der Ausführung.

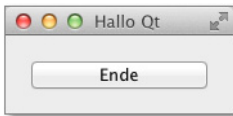


Abbildung 17.10 Das Programm unter Mac OS X

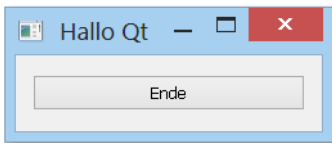


Abbildung 17.11 Das Programm unter Windows

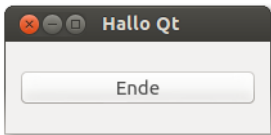


Abbildung 17.12 Das Programm unter Linux (Ubuntu)

Windows und seine DLLs

Natürlich gilt auch hier: Sofern Sie die Qt-Beispiele unter MS-Windows aus einem Verzeichnis heraus mit einem Mausklick ausführen wollen, müssen auch die entsprechenden DLLs von Qt für das Programm vorhanden sein. Sollten Sie eine Fehlermeldung bekommen wie »Qt...dll auf dem Computer fehlt«, dann kopieren Sie entweder die nötigen DLLs ins Systemverzeichnis von MS-Windows (bspw. *C:\WINDOWS\SYSTEM*) oder in das Verzeichnis, in dem das Programm ausgeführt wird.

Auch wenn hier gezeigt wurde, wie Sie Qt-Anwendungen mit der Entwicklungsumgebung Qt Creator entwerfen können, so sind Sie nicht davon abhängig. Theoretisch

könnten Sie auch in der Kommandozeile den Code mit dem typischen Dreisatz übersetzen:

```
qmake -project
qmake
make
```

In diesem Buch wird allerdings nicht weiter darauf eingegangen.

C++11-Features verwenden

Bei einem ersten Blick auf den Code fällt wohl gleich das nackte `new` zur Speicheranforderung auf. Die ersten Fragen dürften sein, was mit den Smart Pointern ist und ob Qt sich überhaupt mit C++11 verträgt. Eine pauschale Antwort zu geben, ist hier noch nicht einfach. Zumindest sind mit Qt5 schon einige neue C++11-Dinge verfügbar. Beispielsweise können Sie die vereinheitlichte Initialisierung fast schon für alle Typen verwenden. Auch die Move-Semantik wurde für ein paar Typen implementiert. `constexpr` (in Qt mit dem Schlüsselwort `Q_DECL_CONSTEXPR`) ist ebenfalls schon bei vielen Typen vorhanden. Auch das neue `noexcept` wurde an einigen zentralen Stellen hinzugefügt. Die neuen Schlüsselwörter von C++11, die in Qt unterstützt werden, beginnen gewöhnlich mit `Q_DECL_`. Eine Suche mit dem Qt Assistant listet Ihnen dann mehrere dieser Schlüsselwörter auf. Der Vorteil der Qt-Versionen wie bspw. `QT_DECL_CONSTEXPR` gegenüber `constexpr` liegt darin, dass die `QT_DECL_`-Versionen nur verwendet werden, wenn die Anwendung mit einem C++11-Compiler übersetzt wird. Ältere Compiler ignorieren diese `QT_DECL_`-Versionen.

Dies sind nur einige der Dinge, die unterstützt werden. Vieles fehlt allerdings noch, aber ich denke mir, dass diese Features mit den nächsten Versionen von Qt implementiert werden. Dass es trotzdem etwas länger mit der Implementierung von C++11 in Qt dauert, liegt auf der Hand. So will man nicht die Abwärtskompatibilität zur Vorgängerversion von C++ kaputt machen. Der Hauptfokus des Qt-Frameworks liegt zunächst auf der Stabilität, und nicht darauf, mit Gewalt mehr C++11-Features hinzuzufügen.

Und was die Speicherreservierung mit `new` betrifft, können Sie die gerne mit `std::unique_ptr` austauschen, sofern das auf Ihrem System mit Qt und einem C++11-Compiler funktioniert. So könnten Sie durchaus auch Folgendes verwenden:

```
00 // kapitel17/qt_example01_cpp11/hallo.cpp
01 #include <QApplication>
02 #include <QtWidgets>
03 #include <memory> // für std::unique_ptr

04 int main(int argc, char **argv) {
05     QApplication app(argc, argv);
06     QWidget window;
```

```

07  window.setWindowTitle("Hallo Qt");
08  std::unique_ptr<QPushButton> button(
        new QPushButton{"Ende"});
09  QObject::connect( button.get(), SIGNAL(clicked()),
        &app, SLOT(quit()));
10  std::unique_ptr<QVBoxLayout> layout(new QVBoxLayout);
11  layout->addWidget(button.get());
12  window.setLayout(layout.get());
13  window.show();
14  return app.exec();
15 }

```

Damit *qmake* auch C++11 kennt, müssen Sie hier außerdem zur Projektdatei folgende Zeile hinzufügen:

```
QMAKE_CXXFLAGS += -std=c++11
```

Wie allerdings bereits erwähnt, klappt es noch nicht überall durchgängig, neue C++11-Sachen mit Qt zu verwenden. Aus diesem Grund sind die Listings hier alle noch im klassischen C++-Stil gehalten. Bevor Sie sich mit C++11 und Qt befassen, sollten Sie sich ohnehin erst einmal mit dem Qt-Framework vertraut machen.

17.2 Signale und Slots

Im Prinzip läuft die GUI-Programmierung mit Qt immer recht ähnlich ab. Wenn sich bspw. der Zustand eines Kontrollelements (bei Qt auch Widget genannt) verändert hat (etwa eine Schaltfläche wurde angeklickt), will man ein anderes Kontrollelement darüber informieren. Die grafischen Kontrollelemente werden also im Programm mit einer Funktion verknüpft, welche ausgeführt wird, sobald der Anwender dieses, bspw. durch einen Mausklick, aktiviert hat. Drückt der Anwender etwa eine ÖFFNEN-Schaltfläche, wird eine entsprechende Funktion aufgerufen, die gegebenenfalls einen neuen Dialog zum Öffnen einer Datei präsentiert.

Viele grafische Toolkits verwenden häufig zur Kommunikation zwischen den Widgets eine Callback-Funktion. Ein solcher Callback ist nichts anderes als ein Zeiger auf eine Funktion. Allerdings haben solche Callbacks zwei kleine Makel. Zum einen sind sie nicht typsicher. Man ist dabei nie sicher, dass die gerade ausführende Funktion den Callback mit den richtigen Argumenten aufruft. Und zweitens ist der Callback fest mit der ausführenden Funktion verbunden, weil diese Funktion wissen muss, welcher Callback aufgerufen werden soll.

Qt geht für die Kommunikation zwischen den Kontrollelementen mit dem *Signal-und-Slot-Konzept* einen anderen Weg. Beim Signal-und-Slot-Konzept hat man den

Vorteil, dass Qt die Verbindung automatisch trennt, wenn eines der kommunizierenden Objekte zerstört wird. Dadurch werden viele Abstürze vermieden, weil Versuche, auf ein nicht vorhandenes Objekt zuzugreifen, nicht mehr möglich ist.

Qt-Signale vs. UNIX/POSIX-Signale

An dieser Stelle muss kurz darauf hingewiesen werden, dass die Qt-Signale nichts mit den Signalen von UNIX/Posix gemeinsam haben. Dies sollte kurz erwähnt werden, damit sich ein Linux-Kenner nicht auf der falschen Fährte wiederfindet.

17.2.1 Verbindung zwischen Signal und Slot herstellen

Am einfachsten lässt sich das Signal-und-Slot-Konzept anhand unseres ersten Beispiels *hallo.cpp* beschreiben. Das Beispiel wollen wir nun erweitern, indem wir beim Anklicken des Buttons die Anwendung beenden. Hier die entsprechende Codezeile:

```
08    QObject::connect( button, SIGNAL( clicked() ),
                        &app,  SLOT( quit() ) );
```

`connect()` ist eine statische (static) Methode, welche eine Verbindung zwischen einem Signal und einem Slot herstellt. Hierzu eine genauere Syntax:

```
MetaObject::Connection QObject::connect (
    const QObject * sender, const char * signal,
    const QObject * receiver, const char * method,
    Qt::ConnectionType type = Qt::AutoCompatConnection );
```

Mit `sender` legen Sie das Qt-Objekt fest, welches das Signal aussendet. Mit `signal` wird das Signal angegeben, welches abgefangen werden soll. `receiver` ist dann die Instanz einer Klasse, welche die Methode `method` aufruft, wenn das Signal `signal` von `sender` ausgesendet wurde. Zusammengefasst: Empfängt das Objekt `sender` das Signal `signal`, wird es an den Slot gebunden. Dann nimmt das Objekt `receiver` dieses vom Slot entgegen und führt `method` aus. Bezieht man dies auf die folgende Zeile

```
08    QObject::connect( button, SIGNAL( clicked() ),
                        &app,  SLOT( quit() ) );
```

verbinden Sie hiermit das Objekt `button` von der Klasse `QPushButton` mit dem Objekt `app` von der Klasse `QApplication`. Als Aktion legen Sie fest, sollte bei der Schaltfläche `button` das Signal `clicked()` eingegangen sein, wird bei der Klasse `QApplication` die Methode `quit()` ausgeführt, was in diesem Fall das Ende des Programms wäre.

`QObject::connect()` gibt den `Handle Connection()` der Signal-Slot-Verbindung zurück. Dieser `Handle` kann verwendet werden, um eine Signal-Slot-Verbindung

wieder zu trennen oder um zu überprüfen, ob eine Verbindung erfolgreich hergestellt werden konnte. Außerdem müssen Sie die Makros `SIGNAL()` und `SLOT()` verwenden, wenn Sie das Signal und die Methode festlegen wollen, weil für die beiden Argumente eine Zeichenkette vorgesehen ist und diese beiden Makros dafür sorgen, dass eine korrekte Zeichenkette eingesetzt wird.

Weiteres optionales Argument

Das letzte Argument `type` von `QObject::connect()` ist optional und beschreibt den Typ der Verbindung, auf dem diese aufbaut.

Somit handelt es sich bei den Slots um ganz normale Methoden einer Klasse, die auf Signale einer anderen Klasse reagieren, wobei die Signale im Grunde wiederum nur einfache Methoden einer Klasse sind. Natürlich ist es auch möglich, einen Sender mit mehreren Empfängern und umgekehrt zu verknüpfen. Ist bspw. ein Signal mit zwei oder mehreren Slots verbunden, so werden die Slots der Reihe nach ausgeführt, wie diese im Code geschrieben wurden. Jede Unterklasse von `QObject` kann somit solche Signale und Slots definieren.

Überladenes `connect()`

Es existiert auch ein überladenes `connect()`, das neben der String-basierenden Syntax, zwei `QObject`s mit nicht String-basierenden Objekten verbindet. Ebenso ist es damit auch möglich, eine einfache Funktion mit `connect()` zu verbinden. Mehr dazu entnehmen Sie – wie immer – der Dokumentation von Qt mithilfe des Qt-Assistenten.

17.2.2 Signal und Slot mithilfe der Qt-Referenz ermitteln

Sicherlich stellt sich für Sie jetzt die Frage, woher Sie die Informationen bekommen, welche Signale und/oder Slots Sie wofür verwenden können. Hierzu sind zwar tiefere Kenntnisse von Qt und dessen Klassen von Vorteil, aber die einzelnen Signale und Slots kann man unmöglich auswendig wissen. Für solche Fälle ist es ratsam, den Qt Assistant über den HILFE-Modus zu verwenden, welcher beim SDK mitinstalliert wurde (Abbildung 17.13).

Wollen Sie bspw. ermitteln, welche Signale bei der Klasse `QPushButton` auftreten können, müssen Sie zunächst im Indexverzeichnis von Qt Assistant nach der entsprechenden Klasse suchen (Abbildung 17.14).

Als Ergebnis erhalten Sie die Klassenreferenz zu `QPushButton` mit all ihren Eigenschaften, Methoden, Signalen und Slots (Abbildung 17.15). Wichtig ist hierbei auch der Inhalt von geerbten Mitgliedern.

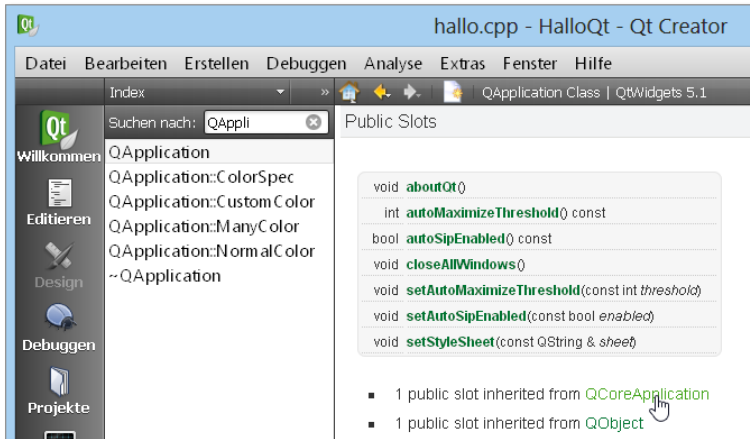


Abbildung 17.13 Unverzichtbar, um Qt-Anwendungen zu entwickeln, ist der Modus »Hilfe«, in dem Qt Assistant nahtlos in die Entwicklungsumgebung integriert wurde.

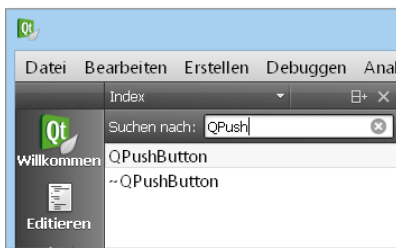


Abbildung 17.14 Mithilfe des Index auf der Suche nach der Dokumentation zur entsprechenden Klasse

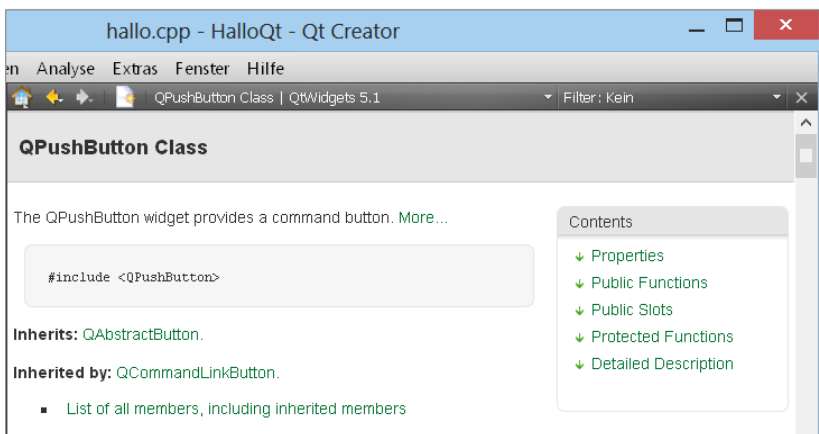


Abbildung 17.15 Komplette Dokumentation mit Referenz aller Methoden und Funktionen zu »QPushButton«

Bei der Durchsicht der Dokumentation von `QPushButton` fällt auf, dass sich hier zwar öffentliche Slots befinden, aber keine direkten Signale. Erst bei den geerbten Mitgliedern finden Sie einen Eintrag (mit Verweis), wie bspw. »4 signals inherited from `QAbstractButton`«.

Also folgen Sie dem Link zur Klasse `QAbstractButton`, und siehe da, Sie finden die gewünschten Signale, die ja auch an `QPushButton` weitervererbt wurden (da public). Neben `clicked()` finden Sie hier auch noch die Signale `pressed()`, `released()` und `toggled()`. Des Weiteren finden Sie hier noch zwei weitere Signale, die je von `QWidget` und `QObject` geerbt wurden (Abbildung 17.16).

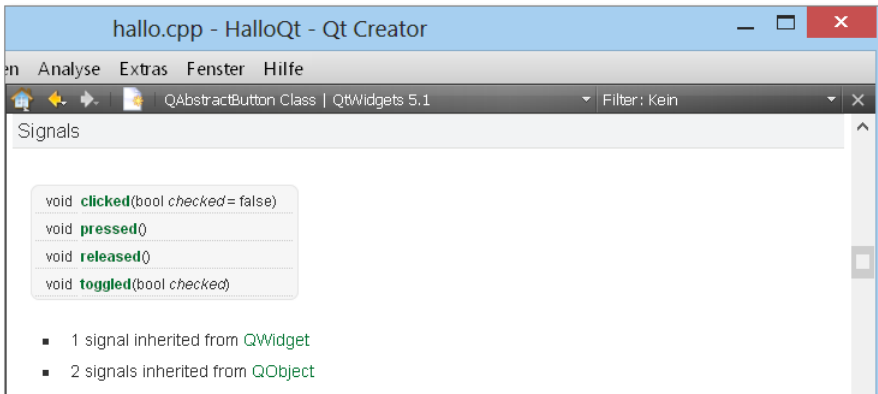


Abbildung 17.16 Geerbte Signale der Klasse »`QAbstractButton`«

Wollen Sie noch mehr zu einem Signal erfahren, folgen Sie einfach wieder dem entsprechenden Verweis.

Im Beispiel zuvor haben wir auch einen Slot (`quit()`) der Klasse `QApplication` verwendet. Um auch hier an den (die) entsprechenden Slot(s) einer Klasse zu kommen, gehen Sie ebenso vor, wie schon für die Signale vorhin beschrieben wurde, und nehmen den Qt Assistant zu Hilfe. In der Klassenreferenz von `QApplication` finden Sie dann unter dem Eintrag PUBLIC SLOTS die entsprechenden Slots (Abbildung 17.17).

Zwar sind hier mehrere vorhandene Slots definiert, nicht aber der Slot `quit()`. Auch hier wurde der Slot `quit()` von einer anderen Klasse geerbt, in dem Fall von `QCoreApplication`. Dies können Sie dem entsprechenden Verweis »1 public slot inherited from `QCoreApplication`« entnehmen (Abbildung 17.18).

Die Verwendung des Qt Assistants ist unerlässlich für jeden ernst zu nehmenden Qt-Programmierer. Das Prinzip ist immer dasselbe, man sucht nach der entsprechenden Klasse (Widget) im Indexverzeichnis und dann eben nach dem entsprechenden Signal, Slot, der Eigenschaft oder eben der entsprechenden Methode, die man mit einem Widget ausführen will.

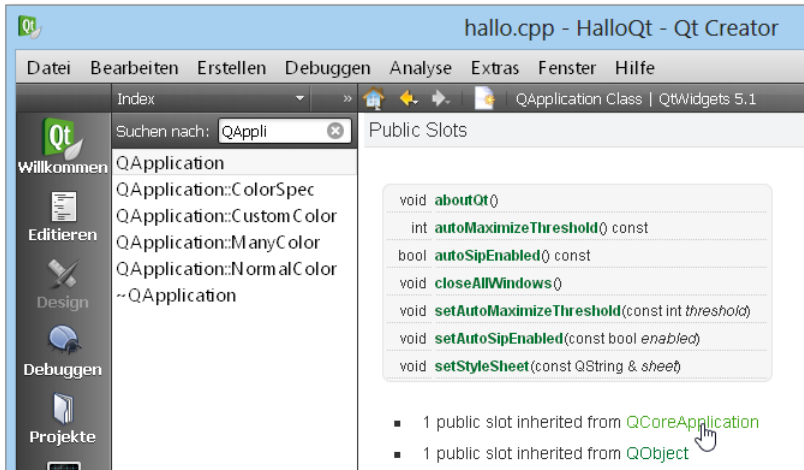


Abbildung 17.17 Klassenreferenz von »QApplication«

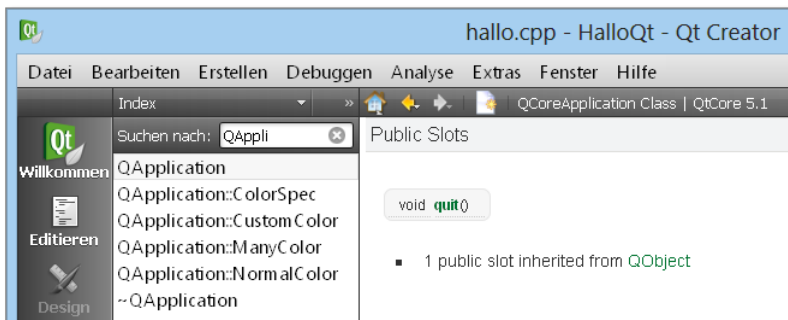


Abbildung 17.18 Slot gefunden in der Referenz von »QCoreApplication«

Signal-Slot-Verbindung mit dem Designer

Noch einfacher wird es, die passende Signal-Slot-Verbindung herzustellen, wenn Sie Anwendungen im Modus DESIGN erstellen. Hier brauchen Sie nur noch im wahrsten Sinne des Wortes die Signal-Slot-Verbindung zwischen den Widgets auszuwählen und »zusammenzuklicken«.

Gegenseitiges Signal- und Slot-Konzept

Häufig kommt es vor, dass zwei Widgets voneinander abhängig sind. Wird bspw. ein Widget verändert, so muss das andere ebenfalls an diese Veränderung angepasst werden – und umgekehrt genauso.

Im folgenden Beispiel wurden die Klassen `QSpinBox` (Eingabefeld mit Pfeiltasten) und `QSlider` (ein Schieberegler) verwendet, welche beide jeweils bei einer Veränderung den Wert des anderen Widgets anpassen müssen. Wurde bspw. der Wert des Eingabefeldes verändert, wird das Signal `valueChanged()` ausgelöst. Ist dies der Fall, soll hier bei dem Schieberegler (`QSlider`) der entsprechende Wert des Eingabefeldes mit dem Slot `setValue()` angepasst werden. Andersherum haben Sie denselben Fall. `valueChanged()` ist jeweils das Signal und `setValue()` der Slot der beiden Widgets. Wie Sie weitere Signale und Slots in Erfahrung bringen können, wurde im Abschnitt zuvor beschrieben. Hier nun unser Beispiel:

```

00 // kapitel17/qt_example02/signal_slot01.cpp
01 #include <QApplication>
02 #include <QtWidgets>

03 int main(int argc, char **argv) {
04     QApplication app(argc, argv);
05     QWidget* win      = new QWidget;
06     win->setWindowTitle("Signal-Slot-Verbindungen");
07     // Neues Eingabefeld
08     QSpinBox* spin    = new QSpinBox;
09     spin->setMinimum(0); // Minimum
10     spin->setMaximum(100); // Maximum
11     // Neuer Schieberegler
12     QSlider* slider   = new QSlider(Qt::Horizontal);
13     slider->setMinimum(0); // Minimum
14     slider->setMaximum(100); // Maximum
15     // Neue horizontale Box
16     QVBoxLayout* layout = new QVBoxLayout(win);
17     layout->addWidget(spin);
18     layout->addWidget(slider);
19     // Signal-und-Slot Verbindungen einrichten
20     QObject::connect( spin,    SIGNAL( valueChanged(int) ),
                       slider, SLOT( setValue(int) ) );
21     QObject::connect( slider, SIGNAL( valueChanged(int) ),
                       spin,    SLOT( setValue(int) ) );
22     win->show();
23     return app.exec();
24 }

```

Abbildung 17.19 zeigt das Programm bei der Ausführung.

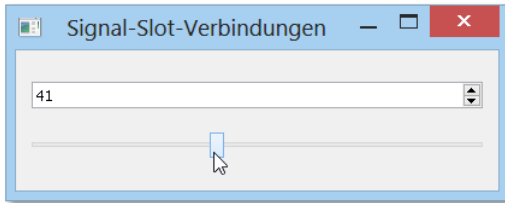


Abbildung 17.19 Hier wird der Wert der Eingabebox mit den Pfeilen an den gerade verschobenen Schieberegler angepasst. Umgekehrt verhält es sich genauso zwischen den beiden Widgets.

Um diese beiden Widgets miteinander zu verbinden, waren im Grunde nur zwei Verbindungen mit `connect()` nötig, die hier mit den Zeilen 20 und 21 realisiert wurden. Verändern Sie hierbei den Schieberegler, wird automatisch auch die Eingabebox mit entsprechendem Wert versehen/verändert. Verändern Sie die Eingabebox mit den Pfeilen oder als gültigen Wert durch die Tastatureingabe, wird auch der Schieberegler entsprechend an die Position angepasst. Voraussetzung dafür, dass dies auch sinnvoll funktioniert, ist, dass beide Werte auch denselben Minimal- und Maximalwert habe, welche hier in den Zeilen 09 und 10 für das Eingabefeld und 13 und 14 für den Schieberegler auf minimal 0 und auf maximal 100 gesetzt wurden.

Argumente von Signal-Slot-Verbindungen

Im Beispiel zuvor hatten die Signal-Slot-Verbindungen beide dieselben Argumente, also sowohl `valueChanged(int)` als auch `setValue(int)` waren vom Typ `int`. Dies muss unbedingt beachtet werden, weil bei einer solchen Signal-Slot-Verbindung keine automatische Typumwandlung erfolgt. Bei einem `connect()` muss die Funktionssignatur der Signal-Slot-Verbindung übereinstimmen.

Benötigen Sie bspw. einen String statt eines Integer-Wertes, so müssen Sie von der Klasse ableiten und den entsprechenden Slot implementieren. Hierbei konvertiert man im Grunde nur das Argument und ruft anschließend den eigentlichen Slot auf. In Qt kann man nicht einfach beliebige Funktionen als Slots verwenden: Diese müssen zuerst als solche gekennzeichnet werden. Aber dazu folgt im nächsten Abschnitt noch mehr.

Es ist außerdem nicht möglich, eine Verbindung herzustellen, wenn ein Slot mehr Argumente erwartet, als das Signal beinhaltet. Einfachstes Beispiel:

```
QObject::connect( button, SIGNAL( clicked() ),
                  label, SLOT( setText("Neuer Text") ) );
```

Hier wird versucht, eine Verbindung zwischen einer Schaltfläche `button` und einem Text-Label `label` herzustellen. Sollte die Schaltfläche gedrückt werden, dann soll das

Text-Label mit einem neuen Text versehen werden. Allerdings funktioniert dies hier nicht, weil das Signal `clicked()` kein Argument enthält, aber der Slot `setText()` eines erwartet.

Der Rückgabewert von `connect()` ist ein Handler vom Typ `QObject::Connection` auf die Verbindung, wenn eine zwischen dem Signal und dem Slot hergestellt werden konnte. Ist es nicht möglich, eine gültige Verbindung herzustellen, können Sie den Rückgabewert vom Typ `QObject::Connection` auf Gültigkeit hin überprüfen, indem Sie diesen in ein `bool` konvertieren. Zum Beispiel:

```
if( ! (QObject::connect(
    button, SIGNAL( clicked() ),
    label, SLOT( setText("Neuer Text") ) ) ) ) {
    // Fehler bei der Signal-Slot-Verbindung
}
```

Eigene Klasse mit Signalen und Slots definieren bzw. erweitern

Um eine Klasse mit eigenen Signalen und Slots zu versehen, gilt es, folgende Regeln einzuhalten:

- ▶ Eigene Signale und Slots können nur in Klassen definiert werden, die von der Klasse `QObject` abgeleitet sind. Da `QObject` die Basisklasse aller Objekte ist, sollten Sie, speziell bei den GUI-Elementen, keine Probleme damit bekommen. Im Falle einer Mehrfachvererbung muss `QObject` oder die davon abgeleitete Klasse als erste Elternklasse notiert werden.
- ▶ In der neu definierten Klasse muss das Makro `Q_OBJECT` (ohne Semikolon) gesetzt werden (mehr dazu gleich).
- ▶ Slots werden im Grunde wie gewöhnliche Methoden implementiert und aufgerufen. Nur müssen Sie hierfür das Schlüsselwort `slots` nach einem Zugriffsspezifizierer (`public`, `private`, `protected`) verwenden.
- ▶ Signale hingegen müssen mit dem Schlüsselwort `signals` ohne einen Zugriffsspezifizierer gekennzeichnet werden. Beachten Sie außerdem, dass Signale niemals implementiert, sondern lediglich deklariert werden. Zur besseren und deutlicheren Lesbarkeit kann (muss aber nicht) ein solches Signal mit dem Schlüsselwort `emit` ausgelöst werden.

Am besten sehen wir uns diese Aufzählungspunkte in der Praxis an:

```
01 class MyClass : public QObject {
02     Q_OBJECT
03     public:
04         // Konstruktor
05         MyClass();
```

```

06      // Zugriffsmethoden
07      int value() const { return val; }
08      public slots:
09          void setValue(int);
10      signals:
11          void valueChanged(int);
12      private:
13          int val;
14  };

```

Nachdem Sie Ihre Klasse in Zeile **01** von `QObject` abgeleitet haben, können Sie Signale und Slots im Grunde wie normale Methoden deklarieren.

Wichtig ist auch das Makro `Q_OBJECT` in Zeile **02** (Achtung, ohne Semikolon!). In der Deklaration der Signal- und Slot-Elementfunktionen werden auch die neuen Spezifizierer `slots` (Zeile **08**) und `signals` (Zeile **10**) verwendet. Abhängig von der gewünschten Sichtbarkeit nach außen bzw. der Weitervererbbarkeit werden noch die Spezifizierer `private`, `public` und `protected` vor `slots` gesetzt.

Von Ihnen als Programmierer werden nur die Slots als gewöhnliche Methoden definiert. Einsteiger in Qt, die Quellcode studieren, sind häufig verwirrt, weil Sie die Definition zu Signal-Methoden nicht finden. Der Code für Signal-Methoden wird aber nicht vom Programmierer geschrieben, sondern dies erledigt ein in Qt verwendeter Präprozessor, der *Meta Object Compiler* (kurz MOC).

Meta Object Compiler (MOC)

MOC ist ein in Qt verwendeter Präprozessor, der C++ um Fertigkeiten erweitert, welche der Sprachstandard nicht unterstützt, wie bspw. Signale und Slots. Vereinfacht heißt dies, der MOC liest einen Code ein, und generiert daraus einen C++-Code. Der so vom MOC erzeugte C++-Code kann dann mit einem üblichen Compiler übersetzt werden. Daher ist für die Nutzung des Signal-Slots-Konzeptes die Verwendung des MOC-Präprozessors unbedingt nötig, da dieser die zur Prüfung nötigen Laufzeitinformationen zu den entsprechenden Klassen hinzufügt.

Die Datei, die MOC dabei erzeugt, haben häufig das Präfix *moc_*. Somit würde bspw. aus Dateien wie *myclass.h* und *myclass.cpp* die Datei *moc_myclass.cpp* erzeugt. Allerdings bekommen Sie diese vom MOC generierten Dateien nur auf ausdrücklichen Befehl für den MOC zu Gesicht (cmdline-Argument für den MOC).

Hierzu ein echtes Beispiel, welches allerdings zunächst noch keine »richtigen« GUI-Elemente verwendet. Wir erstellen eine einfache Klasse mit einem Integer-Wert als Eigenschaft. In dieser Klasse wird das Signal `valueChanged(int)` deklariert, welches ausgelöst wird, sobald der Integer-Wert des Objekts verändert wird. Natürlich imple-

mentieren wir hierbei auch den entsprechenden Slot `setValue(int)`, um zwei Objekte der Klasse mit `connect()` verknüpfen zu können. Hier zunächst die Deklaration der Klasse `MyClass`:

```
00 // kapitel17/signalslot1/myclass.h
01 #ifndef MY_CLASS_H
02 #define MY_CLASS_H
03 #include <QObject>

04 // Eine Klasse, die Signals und Slots besitzt.
05 class MyClass : public QObject {
06     Q_OBJECT
07     public:
08         MyClass(); // Konstruktor
09         int value() const { return val; }
10     public slots:
11         // Der Wert von "val" wird geändert.
12         void setValue( int );
13     signals:
14         // Das Signal soll ausgesandt werden,
15         // wenn "val" geändert wird.
16         void valueChanged( int );
17     private:
18         int val;
19 };
20 #endif
```

Dem Projekt weitere (Quell- und Header-)Dateien hinzufügen

Wie Sie dem Projekt eine Quelldatei hinzufügen können, haben Sie bereits in der Anleitung in Abschnitt 17.1.2, »Ein einfaches Projekt erstellen«, ab Arbeitsschritt 6 gesehen. Genauso gehen Sie auch vor, wenn Sie eine Headerdatei hinzufügen, nur dass Sie eben in Arbeitsschritt 7 eine C++-Headerdatei statt einer C++-Quelldatei auswählen.

Die Definition der Klasse ist ebenso einfach. Da wir das Signal nicht definieren müssen, reichen in diesem Fall der Konstruktor und die Slot-Methode `setValue(int)` aus:

```
00 // kapitel17/signalslot1/myclass.cpp
01 #include "myclass.h"

02 MyClass::MyClass() {
03     val = 0;
04 }
```

```

05 void MyClass::setValue( int v ) {
06     // val wird nur neu gesetzt,
07     // wenn tatsächlich ein anderer Wert übergeben wird.
08     if ( v != val ) {
09         val = v;
10         emit valueChanged(v);
11     }
12 }

```

Auffällig ist hier Zeile 10 mit dem `emit`-Signalbezeichner. Mit dem Bezeichner `emit` machen Sie deutlich, dass es sich bei dem Aufruf nicht um einen normalen Funktionsaufruf handelt, sondern um einen Signalaufruf. Allerdings erreichen Sie Selbiges auch ohne `emit`. Der Bezeichner ist nur dafür gedacht, dass man sofort erkennen kann, dass es sich hierbei um einen Signalaufruf handelt.

Hierzu folgt noch eine Hauptfunktion, welche, abgesehen von den Nachrichtenboxen (`QMessageBox`), auch ohne GUI-Elemente auskommt und das eigene Signal-Slot-Konzept demonstriert:

```

00 // kapitel17/signalslot1/main.cpp
01 #include "myclass.h"
02 #include <QApplication>
03 #include <QMessageBox>
04 #include <QString>

05 // Eine einfache Nachrichtenbox
06 void MyMessageBox(MyClass& a, MyClass& b, QString title) {
07     QString Qstr, Qval;

08     Qstr.append(title); Qstr.append("\na: ");
09     Qval.setNum(a.value()); Qstr.append(Qval);
10     Qstr.append("\nb: "); Qval.setNum(b.value());
11     Qstr.append(Qval);
12     QMessageBox::information(
13         0, "MyClass Information", Qstr, QMessageBox::Ok);
13 }

14 int main(int argc, char **argv) {
15     QApplication app(argc, argv);
16     // Zwei MyClass-Objekte
17     MyClass *a = new MyClass();
18     MyClass *b = new MyClass();

19     // Das Signal des einen wird mit dem Slot

```

```

20 // des anderen Objekts verbunden.
21 QObject::connect( a, SIGNAL( valueChanged(int) ),
                   b, SLOT( setValue(int) ) );

22 // b.val bekommt den Wert 100
23 b->setValue( 100 );
24 QMessageBox(*a, *b, "b->setValue(100)");
25 // a.val bekommt den Wert 99. Durch die Signal-Slot-
27 // Verknüpfung bekommt jetzt auch b.val den Wert 99
28 a->setValue( 99 );
29 // Der Beweis
30 QMessageBox(*a, *b, "a->setValue(99)");
31 return app.exec();
32 }

```

Die Zeilen 05 bis 13 können Sie hierbei ignorieren. Damit wird nur eine Nachrichtenbox auf dem Bildschirm ausgegeben, mit dem Inhalt der Integer-Werte der beiden Objekte, die in den Zeilen 17 und 18 neu erzeugt werden.

In Zeile 21 verknüpfen wir dann die beiden Objekte, und zwar wird der Wert von Objekt a verändert, so dass das Signal `valueChanged()` ausgelöst und mit Objekt b und dem Slot `setValue()` verbunden wird. Sobald Sie also den Wert vom Objekt a verändern, wird der Wert von Objekt b durch unseren selbst erstellten Slot `setValue()` angepasst. In diesem Beispiel erhält das Objekt b durch `setValue()` denselben Wert wie Objekt a.

In Zeile 23 bekommt das Objekt b mit 100 einen neuen Wert zugewiesen. Hierbei haben wir keine Signal-Slot-Verknüpfung eingerichtet, so dass a und b folgende Werte besitzen, die Sie in Abbildung 17.20 sehen.

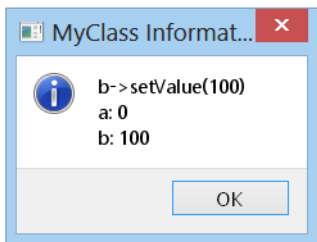


Abbildung 17.20 Nachdem Objekt b einen Wert erhalten hat

In Zeile 28 weisen wir dann Objekt a mit 99 einen neuen Wert zu. Und hierbei wird auch gleich unsere eingerichtete Signal-Slot-Verbindung aktiv, so dass auch Objekt b denselben Wert erhält, wie Objekt a. Dadurch ergeben sich die Werte, die Sie in Abbildung 17.21 sehen.

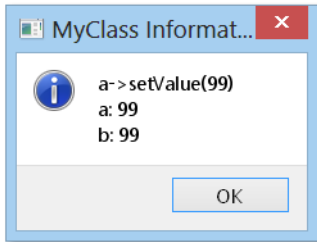


Abbildung 17.21 Das Signal wurde ausgelöst.

Ein Widget mit einem eigenen Slot

Häufig wird man seinen Widgets eigene Slots hinzufügen wollen. Ein gerne zitiertes Beispiel zeigt der folgende Codeausschnitt:

```
QObject::connect( button, SIGNAL( clicked() ),
                 label, SLOT( setText("neuer Text") ) );
```

Hierbei haben wir eine versuchte Signal-Slot-Verknüpfung, die keinerlei Effekt haben wird. Der Programmierer wollte hiermit erreichen, dass beim Anklicken eines Buttons (bspw. `QPushButton`) der Text des Labels (`QLabel`) verändert wird. Der Button verwendet dabei das Signal `clicked()` und das Label den Slot `setText()`. Dass dieses Beispiel nicht funktioniert, liegt allerdings nicht an der falschen Signatur, sondern schlicht und einfach daran, dass hier gar keine Verbindung aufgebaut wurde.

Widget

Sie werden im Verlauf des Buches noch häufig mit diesem Begriff konfrontiert, daher hierzu eine kurze Erklärung. Ein Widget ist im Grunde ein Element eines grafischen Fenstersystems (Fenstermanager). Ein solches Widget besteht aus einem Fenster, wo Maus- und-/oder Tastaturereignisse empfangen und verarbeitet werden können. Ein Widget kann aber auch einen nicht sichtbaren Bereich enthalten, worin wiederum andere Elemente (auch Widgets) oder Daten gespeichert werden. Widgets dienen der Interaktion mit dem Anwender und/oder anderen Widgets auf dem Fenstersystem.

Um das folgende Beispiel zu realisieren, leiten wir unsere neue Klasse `MyWindow` von `QWidget` ab. Zunächst die Headerdatei mit der Klasse:

```
00 // kapitel17/signalslot2/mywindow.h
01 #ifndef MYWINDOW_H
02 #define MYWINDOW_H
03 #include <QWidget>
04 #include <QVBoxLayout>
05 #include <QPushButton>
06 #include <QLabel>
```



```

07 #include <QObject>

08 class MyWindow : public QWidget {
09     Q_OBJECT
10 public:
11     MyWindow(QWidget *parent = 0);
12 private:
13     QLabel *label;
14     QPushButton *button0;
15     QPushButton *button1;
16     QVBoxLayout* layout;
17     int cnt; // zählt die Klicks mit
18 private slots:
19     void setText();
20 };
21 #endif

```

Verglichen mit dem Beispiel zuvor, in dem wir eigene Signale und Slots verwendet haben, finden Sie auch hier nichts wirklich Neues, nur dass wir jetzt statt von `QObject` von der Klasse `QWidget` (Zeile 08) ableiten und dass hier nur ein Slot (Zeilen 18 und 19) verwendet wird. Da `QWidget` bereits von `QObject` abgeleitet ist, lässt sich dies auch so realisieren. Die Definition des Konstruktors und der Slot-Ereignisfunktion finden Sie in der folgenden Quelldatei wieder:

```

00 // kapitel17/signalslot2/mywindow.cpp
01 #include "mywindow.h"
02 #include <QApplication>

03 MyWindow::MyWindow(QWidget *parent): QWidget(parent) {
04     label = new QLabel("alter Text");
05     button0 = new QPushButton ("Label aktualisieren");
06     button1 = new QPushButton ("Beenden");
07     layout = new QVBoxLayout(this);
08     layout->addWidget(button0);
09     layout->addWidget(button1);
10     layout->addWidget(label);
11     setLayout(layout);
12     connect( button0, SIGNAL( clicked() ),
13             this, SLOT( setText() ) );
13     connect( button1, SIGNAL( clicked() ),
14             qApp, SLOT( quit() ) );
14     cnt = 0; // Zähler initialisieren
15 }

```

```

16 void MyWindow::setText() {
17     QString msg("Label aktualisiert ");
18     msg.append(QString().setNum(++cnt));
19     msg.append("-mal");
20     label->setText(msg);
21 }

```

Zunächst werden wie gewohnt die einzelnen Widgets, wie Buttons (QPushButton) und Labels (QLabel), erzeugt und der vertikalen Box (QVBoxLayout) hinzugefügt. Anschließend werden die Signal-Slot-Verknüpfungen eingerichtet (Zeilen 12 und 13). Ein besonderes Augenmerk sei hierbei auf den Slot von Zeile 12 gerichtet. Hierbei handelt es sich nicht mehr um den von QLabel zur Verfügung gestellten Slot `setText()`, sondern um unseren selbst definierten Slot ohne Argumente, dessen Implementierung Sie in den Zeilen 16 bis 21 vorfinden.

Klicken Sie jetzt im Beispiel auf den Button mit dem Label LABEL AKTUALISIEREN, wird das Signal `clicked()` ausgelöst. Dieses Signal ist allerdings nicht mit dem Label verbunden, sondern mit dem Fenster bzw. `QWidget`. Daher finden Sie hierbei auch den `this`-Zeiger vor. Erst mit der Slot-Ereignisfunktion `MyWindow::setText()` wird das Text-Label verändert.

qApp

Hierbei dürfte sich wohl gleich die Frage auf tun, was `qApp` in Zeile 13 zu suchen hat? `qApp` ist ein Zeiger auf eine globale Variable in `<QApplication>`. Dieser verweist auf die eindeutige Instanz der Anwendung. Wir verwenden diesen globalen Zeiger, um die Anwendung wieder zu beenden.

Jetzt fehlt nur noch die Hauptfunktion, die jetzt kaum noch Code enthält:

```

00 // kapitel17/signalslot2/main.cpp
01 #include <QApplication>
02 #include "mywindow.h"

03 int main(int argc, char **argv) {
04     QApplication app(argc, argv);
05     MyWindow window;
06     window.show();
07     return app.exec();
08 }

```

Abbildung 17.22 und Abbildung 17.23 zeigen das Programm bei Ausführung.



Abbildung 17.22 Das Programm nach dem Start (hier mit Mac OS X)



Abbildung 17.23 Nach fünfmaligem Drücken von »Label aktualisieren«

Widget mit eigenem Signal

Natürlich will ich Ihnen Ähnliches auch mit einem Signal demonstrieren. Hierzu wollen wir die `QPushButton`-Klasse um das Signal `clicked(int)` erweitern. Ursprünglich gibt es bei dieser Klasse ja nur das Signal `clicked()` ohne ein Argument. Wir wollen ein neues `clicked(int)` mit Argument hinzufügen, welches beim Drücken einer Schaltfläche ein Signal mit einer Identifikationsnummer (ID) der Schaltfläche mitschickt.

Hierzu zunächst die Klasse `MyButton`, welche wir von der Klasse `QPushButton` ableiten:

```
00 // kapitel17/signalslot3/mybutton.h
01 #ifndef MYBUTTON_H
02 #define MYBUTTON_H
03 #include <QString>
04 #include <QWidget>
05 #include <QPushButton>

06 class MyButton : public QPushButton {
07     Q_OBJECT
08     public:
09         MyButton( const QString& text, int id,
10                 QWidget* parent = 0 );
11         MyButton( const QString& text, QWidget* parent = 0);
12     private:
13         static int nextId;
14         int m_id;
15     public slots:
```

```

15     //click() überschreiben
16     void click();
17     signals:
18     //ein neues Signal hinzufügen
19     void clicked(int id);
20 };
21 #endif

```

Zusätzlich zu einem neuen Signal, welches hier in den Zeilen 17 bis 19 deklariert wird, überschreiben wir den Slot `click()` in Zeile 16, welcher in der Klasse `QAbstractButton` definiert ist. Der Slot `click()` führt einen wortwörtlichen Klick durch.

Um den Vorgang besser zu verstehen, benötigen Sie die Implementierung der Klasse `MyButton`:

```

00 // kapitel17/signalslot3/mybutton.cpp
01 #include "mybutton.h"
02 int MyButton::nextId = 0;

03 MyButton::MyButton( const QString& text, int id,
    QWidget* parent ) : QPushButton(text, parent), m_id(id) {
04     connect(this, SIGNAL(clicked()), this, SLOT(click()));
05 }

06 MyButton::MyButton(const QString& text, QWidget* parent )
    : QPushButton(text, parent), m_id(nextId++) {
07     connect(this, SIGNAL(clicked()), this, SLOT(click()));
08 }

09 void MyButton::click() {
10     emit clicked(m_id);
11 }

```

Zunächst fallen die beiden `connect()` (in den Zeilen 04 und 07) in den Konstruktoren auf. Hiermit verbinden wir das alte (oder nennen Sie es das echte) `clicked()`-Signal mit dem eigenen hausgemachten Slot `click()` der Klasse `MyButton`. Es ist maßgebend entscheidend, dass hierbei wirklich die Slot-Elementfunktion von `MyButton` (Zeilen 09 bis 11) ausgeführt wird. Würde die Methode `QPushButton::click()` verwendet, so hätte dies eine Endlos-Rekursion zur Folge, weil `QPushButton::click()` wieder das Signal `clicked()` auslöst. Sobald der Slot `MyButton::click()` ausgelöst wurde, können Sie das neue Signal mit der Identifikationsnummer (ID) verschicken (Zeile 10).

Um aus diesem Beispiel ein Qt-Programm zu machen, wollen wir hierzu auch noch eine Klasse `MyWindow` (abgeleitet von `QWidget`) mit unseren eigenen Schaltflächen

(MyButton) und einem Slot implementieren, der sich um die Ausgabe auf dem Bildschirm kümmert. Hier die Headerdatei unserer Klasse *mywindow.h*:

```
00 // kapitel17/signalslot3/mywindow.h
01 #ifndef MYWINDOW_H
02 #define MYWINDOW_H
03 #include <QWidget>
04 #include <QVBoxLayout>
05 #include <QMessageBox>
06 #include "mybutton.h"

07 class MyWindow : public QWidget {
08     Q_OBJECT
09     public:
10         MyWindow(QWidget* parent = 0);
11     private:
12         MyButton* myButton[3];
13     public slots:
14         // Slot für die Ausgabe
15         void myMessageBox(int id);
16 };
17 #endif
```

Hier finden Sie in Zeile 12 unsere Schaltflächen MyButton mit eigenem implementiertem Signal wieder. In Zeile 13 bis 15 deklarieren wir noch einen eigenen Slot für die Klasse MyWindow. Die Slot-Elementfunktion ist im Grunde eine einfache Nachrichtenbox (QMessageBox), welche die Identifikationsnummer des Signals clicked(int) unserer Buttons zurückgibt. Hierzu die Implementierung der Klasse MyWindow:

```
00 // kapitel17/signalslot3/mywindow.cpp
01 #include "mywindow.h"
02 #include "mybutton.h"
03 #include <QtCore>
04 #include <QtGui>

05 MyWindow::MyWindow(QWidget* parent) : QWidget(parent) {
06     myButton[0] = new MyButton(tr("Drück mich"));
07     myButton[1] = new MyButton(tr("Mich auch"));
08     myButton[2] = new MyButton(tr("Und mich"));
09     QVBoxLayout* layout = new QVBoxLayout(this);
10     for (int i = 0; i < 3; ++i) {
11         layout->addWidget(myButton[i]);
12         connect( myButton[i], SIGNAL( clicked(int) ),
13             this, SLOT( myMessageBox(int) ) );

```

```

13     }
14 }

15 void MyWindow::myMessageBox(int id) {
16     QMessageBox::information(
        this, tr("Ein Button wurde betätigt"),
        QString(tr("Button ID: %1")).arg(id),
        QMessageBox::Ok );
17 }

```

In Zeile 12 wird unser neues clicked(int)-Signal mit dem Ausgabe-Slot myMessageBox verbunden. Sobald also unser neues Signal eintrifft, wird eine Nachrichtenbox mit der ID der Schaltfläche ausgegeben.

Jetzt zum Schluss noch unser Hauptprogramm zur Demonstration:

```

00 // kapitel17/signalslot3/main.cpp
01 #include <QApplication>
02 #include "mywindow.h"
03 #include "mybutton.h"

04 int main(int argc, char *argv[]) {
05     QApplication app(argc, argv);
06     MyWindow window;
07     window.show();
08     return app.exec();
09 }

```

Abbildung 17.24 und Abbildung 17.25 zeigen das Programm bei der Ausführung.

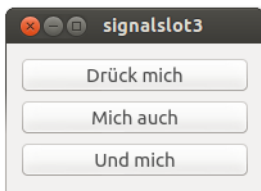


Abbildung 17.24 Das Programm nach dem Start (hier unter Linux)



Abbildung 17.25 Die Schaltfläche »Mich auch« mit der ID 1 wurde betätigt.

QObject::disconnect()

Es ist auch möglich, ein Signal von seiner Verbindung, die mit `connect()` hergestellt wurde, mit `QObject::disconnect()` wieder zu trennen.

17.3 Klassenhierarchie von Qt

Das Qt-Framework umfasst mittlerweile mehrere hundert Klassen. Hierbei die Übersicht zu behalten, ist zunächst für Einsteiger nicht ganz einfach. Qt basiert auf einer Ableitungshierarchie. Um sich also umgehender mit Qt zu befassen, müssen Sie zunächst die Wurzeln und den Aufbau des Frameworks kennen. Wenn Sie erst einmal wissen, wie Qt tickt, werden Sie sich schnell damit zurechtfinden. Dieses theoretische Kapitel geht also kurz auf den Aufbau des Qt-Frameworks ein.

17.3.1 Basisklasse QObject

QObject wurde bereits im Abschnitt zuvor, wo es um das Signal-Slots-Konzept ging, mehrmals erwähnt. Dabei werden Sie sicherlich festgestellt haben, dass es sich hierbei um eine ziemlich wichtige Klasse in Qt handelt. Und in der Tat ist QObject Basis-klasse von den meisten Qt-Klassen. Alle Widgets bspw., die auf dem Bildschirm dargestellt werden, leiten von der Klasse QWidget ab. QWidget wiederum wurde von der Klassen QObject abgeleitet.

Wie Sie bereits im Abschnitt zuvor erfahren haben, fordert auch das Signal-Slot-Konzept, dass die beteiligten Klassen von der Klasse QObject abstammen. Aber auch viele andere nichtgrafische Klassen basieren auf QObject als Basisklasse. Klassen bspw., die der Kommunikation zwischen den Prozessen dienen, werden von QObject abgeleitet, damit sie über Signale und Slots kommunizieren können.

17.3.2 Klassenhierarchie

Natürlich gibt es auch viele weitere Klassen, die nicht von QObject abgeleitet wurden. Dies sind im Grunde alle Klassen, die kein Signal-und-Slot-Konzept und keine automatische Speicherverwaltung benötigen. Bekanntester Vertreter dürfte hier QString sein. QString ist das, was für den C++-Programmierer die Klasse `std::string` aus der Standardbibliothek ist.

Den besten Überblick über die Klassenhierarchie von Qt bekommen Sie hierbei wieder mithilfe des Assistenten von Qt. Sie finden hier die Übersicht im Abschnitt QT REFERENCE DOCUMENTATION unter OVERVIEWS mit INHERITANCE HIERARCHY (siehe Abbildung 17.26). Hierbei werden Sie feststellen, dass QObject den größten Anteil von abgeleiteten Klassen besitzt.

QString oder std::string

Es wird empfohlen, in Qt die Klasse `QString` der Standardklasse `std::string` vorzuziehen. `QString` speichert und verarbeitet den Text im Unicode-Format und erlaubt somit, fast alle Schriftsysteme dieser Erde zu verwenden. Oder kurz, mit `QString` müssen Sie sich nicht um eine Decodierung des Zeichensatzes kümmern.

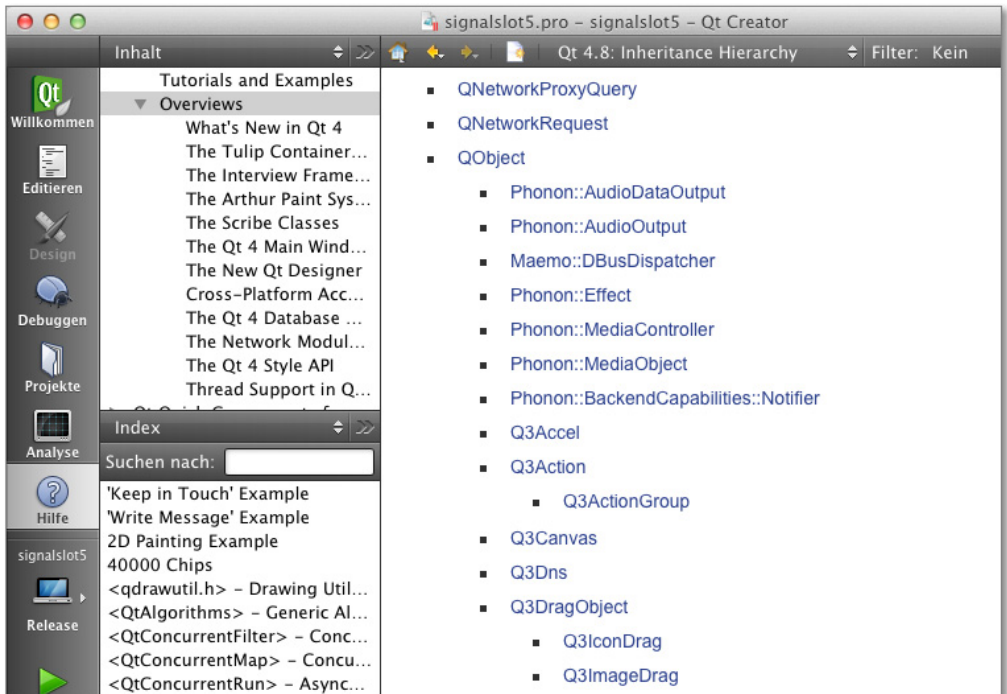


Abbildung 17.26 Übersicht zur Klassenhierarchie von Qt

Trotzdem werden Sie hier auch Klassen wie `QPaintDevice` entdecken, von der auch `QWidget` wiederum einen Teil geerbt hat, womit sie auch bei den sichtbaren GUI-Elementen eine Rolle spielt. Ebenso gilt dies bspw. für die Klasse `QLayout`. Die Klasse wurde ebenfalls von `QObject` abgeleitet, hat aber auch die Superbasisklasse `QLayoutItem`. Die Klasse `QLayout` beinhaltet Widgets, die zur Anordnung der GUI-Elemente auf dem Fenster benötigt werden. Sie haben mit `QVBoxLayout` bereits einen solchen Abkömmling der beiden Klassen kennengelernt.

Abbildung 17.27 können Sie einen kleinen Überblick zum Aufbau der Qt-Klassenhierarchie entnehmen. Hieran kann man schön erkennen, dass viele Klassen zum Teil von mehreren Klassen abgeleitet wurden. Andere Klassen wiederum, wie bspw. `QString`, haben keine Basisklasse.

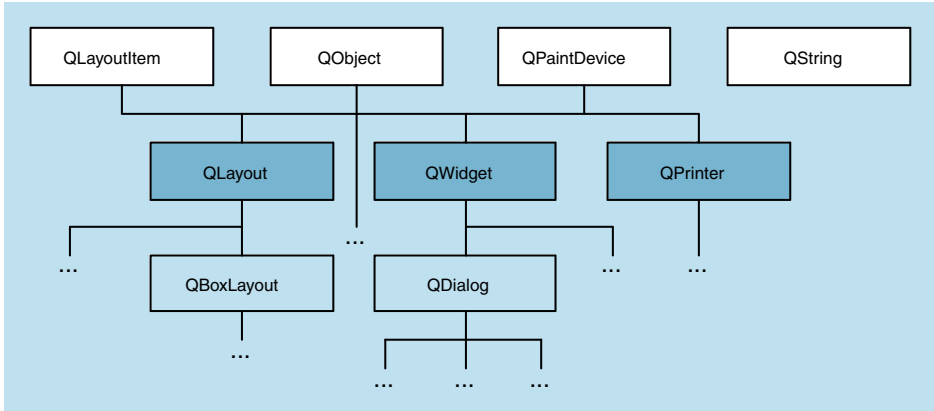


Abbildung 17.27 Schematischer Überblick der Klassenhierarchie von Qt

Gerade für Einsteiger sind solch umfangreiche Frameworks zunächst ziemlich erdrückend, und es fällt schwer, irgendwo anzufangen. Aber auch hier geht man am einfachsten mit dem Qt Assistant die Klassenhierarchie nach oben (oder nach unten).

Um z.B. auf die Schaltfläche `QPushButton` zurückzukommen, finden Sie ganz am Anfang bei seiner Klassenreferenz im Assistant eine Zeile wie `INHERITS: QABSTRACTBUTTON`, also abgeleitet von `QAbstractButton`. Klicken Sie auf den Verweis von `QAbstractButton`. Bei dieser Klassenreferenz erfahren Sie auch gleich, welche weiteren Widgets aus `QAbstractButton` erstellt wurden. Dies steht dann unter `INHERITED BY`, und dies sind bei `QAbstractButton` die Klassen `Q3Button`, `QCheckBox`, `QPushButton`, `QRadioButton` und `QToolButton`. Das Eltern-Widget von `QAbstractButton` hingegen (*Inherits*) ist `QWidget`. Hier (bei `QWidget`) finden sie auch gleich endlos viele GUI-Elemente, die davon abgeleitet wurden. Und `QWidget` wiederum wurde abgeleitet von `QObject` und `QPaintDevice` (siehe Abbildung 17.28).

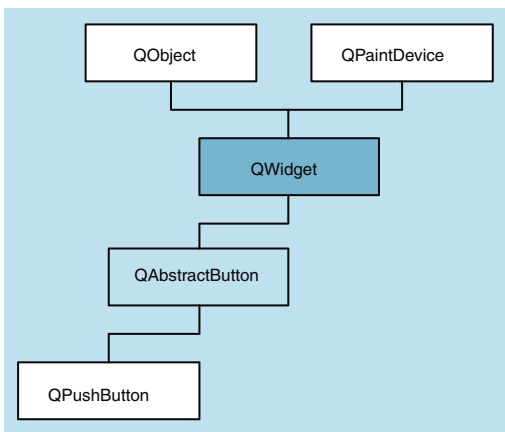


Abbildung 17.28 Zur Demonstration die Klassenhierarchie von »QPushButton«

Dieses Durchlaufen der Klassen mit dem Assistant von Qt ist unerlässlich, wenn es darum geht, nach Funktionen, Methoden, Eigenschaften, Slots oder Signalen bestimmter Klassen zu suchen oder eben eigene Widgets zu implementieren.

17.3.3 Speicherverwaltung

Da C++ keine automatische Speicherverwaltung, wie bspw. Java, besitzt, haben wir in den Beispielen vom Kapitel zuvor das Öffnen des `new`-Operators zum Erzeugen neuer Objekte verwendet.

In Abschnitt 17.3.2 haben Sie die Klassenhierarchie von Qt kennengelernt. Sie haben gesehen (Abbildung 17.28), wie aus von `QObject` abgeleiteten Klassen eine Baumhierarchie gebildet wird. Dadurch entsteht eine Art von Eltern-Kind-Beziehung – genauer eine *Komponenten-Hierarchie* (oder auch Composite-Pattern mit Komponenten). Dank einer solchen Komponenten-Hierarchie werden, wenn bspw. das Hauptfenster zerstört wird, auch alle Kinder und Kindeskiner zerstört.

Damit die Qt-Klassen verwendet werden können, wird bei der Initialisierung immer die Übergabe eines Zeigers auf das aufrufende Elternobjekt verlangt. Dadurch ist gewährleistet, dass die Elternklasse Kenntnis von ihren Kindobjekten hat und diese bei der eigenen Löschung auch aus dem Speicher mit `delete` zerstört werden. Aus diesem Grund muss jedes Qt-Objekt im Heap erzeugt werden, was der Operator `new` ja macht.

Also nochmals etwas anders: Durch den Aufruf des geerbten Konstruktors mit `parent` als Argument, wird `this` (das Kind/Widget) in die Liste beim Vater/umgebenden Widget eingetragen. So kann dieser in seinem eigenen Destruktor `delete` für alle Kinder/Widgets aufrufen.

Wenn Sie sich den folgenden Code ansehen, werden Sie feststellen, dass am Ende immer ein Hauptwidget die Elternschaft übernimmt. Wird das Hauptwidget zerstört, versucht es, auch den Speicher der Kinderwidgets wieder freizugeben. Wenn hier keine Zeiger verwendet wurden, wird versucht, bereits freigegebene Speicher zu zerstören, was Sie natürlich nicht wollen. Hierzu bspw. folgende Zeilen:

```
...
01  QWidget win;
02  QVBoxLayout* layout = new QVBoxLayout(&win);
03  QPushButton* but01 = new QPushButton("Label");
04  // Widgets der vertikalen Box hinzufügen
05  layout->addWidget(but01);
06  win.show();
...
```

Hierbei wird `QVBoxLayout` in der Zeile 02 als Kind von `QWidget` erzeugt. Alternativ könnten Sie die Elternschaft des Layouts auch ohne die Adressübergabe von `win` in

der Zeile 02 und erst am Ende vor der Zeile 06 mit `win.setLayout(layout)` realisieren. Die Schaltfläche in der Zeile 03 hingegen ist bei der Erzeugung noch elternlos und würde jetzt noch nicht angezeigt werden. Erst beim Aufruf von `addWidget()` in der Zeile 05 wird dafür gesorgt, dass jemand die Elternschaft für die Schaltfläche übernimmt. Allerdings entsteht hierbei auch häufig ein wenig Verwirrung, denn nicht die Klasse `QVBoxLayout` übernimmt die Elternschaft von `QPushButton`, sondern `QWidget`. Betrachtet man hierbei die Widget-Hierarchie, so wird man feststellen, dass `QPushButton` und `QVBoxLayout` in der Hierarchie etwa gleichgestellt sind. Deswegen müssen die Kindelemente eines Widgets immer GUI-Elemente eines übergeordneten Widgets sein, damit die Speicherverwaltung funktioniert. Das Hauptwidget `win` hat somit die Elternschaft über die Widgets in den Layouts übernommen und somit auch die Speicherverwaltung. Wird das Hauptwidget irgendwann hinter der Zeile 06 zerstört (bspw. weil das Fenster geschlossen wurde), wird der Destruktor aufgerufen und gibt den Speicher der Kinderwidgets frei. Durch den Codeausschnitt ergibt sich die in Abbildung 17.29 abgedruckte Hierarchie.

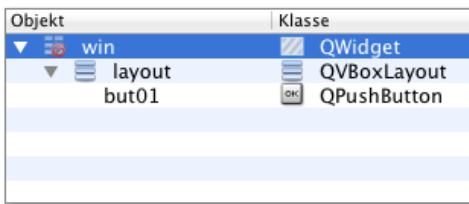


Abbildung 17.29 Eltern-Kind-Elemente

Allerdings ist es auch möglich, das Eltern-Widget immer gleich bei der Erzeugung mit anzugeben. Bezogen auf den eben gezeigten Codeausschnitt würde dies folgendermaßen aussehen:

```
...
01  QWidget win;;
02  // Eltern-Widget als zweites Argument angeben
03  QPushButton* but01 = new QPushButton( "Label", &win );
04  win.show();
...
```

17.4 Eigene Widgets erstellen

In Abschnitt 17.2, »Signale und Slots«, wurde bereits gezeigt, wie man dies über die Ableitung einer Klasse erreichen kann. Der demonstrierte Vorgang ist auch ein üblicher Weg bei der Qt-Programmierung. Man leitet dabei gewöhnlich von einer etwas höher in der Hierarchie angesiedelten Klasse (bspw. meistens `QWidget` oder `QDialog`) ab und platziert darin die Widgets, welche für das Teilobjekt benötigt werden. In der

Praxis werden die Anwendungen aus mehreren solchen Teilobjekten dann zusammengesetzt.

Im Grunde erstellt man meistens sein eigenes Widget (wobei hier erweitertes statt eigenes eher zutrifft) in folgenden Schritten:

- ▶ Widget von der gewünschten Klasse ableiten
- ▶ GUI-Elemente erzeugen, die im Widget angezeigt werden sollen
- ▶ GUI-Elemente anordnen, die angezeigt werden (Layout)
- ▶ Signal- und Slot-Verbindungen einrichten

Hierzu soll ein einfaches Beispiel erstellt werden. Wir wollen eine einfache Nachrichtenbox mit einer Schaltfläche erstellen. Anstatt im Hauptprogramm ein Text-Label, eine Schaltfläche und ein Widget für das Layout zu erzeugen, erstellen wir hier ein eigenes Widget, welches wir jederzeit ohne großen Aufwand im Hauptprogramm erzeugen und anzeigen lassen können.

QMessageBox

Qt bietet mit `QMessageBox` eine einfachere und komfortablere Alternative an, Nachrichtenboxen zu verwenden.

Als Erstes erstellen wir das Grundgerüst der Klasse. Hierzu wird alles als Grundgerüst in einer Headerdatei zusammengefasst, was für eine Nachrichtenbox benötigt wird. In diesem Beispiel wurde folgendes Gerüst erstellt:

```
00 // kapitel17/mywidget/mywidget.h
01 #ifndef MYWIDGET_H
02 #define MYWIDGET_H
03 #include <QWidget>
04 #include <QPushButton>
05 #include <QVBoxLayout>
06 #include <QLabel>
07 #include <QString>

08 class MyWidget : public QWidget {
09 public:
10     MyWidget(QString qstr = "Bitte Button betätigen",
11              QString but = "Ende", QWidget *parent = 0 );
12 private:
13     QPushButton *button0;
14     QLabel*      label;
15     QVBoxLayout* layout;
16 };
17 #endif
```

In diesem Beispiel werden keine Methoden oder eigene Signale bzw. Slots implementiert. Der erste Parameter im Konstruktor (Zeile 10) wird für das Text-Label (QLabel) verwendet. Gibt der Anwender nichts anderes an, steht hier der vorgegebene Text in der Nachrichtenbox (hier »Bitte Button betätigen«). Gleiches gilt für das zweite Argument des Konstruktors in Bezug auf die Schaltfläche (QPushButton). Als zusätzliche Widgets in der Klasse MyWidget wurden hier noch QPushButton, QLabel und QVBoxLayout implementiert (Zeilen 12 bis 14). Jetzt die Definition der Klasse MyWidget:

```
00 // kapitel17/mywidget/mywidget.cpp
01 #include "mywidget.h"
02 #include <QApplication>

03 // Neue Widget-Klasse vom eigentlichen Widget ableiten
04 MyWidget::MyWidget( QString lab, QString but,
                    QWidget *parent): QWidget(parent) {
05     // Elemente des Widgets erzeugen
06     button0 = new QPushButton (but);
07     layout  = new QVBoxLayout(this);
08     label   = new QLabel(lab);
09     // Elemente des Widgets anordnen/anpassen
10     layout->addWidget(label);
11     layout->addWidget(button0);
12     // Signale des Widgets einrichten
13     connect( button0, SIGNAL( clicked() ),
              qApp, SLOT( quit() ) );
14 }
```

Zunächst werden die GUI-Elemente des neuen Widgets erzeugt (Zeilen 06 bis 08), welche anschließend entsprechend einem Layout angeordnet werden (Zeilen 10 bis 11). Am Ende richten wir noch eine Signal-Slot-Verbindung ein (Zeile 13).

So oder so ähnlich sind im Grunde alle Qt-Anwendungen aufgebaut. Jetzt fehlt eigentlich nur noch eine Hauptfunktion, welche die Klasse bei der Ausführung demonstriert:

```
00 // kapitel17/mywidget/main.cpp
01 #include <QApplication>
02 #include "mywidget.h"
03 int main(int argc, char *argv[]) {
04     QApplication app(argc, argv);
05     MyWidget window("Programm mit Ende beenden");
06     window.show();
07     return app.exec();
08 }
```

Abbildung 17.30 zeigt das Programm bei der Ausführung.

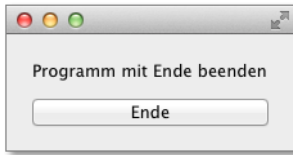


Abbildung 17.30 Unser selbst erstelltes Widget bei der Ausführung

17.5 Eigene Widgets mit dem Qt Designer erstellen

Nicht immer hat man Zeit oder Lust, eigene Widgets manuell schreiben. Für solche Zwecke wurde der Qt Designer entwickelt. Mit diesem klicken Sie sich quasi Ihre Anwendung zusammen. Wie dies aussehen kann, soll die folgende Schritt-für-Schritt-Anleitung zeigen, in der wir das Beispiel aus Abschnitt 17.4, »Eigene Widgets erstellen«, mit dem RAD-Tool (*Rapid Application Development*) Qt Designer erstellen.

Im Beispiel wird davon ausgegangen, dass Sie bereits ein »leeres Qt-Projekt« erstellt haben, wie in Abschnitt 17.1.2, »Ein einfaches Projekt erstellen«, bis einschließlich Arbeitsschritt 5 geschehen, wo Sie ein leeres Projekt nur mit der Projektdatei *.pro vor sich finden.

1. Klicken Sie mit der rechten Maustaste auf den Projektordner, und wählen Sie im Kontextmenü den Befehl HINZUFÜGEN aus (Abbildung 17.31).

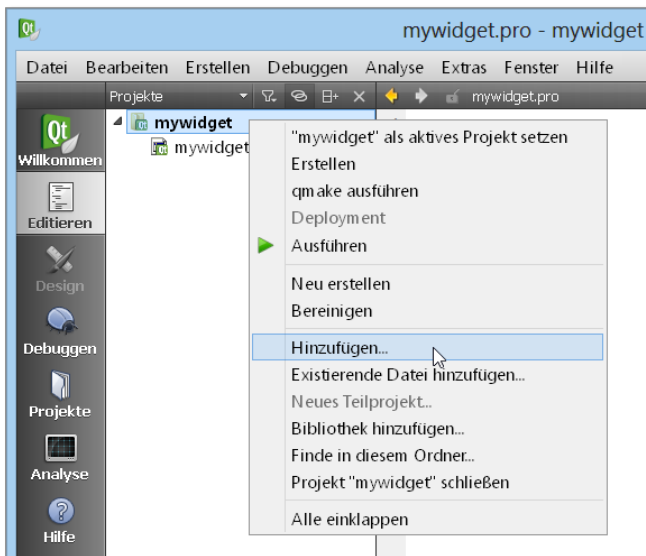


Abbildung 17.31 Etwas dem »noch« leeren Projekt hinzufügen

2. Wählen Sie in der folgenden Dialogbox unter **VORLAGE QT** aus. Daneben wählen Sie dann **QT-DESIGNER-FORMULARKLASSE** aus und klicken auf die Schaltfläche **AUSWÄHLEN** (Abbildung 17.32).

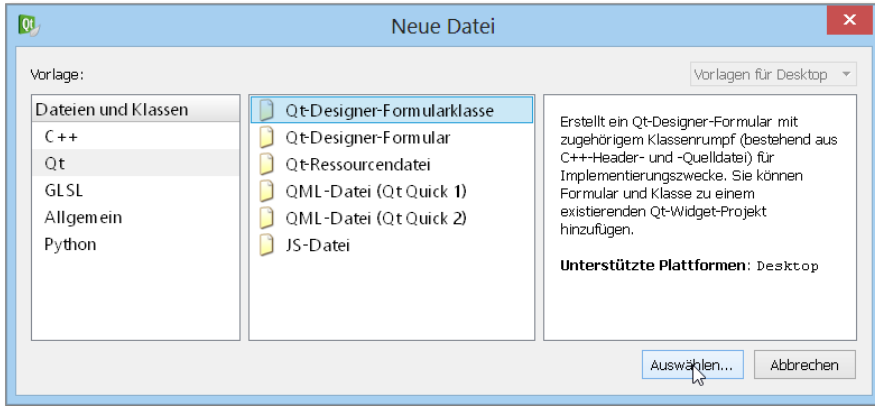


Abbildung 17.32 Neue Qt-Designer-Formularklasse erstellen

3. Wählen Sie im nächsten Schritt bei **TEMPLATES\FORMS** die Art der Formularvorlage aus. Im Beispiel reicht uns **WIDGET** aus. Klicken Sie auf die Schaltfläche **WEITER** (Abbildung 17.33).

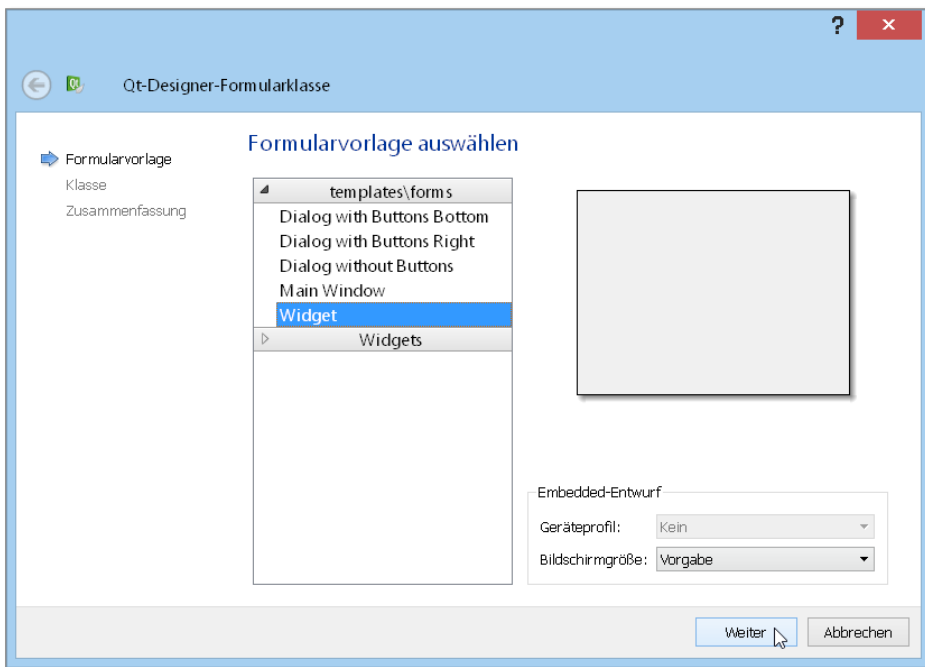


Abbildung 17.33 Formularvorlage auswählen

4. Im nächsten Dialog können Sie den Namen für die Klasse und für HEADER-, QUELLDATEI und FORM-DATEI angeben sowie den PFAD, unter dem die Klasse gespeichert werden soll (Abbildung 17.34). Klicken Sie auf WEITER, wenn Sie diese Angaben gemacht haben.

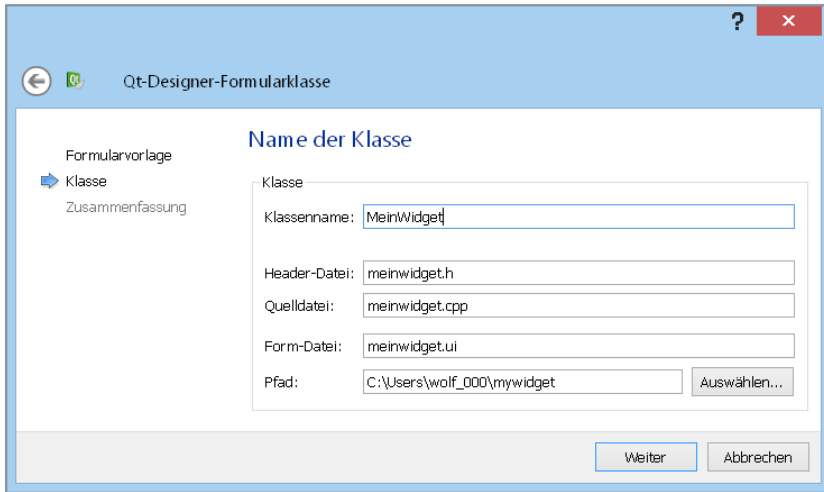


Abbildung 17.34 Angaben zu Klassenname, Header-, Quell- und Form-Datei und dem Pfad können hier gemacht werden.

5. Im letzten Schritt müssen Sie noch angeben, welchem Projekt Sie diese Qt-Designer-Formularklasse hinzufügen wollen, und können gegebenenfalls noch eine Versionskontrolle verwenden. Schließen Sie diesen Schritt mit der Schaltfläche ABSCHLIESSEN ab (Abbildung 17.35).

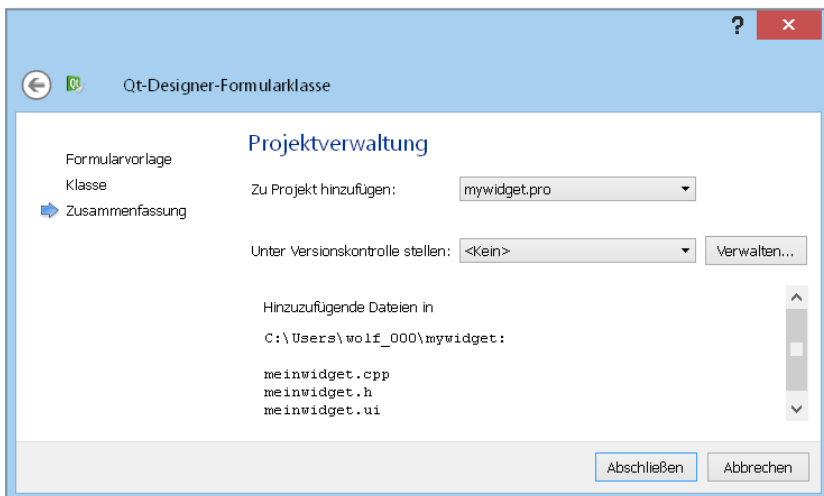


Abbildung 17.35 Formularklasse dem Projekt hinzufügen

UI-Datei

Die Qt-Form-Datei hat die Endung *.ui im Projekt. Doppelklicken Sie diese UI-Datei, schaltet Qt Creator in den Modus DESIGN um.

Wenn alles geklappt hat, dann sollten Sie Qt Creator jetzt im Modus DESIGN **1** vorfinden (Abbildung 17.36). Der Design-Modus ist ordentlich eingeteilt. Links finden Sie die einzelnen Qt-Widgets **2**, die Sie jederzeit einzeln per Drag & Drop rechts auf die graue Fläche **3** ziehen können. Die graue Fläche stellt Ihr selbst erstelltes Widget dar, das hier noch leer ist. Unterhalb des Layouts **4** finden Sie den Aktionseditor und – noch wichtiger – die Möglichkeit, Signale und Slots miteinander zu verbinden. Rechts oben sehen Sie einen Überblick zur aktuellen Klassenhierarchie Ihres Widgets **5**, hier im Augenblick nur unsere Klasse MeinWidget, welche von QWidget abgeleitet wurde. Rechts unten finden Sie die Eigenschaften der Widgets **6**, wo Sie auch die Werte anpassen können.

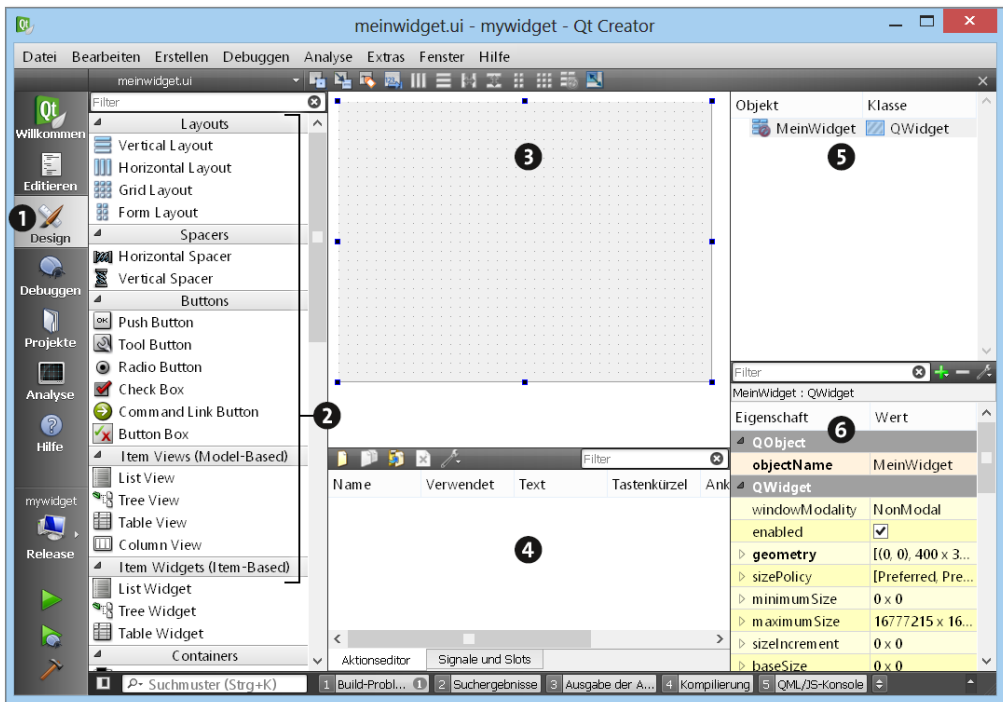


Abbildung 17.36 Qt Creator im Design-Modus

Nach einer kurzen Übersicht soll jetzt weiter an unserem eigenen Widget gearbeitet werden – ohne allerdings jetzt zu sehr auf das Layout und dessen Schönheit zu achten. Auf das Layout wird in Abschnitt 17.6, »Widgets anordnen«, noch etwas umfas-

sender eingegangen. Hier geht es zunächst einmal nur darum, sich ein wenig mit dem Design-Modus von Qt Creator vertraut zu machen.

1. Ziehen Sie das Layout VERTICAL LAYOUT ❶ (QVBoxLayout) mit gedrückter Maustaste auf die graue Fläche ❷, und lassen Sie es dort fallen, so dass Sie einen roten Rahmen sehen können (Abbildung 17.37). Die Größe des Rahmens können Sie nachträglich über die vier Seiten- und Eckpunkte anpassen. Das Gleiche können Sie auch mit der Größe des Widgets selbst über die Seiten- und Eckpunkte machen.

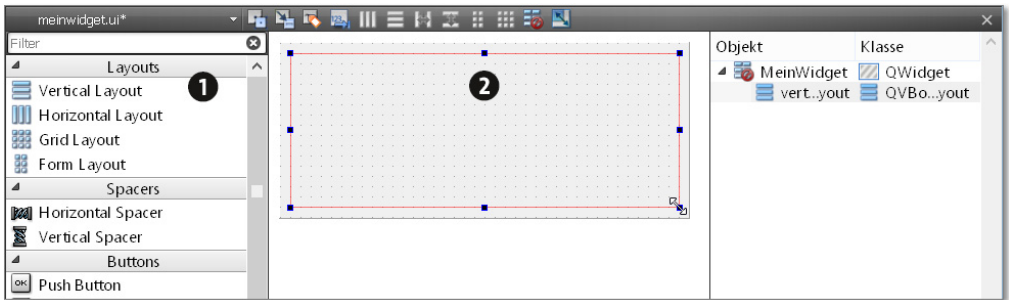


Abbildung 17.37 Eine vertikale Box (QVBoxLayout) hinzugefügt

2. In diese vertikale Box können Sie jetzt per Drag & Drop weitere Qt-Widgets hineinziehen, und diese werden jetzt immer vertikal angeordnet. Im Beispiel wurden hier noch ein Label und ein Push-Button in die vertikal Box gezogen (Abbildung 17.38). Rechts oben ❸ lässt sich dann auch sehr schön die Hierarchie der Widgets betrachten.

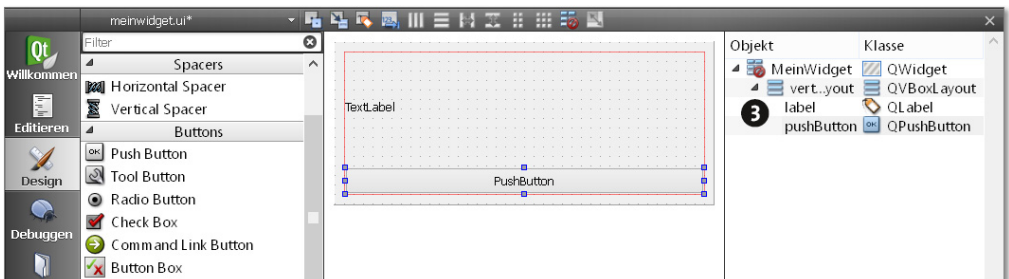


Abbildung 17.38 Nach dem Hinzufügen eines Text-Labels und einer Schaltfläche zum Widget

3. Im nächsten Schritt wurden der Text des Labels und der Schaltfläche angepasst. Dieser lässt sich ganz einfach per Doppelklick auf den Text editieren. Die Ausrichtung des Labels wurde außerdem noch bei den Eigenschaften ❹ unter ALIGNMENT auf HORIZONTAL ZENTRIEREN gesetzt. Dazu müssen Sie auch das Widget ausgewählt haben (Abbildung 17.39).

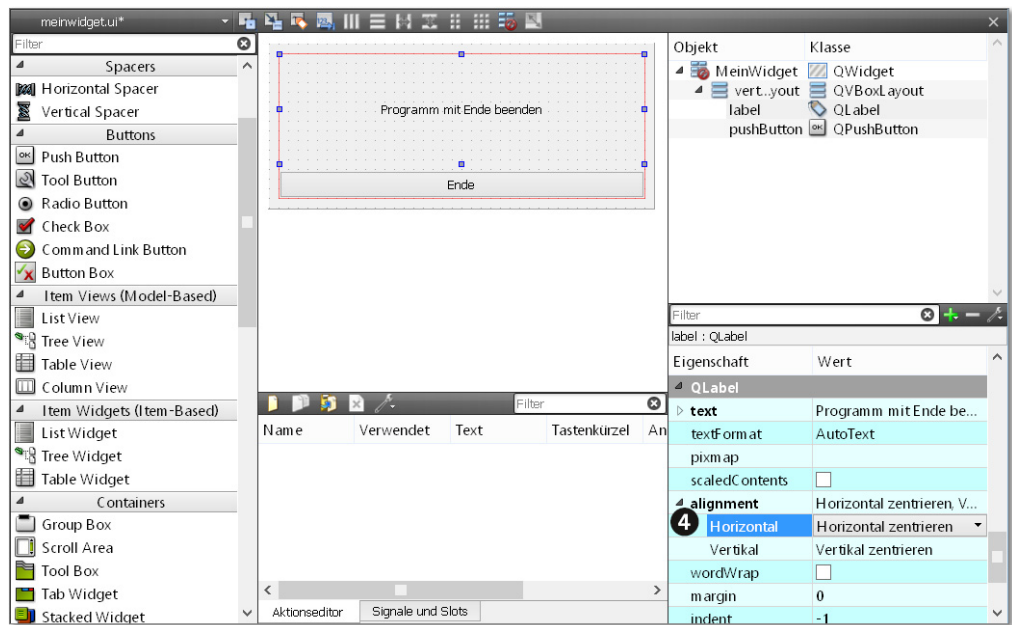


Abbildung 17.39 Der Text für Label und Schaltfläche wurde editiert.

4. Wählen Sie jetzt **SIGNALE UND SLOTS** ❹ aus. Klicken Sie auf das grüne Plus-Symbol ❺, um eine neue Signal-Slot-Verbindung einzurichten (Abbildung 17.40). Wählen Sie jetzt als **SENDER** den Bezeichner der Schaltfläche (hier `pushButton`), als **SIGNAL** `clicked()`, der **EMPFÄNGER** ist unser Widget (hier `MeinWidget`), und der **SLOT** lautet hier `close()`.

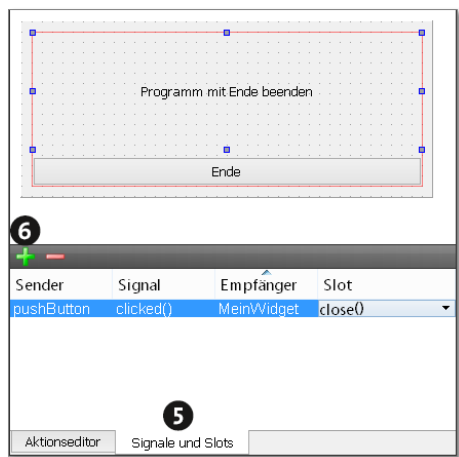


Abbildung 17.40 Signal-Slot-Verbindung einrichten

5. Wenn Sie Ihr Widget fertiggestellt haben, brauchen Sie noch eine `main()`-Funktion, welche dieses Widget auch erzeugt und verwendet. Wechseln Sie daher in den Modus EDITIEREN **7**, und fügen Sie im Beispiel noch eine C++-Quelldatei `main.cpp` dem Projekt hinzu, in der Sie das Widget erzeugen und anzeigen lassen (Abbildung 17.41).

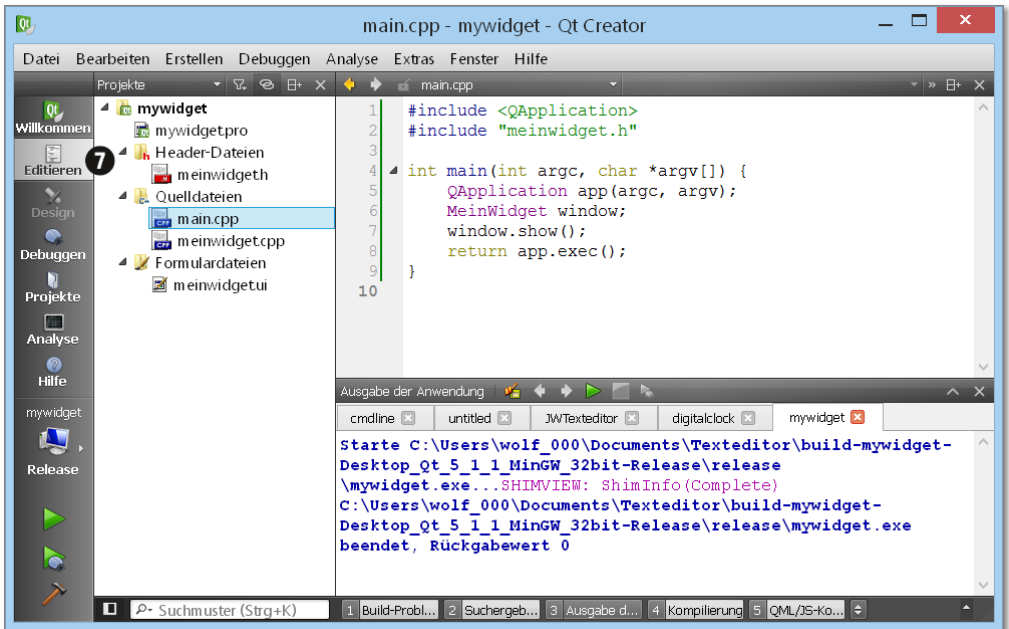


Abbildung 17.41 Eine `main()`-Funktion wird noch benötigt, welche unser Widget (hier »MeinWidget«) erzeugt und anzeigt.

Im Beispiel wurde folgender C++-Quellcode dem Projekt hinzugefügt:

```

01 #include <QApplication>
02 #include "meinwidget.h"

03 int main(int argc, char *argv[]) {
04     QApplication app(argc, argv);
05     MeinWidget window;
06     window.show();
07     return app.exec();
08 }

```

Fehlende Bibliotheken in der .pro-Datei (No such file or directory)

Auch hier gilt: Beim Erstellen eines leeren Qt-Projekts kann es passieren, dass `qmake` die nötigen Qt-Module nicht hinzulinkt. In unserem Beispiel würden hierzu die Module *Qt Core*, *Qt GUI* und *Qt Widgets* benötigt.

Sollten Sie Probleme mit dem Übersetzen haben, brauchen Sie einfach nur in der *.pro*-Datei folgende Zeile einzufügen:

```
QT += core gui widgets
```

6. Unser Beispiel ist nun im Grunde fertig. Sie brauchen das Projekt jetzt nur noch zu speichern, zu übersetzen und auszuführen (Abbildung 17.42). Die UI-Datei können Sie nach wie vor weiter im Modus *DESIGN* bearbeiten, um Änderungen daran vorzunehmen.

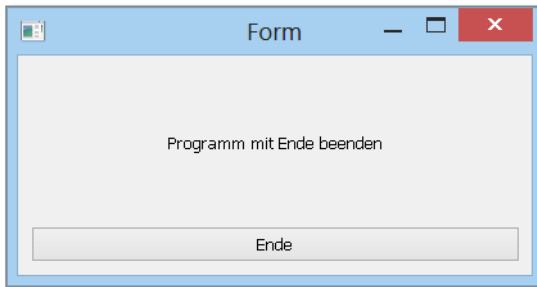


Abbildung 17.42 Fertig ist unser eigenes mit dem Qt Designer erstelltes Widget.

Weitere Widgets

Auf diese Art und Weise können Sie Ihrem Projekt viele weitere UI-Dateien bei Bedarf hinzufügen und sie verwenden.

Dialoge und Hauptfenster mit dem Qt Designer entwerfen

Wie Sie eigene Widgets mit dem Qt Designer erstellen können, war hier erst der Anfang. In Abschnitt 17.9, »Eigenen Dialog mit dem Qt Designer erstellen«, und in Abschnitt 17.13, »Anwendungen in einem Hauptfenster mit dem Qt Designer«, erfahren Sie, wie Sie eigene Dialoge und Hauptfenster-Anwendungen mit dem Qt Designer entwerfen und verwenden können.

17.6 Widgets anordnen

Ein weiteres sehr wichtiges Thema ist das Layout der Anwendungen. Mit dem Layout ist hierbei die Anordnung der Widgets gemeint, bspw. ob die einzelnen Widgets horizontal oder vertikal angeordnet werden sollen oder was passiert, wenn der Anwender die Größe des Fensters verändert usw.

Der eine oder andere wird mir jetzt hierbei entgegenen, das Thema Layout ist doch Sache des Designers von Qt (ein RAD-Tool). Im Grunde ist das auch richtig. Es beansprucht wesentlich mehr Zeit, das Layout Zeile für Zeile selbst zu schreiben. Hierfür kann man bei einem vernünftigen Framework auch immer ein RAD-Tool verwenden. Allerdings ist es unerlässlich für den Einstieg in Qt, zu wissen, was für grundlegende Layout-Widgets es gibt.

17.6.1 Grundlegende Widgets für das Layout

Bevor wir das grundlegende Qt-Layout-Konzept näher erläutern, wollen wir uns zunächst die Klassenhierarchie davon ansehen (Abbildung 17.43).

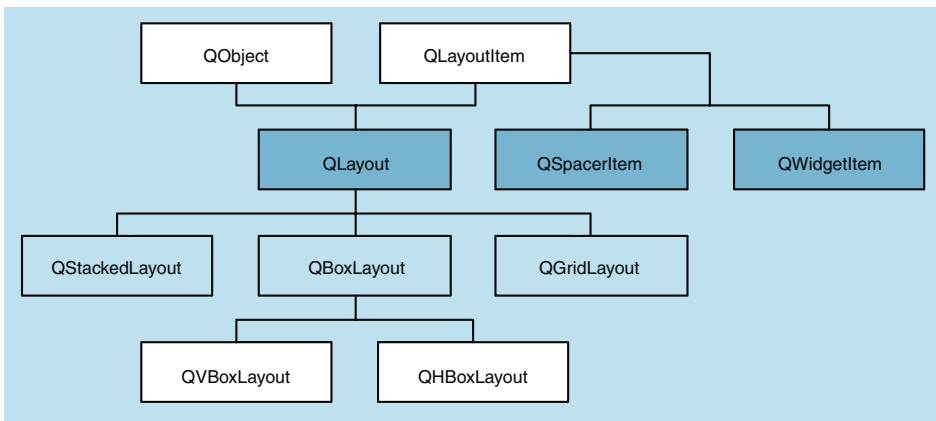


Abbildung 17.43 Klassenhierarchie für das Layout von Qt-Widgets

In der Praxis werden hierbei die Klassen `QGridLayout`, `QStackedLayout`, `QVBoxLayout` und `QHBoxLayout` am meisten und auch direkt verwendet. Alle Klassen sind hierbei von der Basisklasse `QLayout`, und `QLayout` wiederum wurde von der Superbasisklasse `QObject` und `QLayoutItem` abgeleitet. Von `QLayoutItem` allein wurden die beiden Klassen `QSpacerItem` und `QWidgetItem` abgeleitet, welche allerdings beide gewöhnlich nicht direkt verwendet werden. `QBoxLayout` kann wiederum direkt verwendet werden, was aber auch eher seltener der Fall ist, weil die beiden Klassen `QVBoxLayout` und `QHBoxLayout` die meisten Fälle abdecken.

QGridLayout, QVBoxLayout und QHBoxLayout

Der einfachste und schnellste Weg, Ihrer Anwendung ein gutes Layout zu verpassen, dürften die eingebauten Widgets `QHBoxLayout`, `QVBoxLayout` und `QGridLayout` sein (auch Layout-Manager genannt). Alle drei Klassen sind von `QLayout` (siehe Abbildung 17.43) abgeleitet. Die einzelnen Layouts lassen sich recht schnell beschreiben:

- ▶ QVBoxLayout – ordnet die Widgets in einer vertikalen Spalte von oben nach unten an.
- ▶ QHBoxLayout – ordnet die Widgets in einer horizontalen Reihe von links nach rechts an (in Ländern, wo die Schrift von rechts nach links erfolgt, ist auch die Anordnung dann entsprechend).
- ▶ QGridLayout – ordnet die Widgets in einem zweidimensionalen Raster an. Man kann sich dies ähnlich wie bei einer Tabellenkalkulation vorstellen. Ein Widget kommt bspw. in Zeile 1 und Reihe 3. Die Zählung fängt hier bei 0 an.

In Abbildung 17.44 können Sie alle grundlegenden Layouts von Qt näher betrachten.

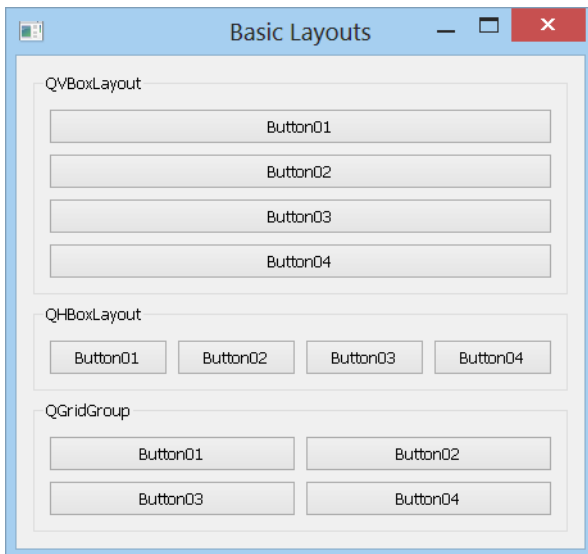


Abbildung 17.44 QVBoxLayout, QHBoxLayout und QGridGroup

Natürlich soll hier nicht nur eine Abbildung gezeigt, sondern auch das entsprechende Beispiel dazu geschrieben werden. Der Einfachheit halber werden hierbei nur Schaltflächen (QPushButton) als Widgets zur Anordnung verwendet. Zunächst wieder das Grundgerüst in einer Headerdatei:

```
00 // kapitel17/layout01/mylayout.h
01 #ifndef MYLAYOUT_H
02 #define MYLAYOUT_H
03 #include <QWidget>
04 #include <QPushButton>
05 #include <QVBoxLayout>
06 #include <QHBoxLayout>
07 #include <QGridLayout>
08 #include <QGroupBox>
```

```

09 class MyLayoutEx : public QWidget {
10 public:
11     MyLayoutEx();
12 private:
13     QPushButton *but01[3], *but02[3], *but03[3], *but04[3];
14     QVBoxLayout* AllBox;
15     QVBoxLayout* VBox;
16     QHBoxLayout* HBox;
17     QGridLayout* Grid;
18     QGroupBox* VGroup;
19     QGroupBox* HGroup;
20     QGroupBox* GridGroup;
21 };
22 #endif

```

Neben den Buttons finden Sie hier die drei Layout-Widgets `QVBoxLayout`, `QHBoxLayout` und `QGridLayout`. Dass hier zweimal Klassen vom Typ `QVBoxLayout` vorhanden sind, hat damit zu tun, dass die einzelnen Layout-Widgets auch wieder im Fenster angeordnet werden müssen. Wie Sie in Abbildung 17.44 sehen können, haben wir die Layout-Widgets vertikal angeordnet.

Bevor wir die einzelnen Layout-Widgets allerdings in die vertikale Box einfügen, stecken wir die einzelnen Layouts mit den Buttons in eine Gruppenbox (`QGroupBox`). Im Grunde ist dies ein Widget, welches ebenfalls Widgets beinhalten kann und zusätzlich einen Rahmen um diese Widgets mit einem Text-Label besitzt.

Auch hier hört sich das Ganze wieder komplizierter an, als es ist. Am besten sehen Sie sich dazu den entsprechenden Konstruktor und die anschließende Erläuterung an:

```

00 // kapitel17/layout01/mylayout.cpp
01 #include "mylayout.h"
02 #include <QApplication>

03 // Neue Widget-Klasse vom eigentlichen Widget ableiten
04 MyLayoutEx::MyLayoutEx() {
05     // 3 mal 4 Buttons erzeugen
06     for( int j = 0; j < 3; ++j ) {
07         but01[j] = new QPushButton("Button01");
08         but02[j] = new QPushButton("Button02");
09         but03[j] = new QPushButton("Button03");
10         but04[j] = new QPushButton("Button04");
11     }
12     // Buttons vertikal anordnen
13     VBox = new QVBoxLayout;

```



```

14     VBox->addWidget(but01[0]);
15     VBox->addWidget(but02[0]);
16     VBox->addWidget(but03[0]);
17     VBox->addWidget(but04[0]);
18     // eine Box mit Label um die Buttons
19     VGroup = new QGroupBox("QVBoxLayout");
20     // die vertikale Box mit den Buttons
21     // in die Group-Box stecken
22     VGroup->setLayout(VBox);
23     // Buttons horizontal anordnen
24     HBox = new QHBoxLayout;
25     HBox->addWidget(but01[1]);
26     HBox->addWidget(but02[1]);
27     HBox->addWidget(but03[1]);
28     HBox->addWidget(but04[1]);
29     // eine Box mit Label um die Buttons
30     HGroup = new QGroupBox("QHBoxLayout");
31     // die horizont. Box mit den Buttons
32     // in die Group-Box stecken
33     HGroup->setLayout(HBox);
34     // Buttons auf einem Raster anordnen
35     Grid = new QGridLayout;
36     Grid->addWidget(but01[2], 0, 0);
37     Grid->addWidget(but02[2], 0, 1);
38     Grid->addWidget(but03[2], 1, 0);
39     Grid->addWidget(but04[2], 1, 1);
40     // eine Box mit Label um die Buttons
41     GridGroup = new QGroupBox("QGridGroup");
42     // die Raster-Box mit den Buttons
43     // in die Group-Box stecken
44     GridGroup->setLayout(Grid);
45     // eine vertikale Box erzeugen
46     AllBox = new QVBoxLayout(this);
47     // alle Group-Boxen in die vertik. Box stecken
48     AllBox->addWidget(VGroup);
49     AllBox->addWidget(HGroup);
50     AllBox->addWidget(GridGroup);
51     // Layout setzen
52     setLayout(AllBox);
53     // Fenstertitel angeben
54     setWindowTitle("Basic Layouts");
55 }

```

In den Zeilen **06** bis **11** erzeugen wir zunächst jeweils 3 mal 4 Schaltflächen der Klasse `QPushButton`. Anschließend erzeugen wir in Zeile **13** eine vertikale Layout-Box (`QVBoxLayout`), worin in den Zeilen **14** bis **17** jeweils vier Buttons mit der Methode `addWidget()` hinzugefügt werden. Jetzt wird in Zeile **19** eine Art Gruppenbehälter mit Rahmen für Widgets (`QGroupBox`) mit dem Text-Label "QVBoxLayout" erzeugt, worin dann in Zeile **22** die vertikale Box (`QVBoxLayout`) mitsamt den darin enthaltenden Schaltflächen platziert wird.

Der gleiche Vorgang erfolgt jetzt auch in den Zeilen **24** bis **33**, nur eben bezogen auf eine horizontale Box (`QHBoxLayout`). Zum Schluss wird Ähnliches auch noch mit einem Raster-Layout (`QGridLayout`) gemacht. Allerdings gestaltet sich hierbei das Hinzufügen der einzelnen Schaltflächen mit `addWidget()` etwas anders. Hierzu nochmals der Codeausschnitt der Zeilen **36** bis **39**, nur jetzt mit Kommentaren versehen:

```
Grid->addWidget(but01[2], 0, 0); // Zeile 0; Spalte 0
Grid->addWidget(but02[2], 0, 1); // Zeile 0; Spalte 1
Grid->addWidget(but03[2], 1, 0); // Zeile 1; Spalte 0
Grid->addWidget(but04[2], 1, 1); // Zeile 1; Spalte 1
```

Natürlich ist `QGridLayout` wesentlich vielseitiger, als es hier den Anschein hat. Verwendet man bspw. folgende Rasterpunkte

```
Grid->addWidget(but01[2], 0, 0); // Zeile 0; Spalte 0
Grid->addWidget(but02[2], 1, 0); // Zeile 1; Spalte 0
Grid->addWidget(but03[2], 1, 1); // Zeile 1; Spalte 1
Grid->addWidget(but04[2], 2, 1); // Zeile 2; Spalte 1
```

ergibt sich die Anordnung in Abbildung 17.45.

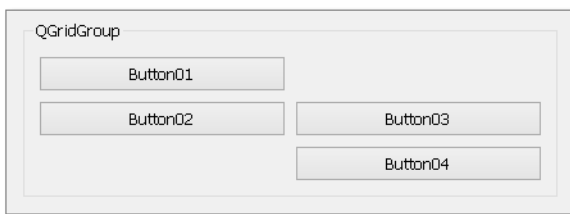


Abbildung 17.45 QGridLayout im Einsatz (1)

Unschön ist hierbei, dass »Button01« und »Button04« relativ verloren herumstehen. Wollen sie, dass die restliche Fläche auch noch mit dem Button ausgefüllt wird, müssen sie bei den Schaltflächen zusätzlich noch angeben, wie weit sich die Abgrenzung der Schaltflächen erstrecken soll. Dies erledigen Sie mit der folgenden Angabe:

```
Grid->addWidget(but01[2], 0, 0, 1, 2);
Grid->addWidget(but02[2], 1, 0);
```

```
Grid->addWidget(but03[2], 1, 1);
Grid->addWidget(but04[2], 2, 0, 1, 2);
```

Bezogen auf die erste Zeile `addWidget(but01[2], 0, 0, 1, 2)` bedeutet dies: Lege die Schaltfläche in Zeile 0 und Spalte 0. Die Schaltfläche soll sich dabei nur über eine Zeile, aber über zwei Spalten erstrecken. Durch diese weiteren Angaben bei `addWidget()` ergibt sich folgende Anordnung in Abbildung 17.46.

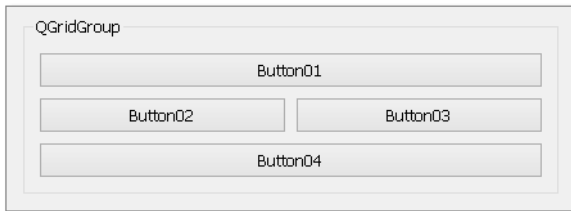


Abbildung 17.46 QGridLayout im Einsatz (2)

Nun zurück zur Implementierung der Klasse `MyLayoutEx`. Am Ende des Beispiels erzeugen wir in Zeile 46 erneut eine vertikale Box, in die in den Zeilen 48 bis 50 die Gruppenbehälter mit den Rahmen hinzugefügt werden. Am Schluss setzen wir noch das Layout (Zeile 52) und den Fenstertitel (Zeile 54).

Jetzt benötigen wir noch eine Hauptfunktion, welche dieses Programm in der Praxis demonstriert, damit Sie Abbildung 17.44 erhalten:

```
00 // kapitel17/layout01/main.cpp
01 #include <QApplication>
02 #include "mylayout.h"

03 int main(int argc, char *argv[]) {
04     QApplication app(argc, argv);
05     MyLayoutEx window;
06     window.show();
07     return app.exec();
08 }
```

QStackedLayout

`QStackedLayout` bietet das Stapeln von mehreren Widgets an, so dass nur ein Widget (oder eine Gruppe von Widgets) sichtbar ist, welche wiederum in einem Layout-Widget angeordnet sein können. Das Prinzip dürften Sie von Tabulatoren her kennen, was Qt mit `QTabWidget` unterstützt. Auf der Buch-CD finden Sie ein Listing dazu (im Ordner `/kapitel17/stapellayout`), welches `QStackedLayout` im Einsatz zeigt. In Abbildung 17.47 bis Abbildung 17.49 sehen Sie das Widget im Einsatz.

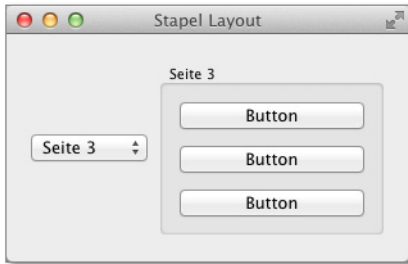


Abbildung 17.47 Auf der linken Seite finden Sie eine Combo-Box (»QComboBox«) und auf der rechten Seite eine Group-Box mit Buttons. Die Widgets, die sich überlagern, befinden Sie auf der rechten Seite und werden hier über die Combo-Box ausgewählt. Wählen Sie bspw. bei der Combo-Box »Seite 3« aus, wird die Group-Box mit den Buttons von einem anderen Widget überlagert, dass sich eben an dieser Position bzw. an diesem Index befindet.

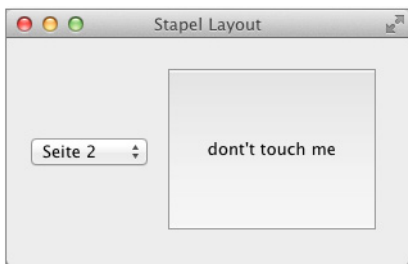


Abbildung 17.48 Jetzt wurde in der Combo-Box »Seite 2« ausgewählt, und rechts wird das entsprechende Widget im Stapel angezeigt.

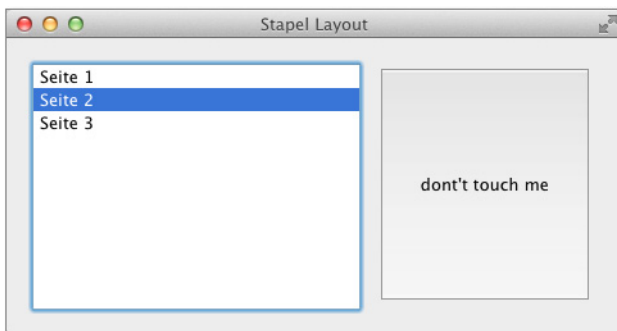


Abbildung 17.49 Statt einer Combo-Box (»QComboBox«), um durch die Stapel zu schalten, wird häufig auch die Listenform mit »QListWidget« verwendet.

Methoden für die Layout-Widgets

Die Widgets für das Layout enthalten enorm viele Methoden, um weitere Dinge wie Ausrichtung, Ausdehnung, Ausfüllen, Abstand usw. einzustellen. Diese hier auf meh-

renen Seiten abzdrukken, wäre ein Unding. Hier müssen Sie den Qt Assistant zu Rate ziehen. Bevor Sie jetzt allerdings hergehen und alles haarklein durcharbeiten, empfehle ich Ihnen, Abschnitt 17.9, »Eigenen Dialog mit dem Qt Designer erstellen«, zu lesen, wo ein wenig darauf eingegangen wird, wie Sie Ihre Widgets mit dem Qt Designer anordnen können. Gerade für solche Dinge bietet sich ein RAD-Tool geradezu an.

Wie dem auch sei, wenn Sie ein wenig in der Dokumentation zu den Methoden für das Layout stöbern wollen, dann sollten Sie etwas höher angelegt bei `QLayout` (Layout-Hierarchie, siehe Abbildung 17.43) anfangen, wo Sie viele allgemeine Methoden für die weiteren Layout-Widgets vorfinden. Dann können Sie sich nach unten durcharbeiten und die Methoden von `QStackedLayout`, `QFormLayout`, `QBoxLayout` (inklusive `QVBoxLayout` und `QHBoxLayout`) und `QGridLayout` studieren.

17.7 Dialoge erstellen mit `QDialog`

Ein Dialogfenster ist ein Top-Level-Fenster, welches gewöhnlich für kurze Anwendereingaben, Einstellungen (Konfigurationen) oder auch nur Informationen für den Anwender verwendet wird. Ein solches Dialogfenster basiert auf der Klasse `QDialog`, welche wiederum von `QWidget` abgeleitet ist (siehe Abbildung 17.50).

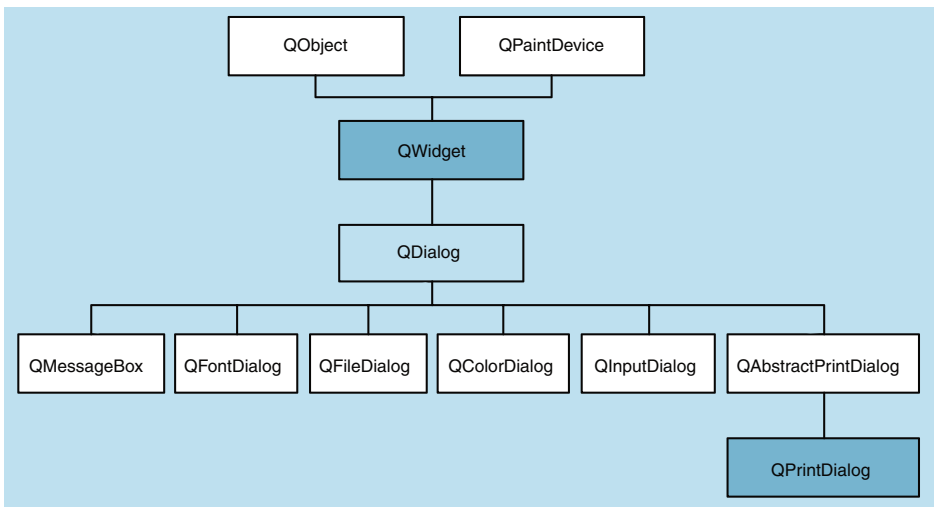


Abbildung 17.50 Klassenhierarchie von »`QDialog`«

Qt bietet eine Menge an vordefinierten Dialogen, wie bspw. Anwendereingaben von Zahlen oder Text, Farbauswahl, Dateiauswahl, Schriftauswahl usw. Darauf wird allerdings erst in Abschnitt 17.8, »Vorgefertigte Dialoge von Qt«, näher eingegangen. Hier beschreiben wir zunächst die »Basisklasse« aller Dialoge mit `QDialog`.

Ein solcher Dialog mit `QDialog` kann modal oder nicht modal sein. Modal bedeutet, dass der Dialog immer vor dem Elternfenster liegt und das Elternfenster so lange blockiert, bis der Dialog geschlossen ist. Damit zwingt man den Anwender quasi, die Interaktion erst auszuführen, bevor er Zugriff auf andere Fenster derselben Anwendung bekommt.

Der übliche Weg, einen modalen Dialog anzuzeigen, ist es, die `exec()`-Funktion zu verwenden. Bei `QDialog` ist diese Methode auch als Slot implementiert. Wenn der Anwender den Dialog schließt, liefert `exec()` einen Rückgabewert zurück. Üblicherweise wird eine Standardschaltfläche (bspw. OK) mit `accept()` und eine weitere Schaltfläche mit `reject()` (bspw. ABBRECHEN) verknüpft.

Alternative zu `exec()`

Alternativ zu `exec()` kann auch die Methode `setModal(true)` und dann `show()` aufgerufen werden, um ein modales Dialogfenster anzuzeigen. Andersherum wird so auch ein nicht modales Dialogfenster angezeigt. Man verwendet die Methode `setModal(false)` und ruft dann `show()` auf.

Hierzu wieder ein einfaches Beispiel. Wir erstellen ein einfaches Fenster (`QWidget`) mit einer Schaltfläche (quasi unser Hauptfenster der Anwendung). Wird die Schaltfläche betätigt, wird ein Dialogfenster angezeigt. Zunächst das Grundgerüst für das Hauptfenster:

```
00 // kapitel17/dialog1/mywidget.h
...
01 #include "mydialog.h"
...
02 class MyWidget : public QWidget {
03     Q_OBJECT
04 public:
05     MyWidget( QString qstr ="Bitte Button betätigen",
                QWidget *parent = 0 );
06 private slots:
07     void checkInputDialog();
08 private:
09     QPushButton *button0;
10     QLabel*      label1, *label2;
11     QVBoxLayout* layout;
12     MyDialog*    dialog;
13 };
...
```

Kein echtes Hauptfenster

Unser Hauptfenster in diesem Beispiel mit der Klasse `MyWidget` ist kein Hauptfenster im eigentlichen Sinne. Ein richtiges Hauptfenster erstellen Sie mit `QWindow` (siehe Abschnitt 17.11, »Anwendungen in einem Hauptfenster«).

Nach dem Grundgerüst des Hauptfensters folgt jetzt das Grundgerüst der Headerdatei `mydialog.h` (Zeile 01) für unseren Dialog, welchen Sie hier in Zeile 12 von `mywidget.h` vorfinden. Hier das Grundgerüst dazu:

```
00 // kapitel17/dialog1/mydialog.h
...
01 #include <QDialog>
...
02 class MyDialog : public QDialog {
03     Q_OBJECT
04 public:
05     MyDialog();
06 };
...
```

Natürlich müssen Sie hierbei auch das Makro `Q_OBJECT` (Zeile 03) verwenden, wie für alle direkten (und indirekten) Ableitungen von Basisklassen (hier `QObject`), sonst würde das Signal-Slot-Konzept nicht funktionieren. Dies ist ein gerne gemachter Fehler, der häufig auch nicht gleich als solcher bemerkt wird, weil weder Compiler noch Linker hier eine Fehlermeldung ausgeben. Eingerichtete Signal-Slot-Verbindungen ohne das Makro werden ignoriert.

Weiter mit der Implementierung der Klasse `MyWidget`, unserem Hauptfenster:

```
00 // kapitel17/dialog1/mywidget.cpp
01 #include "mywidget.h"
02 #include "mydialog.h"
03 #include <QApplication>

04 // Neue Widget-Klasse vom eigentlichen Widget ableiten
05 MyWidget::MyWidget(QString lab, QWidget *parent):
    QWidget(parent) {
06     // Elemente des Widgets erzeugen
07     button0 = new QPushButton ("Dialog starten");
08     layout  = new QVBoxLayout(this);
09     label1  = new QLabel(lab);
10     label2  = new QLabel;
11     dialog = new MyDialog;
```

```

12 // Programm nicht beenden, wenn Dialog zerstört wird
13 dialog->setAttribute(Qt::WA_QuitOnClose);
14 // Elemente des Widgets anordnen/anpassen
15 layout->addWidget(label1);
16 layout->addWidget(button0);
17 layout->addWidget(label2);
18 // Signal-Slot-Verbindungen einrichten
19 connect( button0, SIGNAL( clicked() ),
           dialog, SLOT( exec() ) );
20 connect( dialog, SIGNAL( accepted() ),
           this, SLOT( checkInputDialog() ) );
21 connect( dialog, SIGNAL( rejected() ),
           this, SLOT( checkInputDialog() ) );
22 setWindowTitle("Hauptfenster - Anwendung");
23 }

24 void MyWidget::checkInputDialog( ) {
25     int val = dialog->result();
26     if( val == QDialog::Accepted )
27         label2->setText("\0k\" wurde gewählt");
28     else if( val == QDialog::Rejected )
29         label2->setText("\Abbrechen\" wurde gewählt");
30 }

```

In Zeile 11 erzeugen wir das neue Dialogfenster. Mit dem Attribut `Qt::WA_QuitOnClose`, welches in Zeile 13 gesetzt wird, legen wir fest, dass sich beim Beenden des Dialogfensters nicht gleich die ganze Anwendung beendet. Bei der Ausführung sieht das »Hauptfenster« zunächst aus wie in Abbildung 17.51.

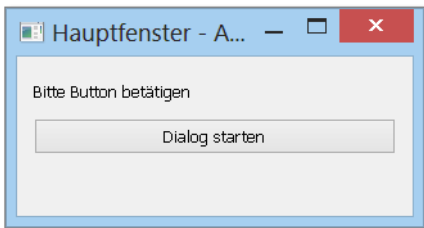


Abbildung 17.51 Das »Hauptfenster« bei der Ausführung

Die Signal-Slot-Verbindung für den Button legen wir in Zeile 19 fest. Erhält der Button das Signal `clicked()`, wird der Slot `exec()` von `QDialog` ausgeführt. Damit wird dann ein modales Dialogfenster angezeigt (siehe Abbildung 17.52). In den Zeilen 20 und 21 richten wir zwei weitere Signal-Slot-Verbindungen ein. Die erste Signal-Slot-Verbindung von Zeile 20 wird aktiv, wenn beim Dialogfenster das Signal `accepted()` ausge-

löst wurde. Die andere Signal-Slot-Verbindung (Zeile 21) wird beim Signal `rejected()` aktiv. Für beide Signale wird unsere eigene Slot-Methode `checkInputDialog()` aufgerufen. Wie die Signale ausgelöst werden, erfahren Sie gleich.

In der eigenen Slot-Methode `checkInputDialog()` werten wir in Zeile 25 das Ergebnis des Signals mit der Methode `result()` aus. Hierbei wird entweder der enum-Typ `Accepted` oder `Rejected` von `QDialog::DialogCode` zurückgegeben. Entsprechend des Rückgabewertes von `result()` wird das Label in Zeile 27 oder 29 gesetzt.

Jetzt fehlt uns noch die Implementierung des Dialogfensters mit *mydialog.cpp*:

```
00 // kapitel17/dialog1/mydialog.cpp
01 #include "mydialog.h"

02 // Neue Widget-Klasse vom eigentlichen Widget ableiten
03 MyDialog::MyDialog() {
04     setFixedSize ( 150, 100 );
05     QVBoxLayout *vbox = new QVBoxLayout;
06     QLabel *label = new QLabel("Bitte Button betätigen");
07     QPushButton *button01 = new QPushButton("Ok");
08     QPushButton *button02 = new QPushButton("Abbrechen");
09     button02->setDefault(true);
10     vbox->addWidget(label);
11     vbox->addWidget(button01);
12     vbox->addWidget(button02);
13     setLayout(vbox);
14     connect( button01, SIGNAL( clicked() ),
15             this, SLOT( accept() ) );
16     connect( button02, SIGNAL( clicked() ),
17             this, SLOT( reject() ) );
18 }
```

Die Ausführung des Dialogs sieht wie in Abbildung 17.52 aus.

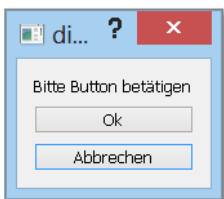


Abbildung 17.52 Der Dialog bei der Ausführung

In Zeile 09 legen wir die ABBRECHEN-Schaltfläche als Standardschaltfläche fest. Das bedeutet, drückt der Anwender auf , wird die ABBRECHEN-Schaltfläche als gedrückt ausgelöst.

In Zeile 14 wird die Signal-Slot-Verbindung der OK-Schaltfläche eingerichtet. Erhält dieser Button das `clicked()`-Signal, dann wird der Slot `accept()` von `QDialog` ausgeführt. Dies sorgt dann bei der Klasse `MyWidget` dafür, dass das Signal `accepted()` der entsprechenden Signal-Slot-Verbindung ausgelöst wird.

Gleiches gilt für die Signal-Slot-Verbindung in Zeile 15, nur gilt dies dann für das Signal `rejected()` auf der anderen Seite. In beiden Fällen wird hierbei die Slot-Methode `checkInputDialog()` von `MyWidget` aufgerufen und entsprechend ausgewertet. Der Slot `reject()` wird ebenfalls ausgeführt, wenn der Anwender die `[Esc]`-Taste betätigt oder das Fenster schließt. Beides entspricht hierbei ebenfalls dem »Anklicken« der ABBRECHEN-Schaltfläche. Wurde bspw. die OK-Schaltfläche betätigt, wird das Dialogfenster wieder geschlossen und im Hauptfenster ein entsprechendes Text-Label gesetzt (Abbildung 17.53).

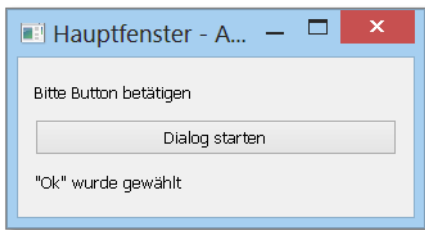


Abbildung 17.53 Die Ok-Schaltfläche im Dialog zuvor (Abbildung 17.52) wurde betätigt.

Beachten Sie dabei, dass das Dialogfenster hierbei nur »versteckt« und nicht zerstört wird.

Nochmals genauer: Wurde die OK-Schaltfläche betätigt, wird das Signal `clicked()` ausgelöst und der Slot `accept()` ausgeführt. Bei `MyWidget` wiederum wird das Signal `accepted()` ausgelöst und der eigene Slot `checkInputDialog()` gestartet. Darin wird mit der Methode `result()` überprüft, ob `Accepted` oder `Rejected` betätigt wurde. `Accepted` wurde in unserem Fall an der OK-Schaltfläche und `Rejected` an der ABBRECHEN-Schaltfläche des Dialogs vergeben. Analog verhält sich dies beim Betätigen der ABBRECHEN-Schaltfläche, nur eben mit `reject()` und `rejected()`.

Hierzu noch eine `main()`-Funktion, mit der Sie das Ganze auch in der Praxis testen können:

```
00 // kapitel17/dialog1/main.cpp
01 #include <QApplication>
02 #include "mywidget.h"

03 int main(int argc, char *argv[]) {
04     QApplication app(argc, argv);
05     MyWidget window;
```

```

06     window.show();
07     return app.exec();
08 }

```

Benutzerdefinierten Wert aus einem Dialog zurückgeben

Den einen oder anderen dürfte jetzt brennend interessieren, wie man denn nun einen anderen Wert als `Accepted` und `Rejected` aus einem Dialogfenster zurückgeben kann. Erweitern Sie dazu am besten zunächst unseren bereits erstellten Dialog `MyDialog` um einen Slot:

```

00 // kapitel17/dialog2/mydialog.h
...
01 class MyDialog : public QDialog {
02     Q_OBJECT
03 public:
04     MyDialog();
05     public slots:
06         void mySetResult();
07 };
...

```

Fügen Sie jetzt bei der Definition der Klasse eine neue Schaltfläche der vertikalen Layout-Box hinzu (siehe Abbildung 17.54):

```

// kapitel17/dialog2/mydialog.cpp
...
QPushButton *button03 = new QPushButton("Ignorieren");
...
vbox->addWidget(button03);
...

```

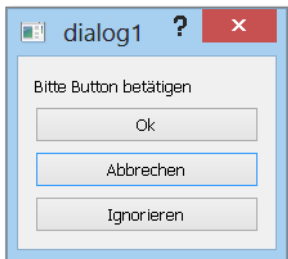


Abbildung 17.54 Eine weitere Schaltfläche »Ignorieren« wurde hinzugefügt.

Für diese Schaltfläche richten Sie nun eine Signal-Slot-Verbindung zu unserem neu implementierten Slot mit ein:

```
// kapitel17/dialog2/mydialog.cpp
...
connect( button03, SIGNAL ( clicked()),
        this, SLOT( mySetResult() ) );
...
```

Die Implementierung des Slots, der aufgerufen wird, wenn die neue Schaltfläche betätigt wurde, sieht folgendermaßen aus:

```
// kapitel17/dialog2/mydialog.cpp
...
void MyDialog::mySetResult() {
    int result = 99;
    emit done(result);
}
```

Hierbei rufen Sie die echte Slot-Methode `done()` mit dem Rückgabewert 99 auf, womit der Dialog geschlossen wird.

Jetzt fügen wir in *mywidget.h* einen weiteren Slot hinzu – genauer, wir überladen einfach den alten Slot mit einem Integer als Argument:

```
// kapitel17/dialog2/mywidget.h
...
private slots:
    void checkInputDialog();
    void checkInputDialog(int);
...
```

Natürlich richten wir auch hier eine neue Signal-Slot-Verbindung ein, die auf den Slot `done(int)` der Klasse `MyDialog` reagiert. Hierzu eignet sich das Signal `finished(int)`, welches ebenfalls ein Integer als Parameter hat. Als Slot für das Signal verwenden wir wieder unsere selbst implementierte und überladene Slot-Methode:

```
// kapitel17/dialog2/mywidget.cpp
...
connect( dialog, SIGNAL ( finished( int )),
        this, SLOT( checkInputDialog( int ) ) );
...
```

Jetzt noch die Definition der überladenen Slot-Methode in *mywidget.cpp* hinzufügen:

```
// kapitel17/dialog2/mywidget.cpp
...
```

```
void MyWidget::checkInputDialog(int val) {
    //val = dialog->result();
    if( val == 99 ) {
        label2->setText("\t\"Ignorieren\" wurde gewählt");
    }
}
```

Mehr ist nicht nötig, um jetzt auch auf andere Rückgabewerte als `Accepted` oder `Rejected` zu reagieren. Im Beispiel wurde auf den Rückgabewert 99 reagiert (Abbildung 17.55).

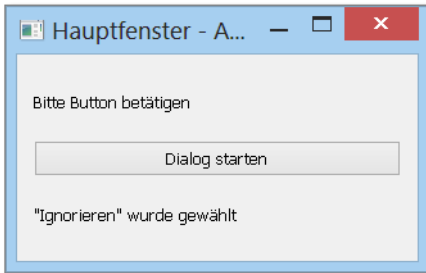


Abbildung 17.55 Unsere Anwendung reagiert jetzt auch auf andere Rückgabewerte vom Dialog als »Accepted« und »Rejected«.

QDialog und seine Methoden

Sie kennen jetzt die grundlegende Verwendung von Dialogen (mit `QDialog`) in Qt. Trotzdem kommen Sie auch hier nicht um ein intensiveres Studieren der Klasse(n rund um) `QDialog` und seiner Mitglieder herum.

17.8 Vorgefertigte Dialoge von Qt

Es wurde bereits im Abschnitt zuvor erwähnt, dass Qt bereits sinnvolle und vorgefertigte Dialoge anbietet. In diesem Abschnitt finden Sie einen kurzen Überblick dazu. Wie immer gilt auch hier, für detailliertere Informationen sollten Sie die sehr gute Referenz von Qt mit dem Assistant verwenden.

17.8.1 QMessageBox – der klassische Nachrichtendialog

Relativ häufig wird die von `QDialog` abgeleitete Klasse `QMessageBox` eingesetzt, um dem Anwender kurze Informationen zu vermitteln oder diesen gewisse Entscheidungen treffen zu lassen. Gewöhnlich wird dieser modale Dialog mit einer kurzen

Nachricht, einem Icon und Schaltflächen angezeigt. Das Aussehen der Icons bzw. des Buttons hängt vom aktuellen Fensterstil und zum Teil auch vom System ab (Abbildung 17.56 bis Abbildung 17.58).

Der einfachste Weg, eine solche Nachrichtenbox anzuzeigen, ist es, die statischen Funktionen `QMessageBox::information()`, `QMessageBox::question()`, `QMessageBox::critical()` und `QMessageBox::warning()` zu verwenden.



Abbildung 17.56 »QMessageBox::warning« unter Mac OS X

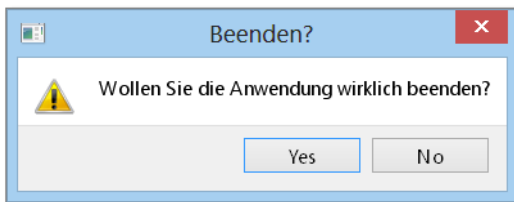


Abbildung 17.57 »QMessageBox::warning« unter Windows

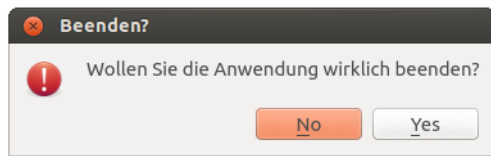


Abbildung 17.58 »QMessageBox::warning« unter Linux

Um diese Nachrichtenbox anzuzeigen, wurde nur Folgendes geschrieben:

```
#include <QMessageBox> // benötigte Headerdatei
...
int ret = QMessageBox::warning(
    this, "Beenden?",
    "Wollen Sie die Anwendung wirklich beenden?",
    QMessageBox::Yes | QMessageBox::No );
...
```

Alternativ können Sie hierbei auch den Konstruktor von `QMessageBox` verwenden, um eine Nachrichtenbox mit Icon und Schaltflächen zu erzeugen. Bezogen auf Abbildung 17.56 bis Abbildung 17.58 sähe dies dann folgendermaßen aus:

```
#include <QMessageBox> // benötigte Headerdatei
...
QString caption("Beenden?");
QString text("Wollen Sie die Anwendung wirklich beenden?");
QMessageBox msg( QMessageBox::Warning, caption,
                text, QMessageBox::Yes | QMessageBox::No );
msg.exec()
...
```

Mit dem ersten Argument geben Sie hier das Icon für die Nachrichtenbox an. Hierbei können Sie aus folgenden vier Symbolen auswählen (neben dem fünften, `QMessageBox::Warning`, das Sie bereits kennen):

- ▶ `QMessageBox::Critical` (schwerwiegender Fehler)
- ▶ `QMessageBox::NoIcon` (kein Icon anzeigen)
- ▶ `QMessageBox::Information` (Icon für Informationen)
- ▶ `QMessageBox::Question` (Icon für Dialogbox mit Fragen)

Sollten Sie ein eigenes Icon hinzufügen wollen, steht Ihnen dafür die Methode `setIconPixmap()` aus `QMessageBox` zur Verfügung.

Mit den Argumenten zwei und drei von `QMessageBox` geben Sie den Fenstertitel und den eigentlichen Nachrichtentext an. Der Nachrichtentext kann, wie alle Qt-Dialoge, HTML-like formatiert werden. Ändern Sie bspw. den String `text` folgendermaßen ab:

```
QString text( "Wollen <i>Sie</i> die "  
             "Anwendung wirklich <u>beenden</u>?" );
```

Dadurch erhält der Nachrichtentext in der Nachrichtenbox das Aussehen, das Sie in Abbildung 17.59 sehen.



Abbildung 17.59 Nachricht mit HTML formatiert

Mit den weiteren Argumenten legen Sie die verschiedenen Schaltflächen fest, wovon `QMessageBox` mehrere vordefiniert, wie Sie mit `QMessageBox::Yes` und `QMessageBox::No` bereits gesehen haben. Hier gibt es bspw. noch `QMessageBox::Ok`, `QMessageBox::Cancel` und noch einige mehr. Hierzu empfehle ich Ihnen, einen Blick in die Dokumentation von `QMessageBox` zu werfen.

QMessageBox mit leerem Konstruktor

Es ist auch möglich, mit einem leeren `QMessageBox`-Konstruktor einen Nachrichtendialog zu erstellen. Hierbei stehen dem Anwender dann verschiedene `set`-Zugriffsmethoden zur Verfügung, um Icon, Titel oder Buttons nachträglich anzugeben.

Nachrichtendialog von QMessageBox auswerten

Auch das Auswerten der Nachrichtendialoge von `QMessageBox` ist recht einfach. Im Grunde müssen Sie nur überprüfen, ob die Schaltfläche mit der entsprechenden Konstante (bspw. `QMessageBox::Yes` oder `QMessageBox::No`) gedrückt wurde. Hierzu wieder ein einfaches Beispiel, zunächst das Grundgerüst:

```
00 // kapitel17/qmessagebox/mywidget.h
...
01 class MyWidget : public QWidget {
02     Q_OBJECT
03 public:
04     MyWidget( QString qstr = "Bitte Button betätigen",
               QString but = "Ende", QWidget *parent = 0 );
05 private:
06     QPushButton *button0;
07     QLabel*      label;
08     QVBoxLayout* layout;
09 public slots:
10     void myquit();
11 };
...
```

Dieses Beispiel haben Sie bereits in ähnlicher Form einmal verwendet. Die Auswertung des Dialogs wird mit der eigenen Slot-Methode `myquit()` (Zeile 09 und 10) realisiert. Hierzu jetzt die Implementierung der Klasse `MyWidget`:

```
00 // kapitel17/qmessagebox/mywidget.cpp
01 #include "mywidget.h"
02 #include <QApplication>

03 // Neue Widget-Klasse vom eigentlichen Widget ableiten
```



```

04 MyWidget::MyWidget(QString lab,QString but,QWidget *parent):
    QWidget(parent) {
05     // Elemente des Widgets erzeugen
06     button0 = new QPushButton (but);
07     layout  = new QVBoxLayout(this);
08     label   = new QLabel(lab);
09     // Elemente des Widgets anordnen/anpassen
10     layout->addWidget(label);
11     layout->addWidget(button0);
12     // Signale des Widgets einrichten
13     connect( button0, SIGNAL( clicked() ),
              this, SLOT( myquit() ) );
14 }

15 void MyWidget::myquit() {
16     int ret = QMessageBox::warning( this, "Beenden?",
17         "Wollen Sie die Anwendung wirklich beenden?",
18         QMessageBox::Yes | QMessageBox::No );
19     if( ret == QMessageBox::Yes )
20         close();
21 }

```

Die entscheidende Zeile ist hier zunächst Zeile 13, wo Sie für die Schaltfläche des Hauptfensters eine Signal-Slot-Verbindung einrichten. Wird die Schaltfläche gedrückt (`clicked()`), wird der eigene Slot `myquit()` ausgeführt.

In der Slot-Methode `myquit()` (Zeile 15 bis 21) erzeugen wir zunächst mit der statischen Funktion `QMessageBox::warning()` einen Nachrichtendialog, wie Sie diesen bereits in Abbildung 17.56 bis Abbildung 17.58 gesehen haben. Die statischen Funktionen von `QMessageBox` geben als Rückgabewert ein Integer zurück. Welcher Wert, bzw. genauer, welche Schaltfläche gedrückt wurde, werten wir in Zeile 19 aus. Entspricht der Rückgabewert in `ret` der Konstante `QMessageBox::Yes`, dann wurde die Yes-Schaltfläche betätigt, und wir beenden die Anwendung mit `close()`. Ansonsten wurde die No-Schaltfläche gedrückt, was allerdings in diesem Fall nicht mehr ausgewertet werden muss.

Ähnlich funktioniert dies auch mit Objekten von `QMessageBox`, die mit dem Konstruktor erstellt wurden. Das gleiche Beispiel lässt sich auch folgendermaßen erstellen:

```

...
void MyWidget::myquit() {
    QString caption("Beenden?");
    QString te("Wollen Sie die Anwendung wirklich beenden?");
    QMessageBox msg( QMessageBox::Warning, caption,

```

```
        te, QMessageBox::Yes | QMessageBox::No );
    if( msg.exec() == QMessageBox::Yes )
        close();
}
```

Sollten Sie mehrere Schaltflächen verwenden und überprüfen wollen, lässt sich auch ein switch()-Schalter dafür verwenden:

```
...
switch ( msg.exec() ) {
    case QMessageBox::Yes: // Yes wurde betätigt
        break;
    case QMessageBox::No: // No wurde betätigt
        break;
    ...
}
```

Eigene Schaltflächen für QMessageBox

Sollten Ihnen die vordefinierten Konstanten für die Schaltflächen nicht ausreichen, können Sie gerne eine eigene Schaltfläche mit der mehrfach überladenen Methode addButton() hinzufügen, der Schaltfläche einen eigenen Text vergeben und mit einer vordefinierten Schaltflächen-Funktion versehen (Abbildung 17.60). Einen Überblick zu einigen Schaltflächen-Funktionen finden Sie in Tabelle 17.2.

Konstante	Beschreibung
QMessageBox::AcceptRole	Wird die Schaltfläche betätigt, wird der Dialog als akzeptiert (accepted) gewertet (bspw. OK).
QMessageBox::RejectRole	Wird die Schaltfläche betätigt, wird der Dialog als abgelehnt (rejected) gewertet (bspw. CANCEL).
QMessageBox::YesRole	ein Yes-ähnlicher Button
QMessageBox::NoRole	ein No-ähnlicher Button

Tabelle 17.2 Einige vordefinierte Schaltflächen-Funktionen

Keine Signale und Slots

QMessageBox besitzt keine Signale und Slots, weshalb Sie auf die vordefinierten Schaltflächen-Funktionen zugreifen müssen, sollten Sie eigene verwenden wollen.

In der Praxis wird die Methode addButton() folgendermaßen verwendet:

```

void MyWidget::myquit() {
    QString caption("Beenden?");
    QString te("Wollen Sie die Anwendung wirklich beenden?");
    QMessageBox msg( QMessageBox::Warning, caption, te );
    QPushButton* yesButton =
        msg.addButton("Ja, sicher!", QMessageBox::YesRole);
    QPushButton* noButton =
        msg.addButton("Keine Lust!", QMessageBox::NoRole);
    msg.exec();
    if( msg.clickedButton() == yesButton )
        close();
    else if( msg.clickedButton() == noButton )
        ;// Nein, wurde ausgewählt
}

```

Wenn Sie, wie hier im Beispiel, eine Instanz von `QMessageBox` mit eigenen Schaltflächen verwenden, können Sie den Wert der Schaltfläche, die gedrückt wurde, mit der Methode `clickedButton()` nach dem Aufruf von `exec()` abfragen.

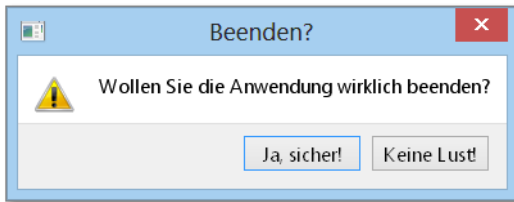


Abbildung 17.60 Eigene Schaltflächen für `QMessageBox`

17.8.2 `QFileDialog` – Dialog zur Dateiauswahl

Die Klasse `QFileDialog` wurde ebenfalls von `QDialog` abgeleitet und zeigt einen Dialog an, mit dem es möglich ist, eine Datei oder ein Verzeichnis auszuwählen. Wie auch schon bei `QMessageBox` bietet `QFileDialog` zwei Wege an. Zum einen sind auch hierbei wieder vorgefertigte statische Funktionen vorhanden, und zum anderen ist es auch hiermit wieder möglich, eine Instanz der Klasse über den Konstruktor zu erzeugen.

`QFileDialog` mit statischer Funktion

Den einfachsten Weg, einen solchen Dateiauswahldialog zu erzeugen, bietet wieder die statische Funktion an. Abhängig vom Betriebssystem wird hierbei immer der native Dateiauswahldialog angezeigt. Das bedeutet, damit bekommt der Anwender den Dialog zu sehen, den er auf seinem System gewohnt ist. Es wird also der Dateiauswahldialog des Betriebssystems verwendet. Dies kann gegebenenfalls auch vermieden werden, wenn die Option `QtFileDialog::DontUseNativeDialog` gesetzt wird.

Am besten hierzu ein einfaches Beispiel, in dem wir wieder die Slot-Methode `MyWidget::myquit()` aus dem Beispiel `qmessagebox/mywidget.cpp` zuvor umschreiben:

```
00 // kapitel17/qfiledialog/mywidget.cpp
...
01 #include <QFileDialog>
02 #include <QMessageBox>
...
03 void MyWidget::myopenfile() {
04     QString file = QFileDialog::getOpenFileName(
        this, "Bitte eine Datei auswählen",
        QDir::homePath(), "Dokumente (*.cpp *.cc *.h)" );
05     if( !file.isNull() ) {
06         QString info("Folgende Datei wurde ausgewählt:\n");
07         info.append(file);
08         QMessageBox::information(
            this, "Ihre Auswahl", info, QMessageBox::Ok );
09     }
10 }
```

Abhängig vom System, auf dem Sie das Beispiel ausführen, sieht der Dateiauswahl-dialog aus (Abbildung 17.61).

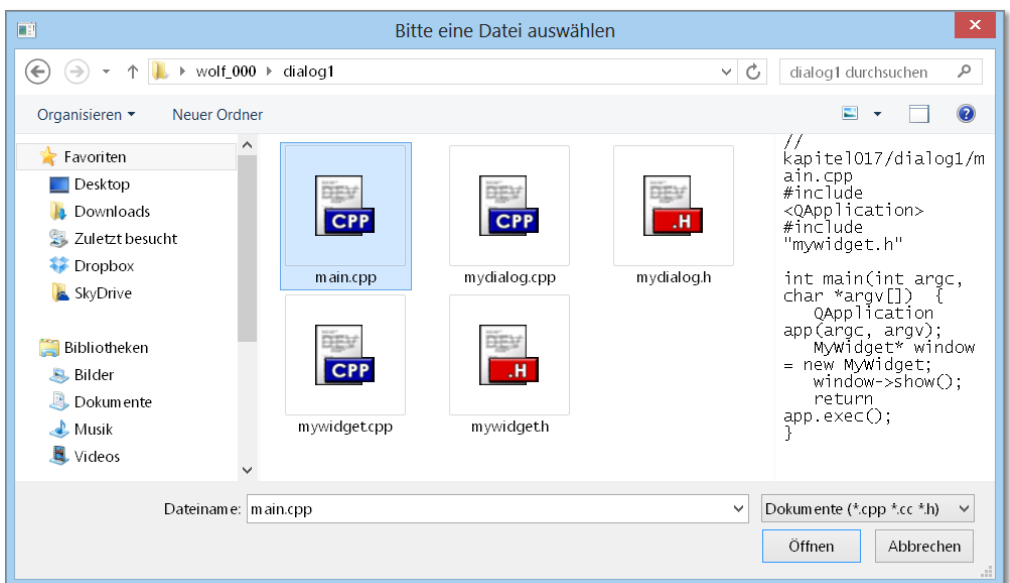


Abbildung 17.61 Dateiauswahldialog mit »`QFileDialog::getOpenFileName()`« unter Windows

Sollten Sie eine Datei (hier eine mit der Endung `.cpp`, `.cc` oder `.h`) ausgewählt haben (wird überprüft mit `!file.isNull()` in Zeile 05), wird dies im folgenden Nachrichten-dialog angezeigt. Hierzu eine etwas genauere Beschreibung.

Nachdem Sie in `QFileDialog::getOpenFileName` in Zeile 04 mit dem ersten Parameter das Eltern-Widget angegeben haben, wird mit dem zweiten Parameter der Titel des Fensters vergeben. Beachten Sie hierbei: Wenn Sie beim ersten Parameter 0 angeben, wird der Dialog nicht modal angezeigt. Mit dem dritten Parameter können Sie das Startverzeichnis angeben. Hierbei sind relative und absolute Pfadnamen erlaubt.

Im Beispiel haben wir mit `QDir::homePath()` eine statische Methode der Klasse `QDir` verwendet. Unter Linux/Unix und auch Mac OS X wird hierbei die Umgebungsvariable `HOME` verwendet. Ist diese nicht gesetzt, wird das Wurzelverzeichnis verwendet. Unter MS-Windows wird ebenfalls nach einer Umgebungsvariablen `HOME` gesucht. Existiert diese nicht, wird die Umgebungsvariable `USERPROFILE` verwendet. Existiert auch diese nicht, wird von Qt `HOMEDRIVE` und `HOMEPATH` verwendet. Neben `QDir::homePath()` finden Sie in `QDir` noch weitere statische Methoden, wie `QDir::currentPath()` (aktuelles Verzeichnis der Anwendung), `QDir::rootPath()` (oberstes Wurzelverzeichnis) oder `QDir::tempPath()` (Pfad zum temporären Verzeichnis).

Der letzte Parameter von `QFileDialog::getOpenFileName()` ist ein Dateifilter. Die Syntax hierzu ist ganz einfach:

```
"Bezeichner (*.ext1 *.ext2 *.ext3)"
```

Der »Bezeichner« ist frei wählbar. Dann verwenden Sie die Dateierendungen, die Sie in den Filter miteinbeziehen wollen, genauer, die im Dateiauswahldialog angezeigt werden sollen. Wollen Sie mehrere Dateifilter verwenden, müssen Sie nur am Ende der geklammerten Filter zwei Semikolons setzen. Zum Beispiel:

```
QString file = QFileDialog::getOpenFileName(
    this, "Bitte eine Datei auswählen",
    QDir::homePath(), "Dokumente (*.pdf *.ps *.doc);;"
    "Bilder (*.jpg *.png *.gif);;"
    "Quelldateien (*.cpp *.cc *.h);;"
    "Alle Dateien (*.*)");
```

Hierdurch können Sie jetzt über eine Dropdown-Liste die hinzugefügten Filter verwenden (siehe Abbildung 17.62).

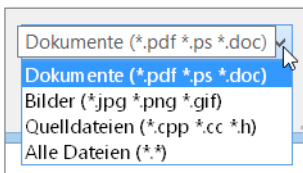


Abbildung 17.62 Mehrere Dateifilter verwenden

Weitere statische Funktionen

Neben der statischen Funktion `QFileDialog::getOpenFileName()` gibt es noch weitere Versionen, wie z.B. `QFileDialog::getOpenFileNames()`, `QFileDialog::getSaveFileNames()`. Mehr zu diesen Dateidialogen entnehmen Sie wie immer der Dokumentation von Qt.

QFileDialog mit Konstruktor

Natürlich ist es auch möglich, einen eigenen `QFileDialog` ohne statische Funktionen zu erzeugen. Bezogen auf das statische Beispiel, sieht die Verwendung mit dem Konstruktor der Klasse `QFileDialog` folgendermaßen aus:

```
// kapitel17/qfiledialog2/mywidget.cpp
...
01 void MyWidget::myopenfile() {
02     QFileDialog* filedlg = new QFileDialog(
03         this, "Bitte eine Datei auswählen");
04     filedlg->setFileMode(QFileDialog::ExistingFiles);
05     filedlg->setDirectory(QDir::homePath());
06     filedlg->setNameFilter("Dateien (*.cpp *.cxx *.cc)");
07     filedlg->setViewMode(QFileDialog::Detail);
08     QStringList fileNames;
09     if (filedlg->exec())
10         fileNames = filedlg->selectedFiles();
11     QString info("Ihre Auswahl:\n");
12     info.append(fileNames.join("\n"));
13     QMessageBox::information(
14         this, "Ihre Auswahl", info, QMessageBox::Ok );
15 }
```

Mit der Methode `setFileMode()` in Zeile 03 geben Sie an, was der Anwender auswählen muss. Die möglichen Modi und deren Bedeutung sind in Tabelle 17.3 aufgelistet.

Konstante	Beschreibung
<code>QFileDialog::AnyFile</code>	der Name einer Datei, egal, ob diese existiert oder nicht
<code>QFileDialog::ExistingFile</code>	der Name einer einzelnen existierenden Datei

Tabelle 17.3 Modus für »`QFileDialog::setMode()`«

Konstante	Beschreibung
QFileDialog::Directory	Der Name eines Verzeichnisses. Trotzdem werden Dateien und Verzeichnisse angezeigt.
QFileDialog::DirectoryOnly	Der Name eines Verzeichnisses. Außerdem werden auch nur Verzeichnisse angezeigt.
QFileDialog::ExistingFiles	der Name einer oder mehrerer existierenden/r Datei(en)

Tabelle 17.3 Modus für »QFileDialog::setMode()« (Forts.)

Mit der Methode `setDirectory()` in Zeile 04 geben Sie das Verzeichnis vor, in dem der Dateiauswahldialog anfängt. Einen Filter setzen Sie mit der Methode `setNameFilter()` in Zeile 05.

QStringList
Mehr Informationen zur hier verwendeten Klasse `QStringList` finden Sie in Abschnitt 17.16.3, »`QStringList`«, und in der Dokumentation von Qt.

Der Dateiauswahldialog hat zwei Ansichten: `QFileDialog::List` und `QFileDialog::Detail`. `QFileDialog::List` zeigt den Inhalt des aktuellen Verzeichnisses mit den Dateien und Verzeichnissen an. `QFileDialog::Detail` hingegen zeigt neben den Dateien und Verzeichnissen Details wie die Größe, den Dateityp und das Veränderungsdatum an. Die entsprechende Ansicht wird mit der Methode `setViewMode()` in Zeile 06 gesetzt.

Die wichtigste Funktion, wenn Sie einen eigenen Dateiauswahldialog erzeugen, ist `selectFiles()` aus Zeile 09 (es gibt auch `selectFile()` für eine einzelne Auswahl). In unserem Beispiel wird der modale Dialog zunächst mit `exec()` in Zeile 08 angezeigt. Wenn der User OK (oder OPEN bzw. ÖFFNEN) anklickt, stehen die Dateinamen im Beispiel in der String-Liste `QStringList fileNames`. Da wir unseren Dialog mit dem Modus `QFileDialog::ExistingFiles` erzeugt haben, können hierbei durchaus mehrere Dateien ausgewählt werden (mit gehaltener `[Strg]`-Taste).

Wir teilen diese String-Liste mit der Methode `QStringList::join()` in einzelnen Hap-pen und hängen diese zeilenweise (mit `QString::append()`) an `QString info` (siehe Zeile 11). Die ausgewählten Dateien werden anschließend mit dem Nachrichtendialog ausgegeben.

17.8.3 QInputDialog – Dialog zur Eingabe von Daten

Für die Eingabe von Zahlen und Zeichenketten bietet Qt mit `QInputDialog` (ebenfalls von `QDialog` abgeleitet) auch vorgefertigte statische Eingabe-Dialoge an. In diesem Fall gibt es nur die statischen Methoden und keine Möglichkeit, eigene Instanzen dieser Klasse zu erzeugen (was allerdings auch gar nicht nötig ist). Alle Dialoge bieten jeweils die Buttons OK und ABBRECHEN (bzw. CANCEL) an. Folgende vier statische Funktionen können dabei verwendet werden:

- ▶ `QInputDialog::getText()`: Liest einen eingegebenen String vom Anwender ein (Abbildung 17.63).
- ▶ `QInputDialog::getDouble()`: Damit kann eine Gleitkommazahl eingelesen werden (Abbildung 17.64).
- ▶ `QInputDialog::getInt()`: Liest eine Ganzzahl vom Anwender ein (Abbildung 17.65).
- ▶ `QInputDialog::getItem()`: Lässt den Anwender aus mehreren vorgegebenen Strings auswählen (Abbildung 17.66).

Für die genaue Syntax und die einzelnen Parameter empfehle ich Ihnen, wieder die Dokumentation von Qt zu verwenden.

Hierzu soll nur kurz ein Code geschrieben werden, welcher alle vier Eingabe-Dialoge (inklusive ihrer Auswertung) in der Praxis demonstrieren soll:

```
00 // kapitel17/qinputdialog/mywidget.cpp
01 #include <QInputDialog>
...
02 void MyWidget::myinputs() {
03     bool ok;
04     QString text = QInputDialog::getText(
        this, "QInputDialog::getText()", "Ihr Name :",
        QLineEdit::Normal, "Name eingeben", &ok);
05     if (ok && !text.isEmpty())
06         QMessageBox::information(this, "Ihre Eingabe", text, QMessageBox::Ok );

07     double dvalue = QInputDialog::getDouble(
        this, "QInputDialog::getDouble()", "Wert eingeben :",
        55.555, 0, 100, 3, &ok );
08     if(ok) {
09         QString qsdvalue = QString("%1").arg(dvalue);
10         QMessageBox::information(
            this, "Ihr double-Wert ", qsdvalue, QMessageBox::Ok);
11     }
```



```

12  int ivalue = QInputDialog::getInt(
13      this, "QInputDialog::getInt()", "Wert eingeben :",10, 0, 20, 1, &ok );
14  if(ok) {
15      QString qsivalue = QString("%1").arg(ivalue);
16      QMessageBox::information(
17          this, "Ihr Integer-Wert ", qsivalue,
18          QMessageBox::Ok );
19  }

17  QStringList items;
18  items << "Sonne" << "Regen" << "Nebel"
19      << "Wind" << "Schnee";
20  QString itemvalue = QInputDialog::getItem(
21      this,"QInputDialog::getItem()", "Ihr Lieblingswetter:",
22      items, 0, false, &ok );
23  if(ok) {
24      QMessageBox::information(
25          this, "Ihr Lieblingswetter",
26          itemvalue, QMessageBox::Ok );
27  }
28  }
29  }

```

Die folgenden vier Abbildungen sollen unser Programm mit den Dialogen zur Eingabe von Daten bei ihrer Ausführung zeigen.

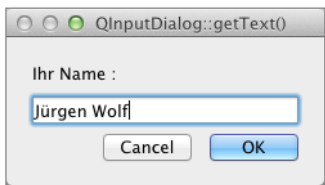


Abbildung 17.63 Einlesen einer Zeichenkette –
»QInputDialog::getText()«

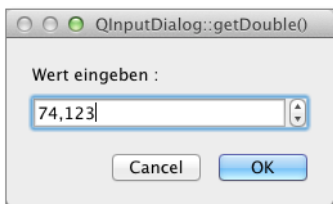


Abbildung 17.64 Einlesen eines double-Wertes –
»QInputDialog::getDouble()«

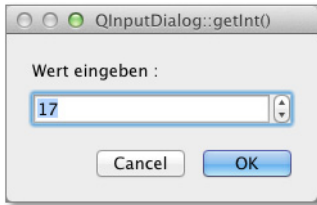


Abbildung 17.65 Einlesen eines Integer-Wertes –
»QInputDialog::getInt()«

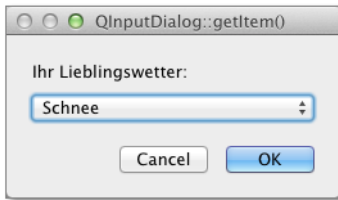


Abbildung 17.66 Einlesen eines Elements –
»QInputDialog::getItem()«

17.8.4 Weitere Dialoge

Es gibt noch einige weitere vorgefertigte Dialoge von Qt, auf die hier allerdings nicht mehr näher eingegangen wird. Dazu gehört `QFontDialog`, welcher bei der Auswahl von Schriften zum Einsatz kommt. Die Schriftarten selbst werden bei Qt mit der Klasse `QFont` verwendet. Für die Auswahl der Farbe, welche in Qt mit `QColor` verwendet wird, steht der Dialog `QColorDialog` zur Verfügung. Und auch für den Drucker, welcher in Qt mit `QPrinter` verwendet wird, steht mit `QPrintDialog` ein Druckerdialog bereit.

17.9 Eigenen Dialog mit dem Qt Designer erstellen

In der gängigen Praxis wird kaum noch einer hergehen und einen Dialog komplett manuell von Hand codieren, wie dies in Abschnitt 17.7, »Dialoge erstellen mit `QDialog`«, gezeigt wurde. Hierfür wird gerne entweder auf vorgefertigte Dialoge zurückgegriffen, worauf in Abschnitt 17.8, »Vorgefertigte Dialoge von Qt«, zuvor eingegangen wurde, oder man verwendet auch hier wieder den Qt Designer. Gerade das anspruchsvolle Layout eines Dialogs lässt sich mit dem Designer von Qt wesentlich einfacher und häufig auch schneller realisieren. In diesem Abschnitt werden Sie erfahren, wie Sie mithilfe des Qt Designers einen eigenen, komplexeren Dialog erstellen können. Dabei erfahren Sie auch gleich noch, wie Sie eine Designer-Datei in einer Qt-Anwendung verwenden können.

1. Starten Sie ein neues leeres Qt-Projekt über das Menü DATEI • NEU, wie Sie dies in Abschnitt 17.1.2, »Ein einfaches Projekt erstellen«, in den Schritten 1 bis 5 gemacht haben.
2. Klicken Sie jetzt die Projektdatei mit der rechten Maustaste an, und wählen Sie im Kontextmenü HINZUFÜGEN aus (Abbildung 17.67).

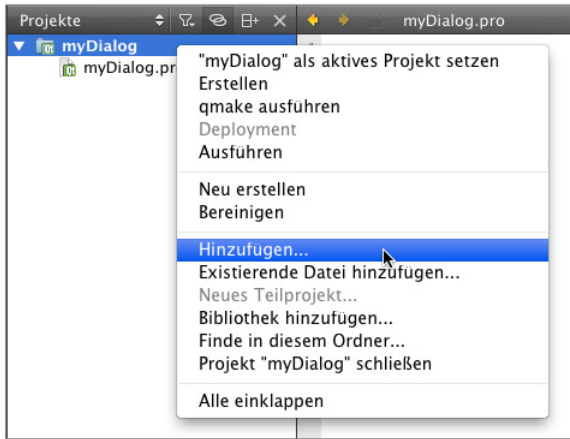


Abbildung 17.67 Dem Projekt etwas hinzufügen ...

3. Wählen Sie im nächsten Dialog unter den Vorlagen bei DATEIEN UND KLASSEN QT aus, so dass Sie in der Mitte QT-DESIGNER-FORMULARKLASSE auswählen können. Klicken Sie anschließend auf die Schaltfläche AUSWÄHLEN (siehe Abbildung 17.68).

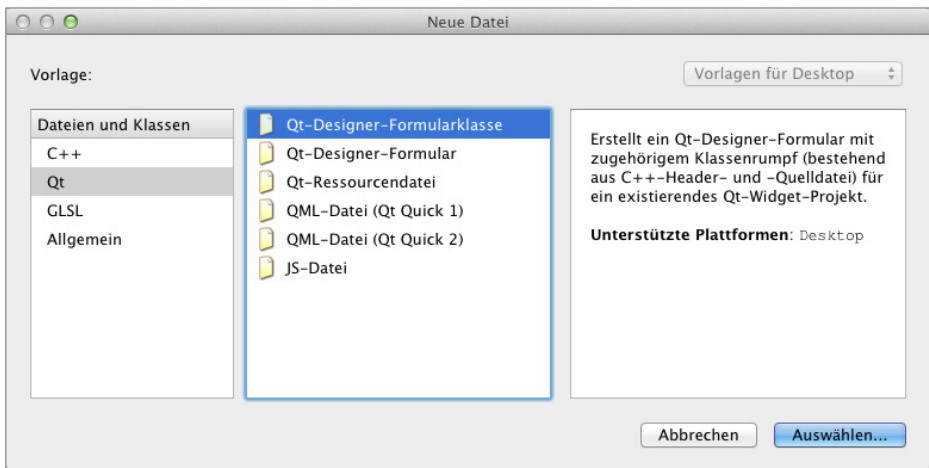


Abbildung 17.68 Eine neue Qt-Designer-Formularklasse erstellen

4. Im nächsten Schritt erscheint ein Fenster, in dem Sie auswählen können, was Sie erstellen wollen. In diesem Beispiel wollen wir zunächst eine Dialogbox ohne

Schaltflächen erstellen, die wir dann noch ausstatten werden. Wählen Sie daher **DIALOG WITHOUT BUTTONS** aus. Klicken Sie dann auf die Schaltfläche **WEITER** (siehe Abbildung 17.69).

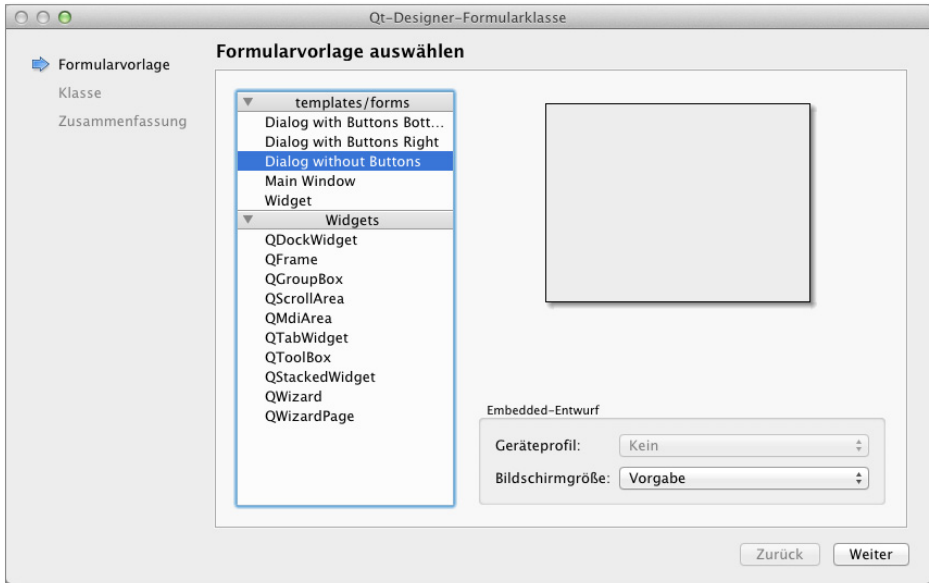


Abbildung 17.69 Eine Vorlage auswählen

- Als Nächstes werden Sie nach dem Klassennamen, den Namen der **HEADER-DATEI**, der **QUELLDATEI** und der **FORM-DATEI** sowie dem Speicherort für das künftige Dialogfenster gefragt (siehe Abbildung 17.70).

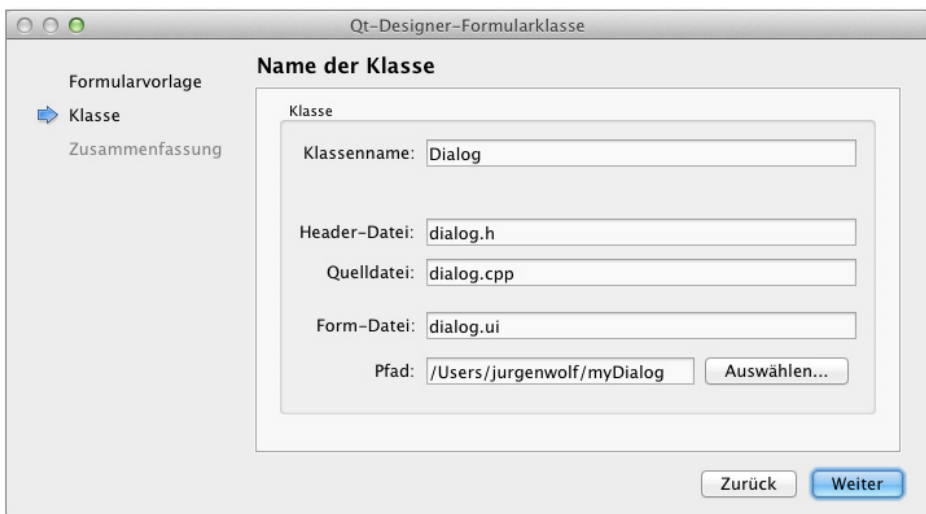


Abbildung 17.70 Hier müssen alle Daten für die künftige UI-Datei angegeben werden.

6. Am Ende erhalten Sie eine Zusammenfassung Ihrer Eingaben und können noch eine Projektverwaltung hinzufügen. Mit der Schaltfläche FERTIG sollten Sie eine Dialogbox im Modus DESIGN von Qt Creator vorfinden (siehe Abbildung 17.71).

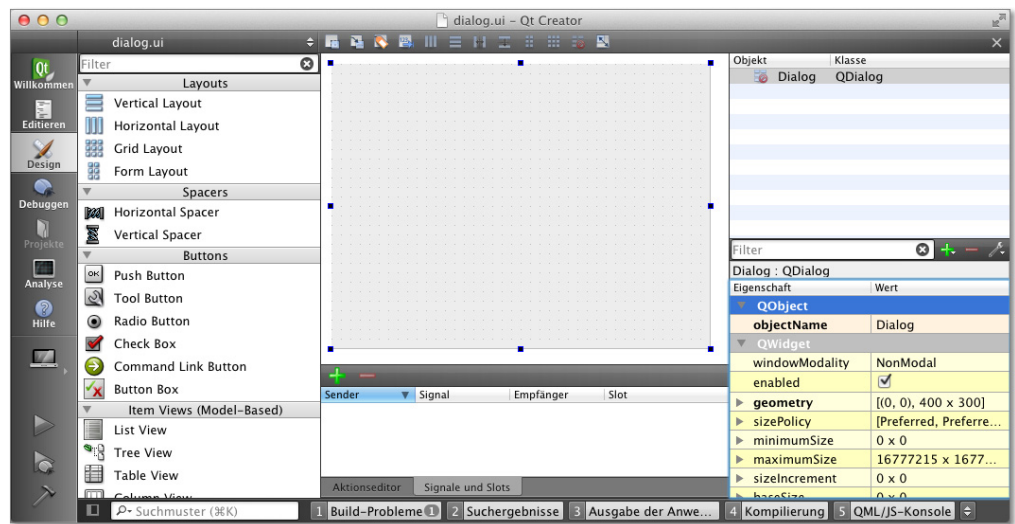


Abbildung 17.71 Eine leere Dialogbox im Design-Modus

Layout entwerfen

1. Von der linken Widget-Box ziehen Sie jetzt per Drag & Drop die gewünschten Bedienelemente (Widgets) in die Dialogbox. In unserem Fall verwenden wir zwei Labels, ein einzeliges Textfeld, eine Spin-Box, eine Group-Box, zwei Radioschaltflächen und eine normale Schaltfläche. Zusätzlich verwende ich noch zwei vertikale und einen horizontalen Spacer (siehe Abbildung 17.72).

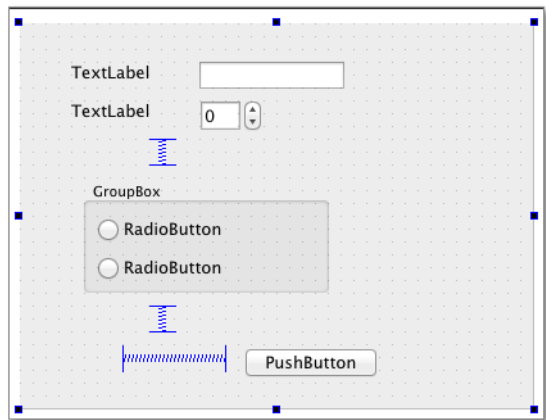


Abbildung 17.72 Ein erster grober Entwurf unserer Dialogbox

2. Zunächst ist es noch nicht wichtig, dass alles millimetergenau an seinem Platz ist. Dies erledigen wir im nächsten Arbeitsvorgang. Priorität hat zunächst das Layout für die beiden Text-Labels, das einzelne Eingabefeld und die Spin-Box. Hierzu markieren Sie mit gedrückter linker Maustaste bzw. auch mit gehaltener `[Strg]`-Taste (bzw. beim Mac `[⌘]`-Taste) diese vier Widgets (siehe Abbildung 17.73).

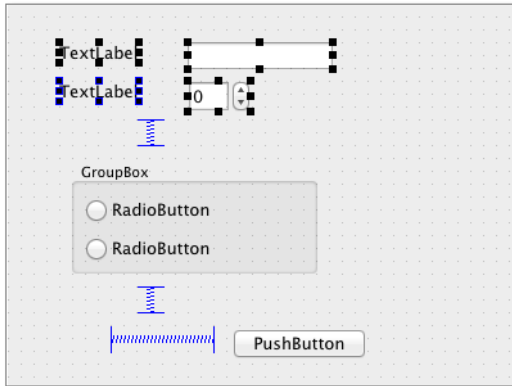


Abbildung 17.73 Widgets für das Anpassen des Layouts markieren

3. Im Beispiel bietet sich hervorragend das Tabellen-Layout (`QGridLayout`) an. Hierzu finden Sie in der Werkzeugleiste einen entsprechenden Button (siehe Abbildung 17.74), oder Sie verwenden die Tastenkombination `[Strg] + [G]` (bzw. `[⌘] + [G]` beim Mac).

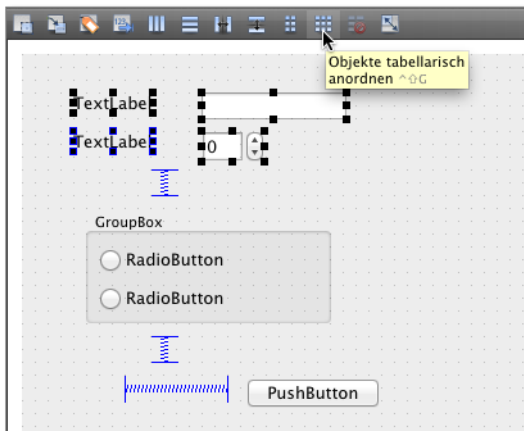


Abbildung 17.74 Das Tabellen-Layout (`QGridLayout`) auswählen

4. Wenn Sie das Tabellen-Layout aktiviert haben, erhalten Sie eine saubere Formatierung der markierten Widgets (siehe Abbildung 17.75).

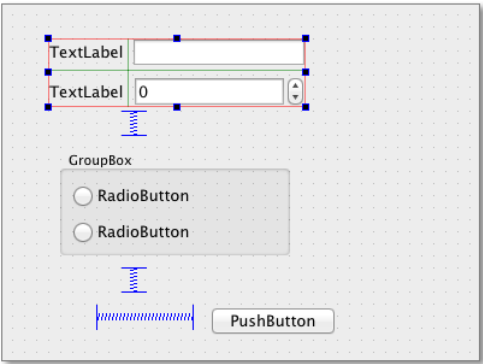


Abbildung 17.75 Saubere Formatierung dank »QGridLayout«

5. Ähnliches wollen wir jetzt auch mit der Group-Box und den Radioschaltflächen machen, nur dass wir hierbei statt des Grid-Layouts ein vertikales Layout verwenden (Tastenkombination `Strg`+`L` bzw. `Ctrl`+`L` beim Mac): also zunächst wieder die entsprechenden Widgets markieren und das vertikale Layout auswählen (siehe Abbildung 17.76 und Abbildung 17.77).

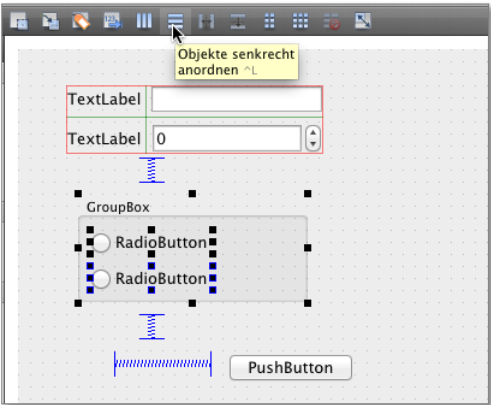


Abbildung 17.76 Group-Box und Radioschaltflächen markieren und ...

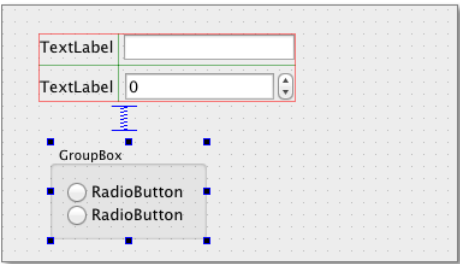


Abbildung 17.77 ... sauber senkrecht anordnen

6. Markieren Sie jetzt noch den horizontalen Spacer und die Schaltfläche, und ordnen Sie diese waagrecht an (siehe Abbildung 17.78).

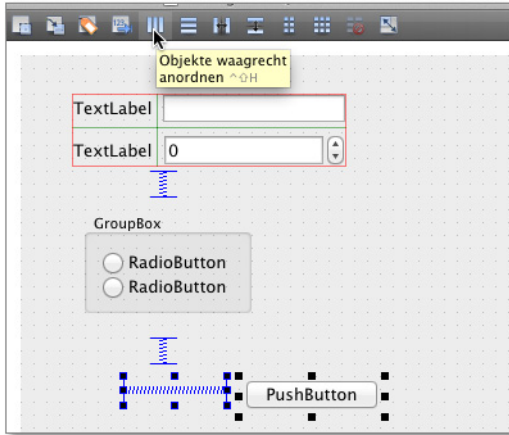


Abbildung 17.78 Der Spacer und die Schaltfläche werden waagrecht angeordnet.

Befehle über das Menü

Natürlich können Sie sämtliche Befehle des Designers auch über das Menü EXTRAS • FORMULAREEDITOR aufrufen.

7. Zum Schluss markieren Sie nochmals alle Bedienelemente (siehe Abbildung 17.79) und ordnen sie erneut senkrecht mit dem vertikalen Layout (QVBoxLayout) an (siehe Abbildung 17.80).

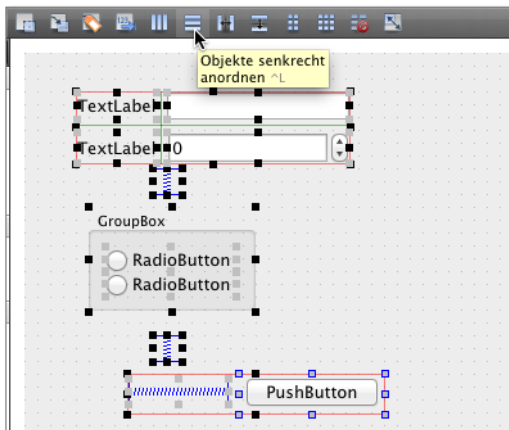


Abbildung 17.79 Zum Schluss nochmals alle Bedienelemente des Dialogs auswählen und ...

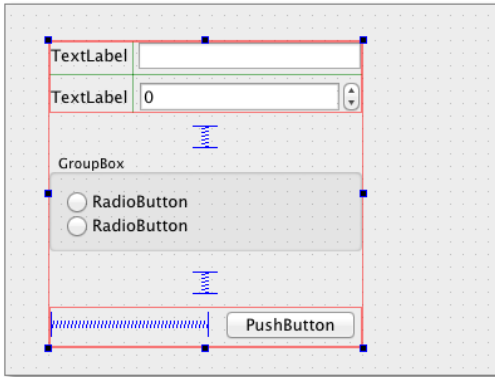


Abbildung 17.80 ... ordentlich senkrecht anordnen

Ergebnis in einer Vorschau betrachten

Jetzt können Sie noch die Größe der Dialogbox oder auch der vertikalen Layouts ein wenig anpassen und sich anschließend den Dialog über EXTRAS • FORMULAREEDITOR • VORSCHAU oder die Tastenkombination `[Alt] + [Strg] + [R]` bzw. `[Alt] + [⌘] + [R]` ansehen (siehe Abbildung 17.81).

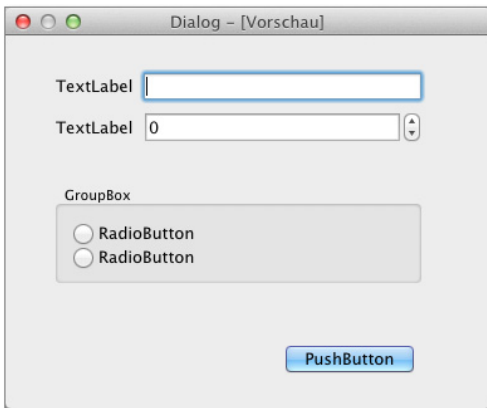


Abbildung 17.81 Vorschau der Dialogbox

Eigenschaften der Bedienelemente anpassen

Im Designer können Sie sich jetzt die Hierarchie der einzelnen Widgets im Dock-Wid-
get Objekt-Inspektor rechts oben vom Qt Designer ansehen (siehe Abbildung 17.82).

Um jetzt die Eigenschaften der Widgets zu bearbeiten, können Sie entweder auf einen entsprechenden Eintrag im Objekt-Inspektor klicken oder aber auch direkt im Design-Modus der Dialogbox auf ein entsprechendes Widget. Für das Anpassen eines Widgets und dessen Eigenschaften gibt es dann den Eigenschaft-Editor. Klicken Sie bspw. im

Objekt-Inspektor auf QDialog oder auf den Fensterrahmen, finden Sie alle möglichen Eigenschaften des Widgets im Eigenschaft-Editor wieder (siehe Abbildung 17.83).

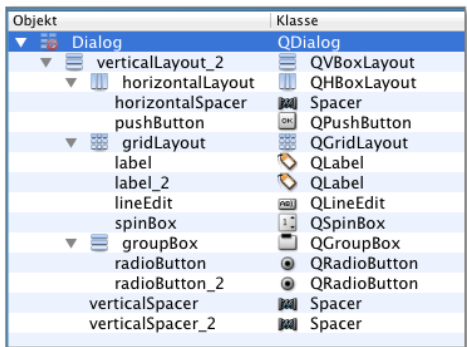


Abbildung 17.82 Rechts oben bietet der Objekt-Inspektor einen sehr guten Überblick zur verwendeten Hierarchie.

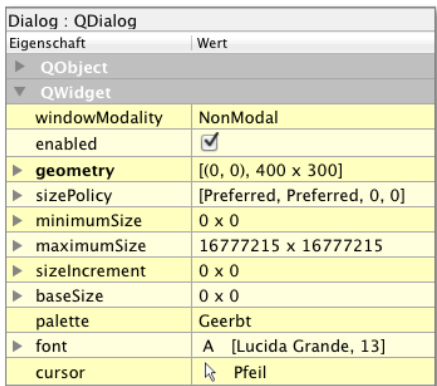


Abbildung 17.83 Im Eigenschaft-Editor können Sie die Eigenschaften der Widgets betrachten und modifizieren.

In diesem Eigenschaft-Editor finden Sie zwei Spalten. Auf der linken Seite befindet sich der Name der EIGENSCHAFT und auf der rechten Seite der WERT. Der WERT ist es, den Sie ändern können. Wollen Sie bspw. einen modalen Dialog verwenden, müssen Sie einfach in der zweiten Spalte von »windowModality« klicken, und es erscheint eine Liste mit den möglichen Werten für diese Eigenschaft, welche Sie verwenden können (siehe Abbildung 17.84).

Um den Fenstertitel zu ändern, müssen Sie einfach nach der Eigenschaft »windowTitle« suchen und in der Spalte WERT den entsprechenden Titel eintragen (siehe Abbildung 17.85).

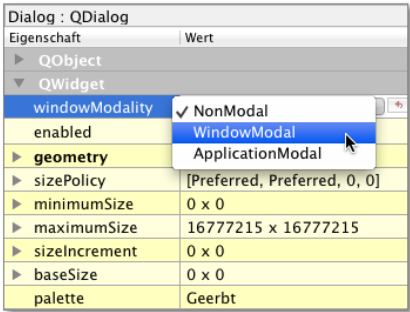


Abbildung 17.84 Hier können Sie den Dialog modal machen.

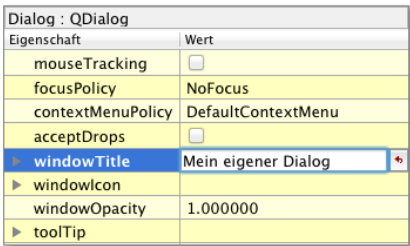


Abbildung 17.85 Einen Fenstertitel für den Dialog vergeben

So können Sie praktisch alle Beschriftungen und Eigenschaften eines Widgets anpassen. Natürlich wird so auch das Setzen der Werte, wie bspw. für die Spin-Box, durchgeführt (siehe Abbildung 17.86).

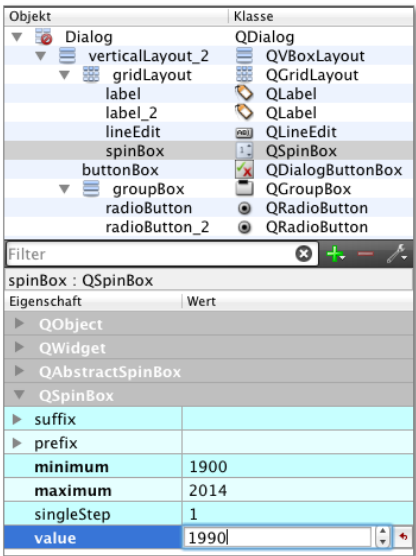


Abbildung 17.86 Eigenschaften der Spin-Box modifizieren

Dies wirkt sich direkt auf die Dialogbox aus (siehe Abbildung 17.87).



Abbildung 17.87 Nach einigen Modifikationen im Eigenschaft-Editor

Nach einigen weiteren Anpassungen mit dem Eigenschaft-Editor der einzelnen Widgets sieht unsere Dialogbox in der Vorschau nun aus wie in Abbildung 17.88.



Abbildung 17.88 Die zusammengeclickte Dialogbox

Wenn man ein wenig Übung mit dem Qt Designer hat, lässt sich ein solcher Dialog in zwei bis drei Minuten zusammenklicken. Mit reinem C++-Code wäre man da schon erheblich länger beschäftigt, besonders was das Layout betrifft.

Experimentieren erwünscht

Die Verwendung des Eigenschaft-Editors lädt geradezu zum Herumexperimentieren ein, was Sie auf keinen Fall versäumen, sondern unbedingt tun sollten.

Signal-Slot-Verbindung(en) hinzufügen

Auch für die Signal-Slot-Verbindungen sieht der Designer etwas Besonderes vor. Im Menü BEARBEITEN finden Sie einen Eintrag SIGNALE UND SLOTS BEARBEITEN, den Sie auch mit **[F4]** (beim Mac häufig auch **[Fn] + [F4]**) aktivieren können. Zurück kommen Sie im selben Menü wieder mit WIDGETS BEARBEITEN oder **[F3]** (bzw. **[Fn] + [F3]** beim Mac).

Aktivieren Sie also das Bearbeiten einer Signal-Slot-Verbindung mit **[F4]** bzw. **[Fn] + [F4]**. Jetzt wählen Sie hier das Widget für das Signal aus und ziehen es per Drag & Drop auf das Widget mit dem Slot. Im Beispiel wollen wir auf ein Signal von `QPushButton` reagieren, und den Slot stellt der Dialog (`QDialog`) zur Verfügung (siehe Abbildung 17.89).

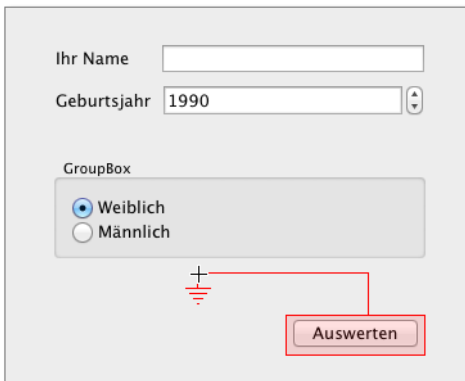


Abbildung 17.89 Eine Signal-Slot-Verbindung per Drag & Drop

Die Verbindung wird vom Designer angezeigt, angefangen beim Widget, welches das Signal auslöst, hin zum Widget, welches den Slot enthält. Für das Ziel-Widget wird eine Art Erdungssymbol angezeigt. Als Nächstes erscheint ein Dialog, in dem Sie sich die passende Signal-Slot-Verbindung aussuchen können (siehe Abbildung 17.90).

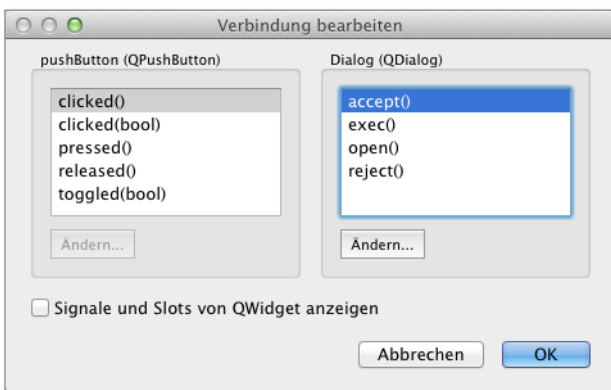


Abbildung 17.90 Signal-Slot-Verbindung auswählen

Im Dialog werden allerdings nur die gängigsten Signal-Slot-Verbindungen angezeigt. Benötigen Sie alle vorhandenen Verbindungen, müssen Sie nur die Checkbox **SIGNALE UND SLOTS VON QWIDGET ANZEIGEN** aktivieren. Im Beispiel wurde für `QPushButton` das Signal `clicked()` eingerichtet, welches im Fall eines Auftretens mit dem Slot `accept()` von `QDialog` reagiert. Haben Sie die neue Signal-Slot-Verbindung eingerichtet, wird dies beim Dialog im Designer auch angezeigt (solange der Modus **SIGNALE UND SLOTS BEARBEITEN** ist) (siehe Abbildung 17.91).

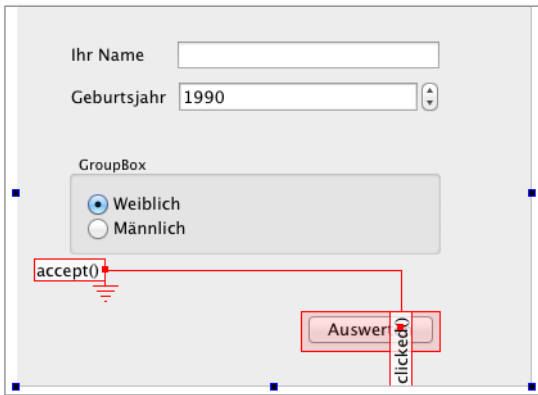


Abbildung 17.91 Virtuelle Übersicht der Signal-Slot-Verbindung

Neben dem eben gezeigten Weg können Sie auch über den Editor **SIGNALE UND SLOTS** unterhalb des Qt Creators die Signale-Slot-Verbindungen einrichten (Abbildung 17.92).



Abbildung 17.92 Signale-und-Slots-Editor

Ebenso können Sie eine Signal-Slot-Verbindung auch nur mit dem Editor **SIGNALE UND SLOTS** einrichten bzw. vorhandene Verbindungen ändern oder löschen. Mit dem roten Minuszeichen entfernen Sie eine Signal-Slot-Verbindung. Mit dem Pluszeichen können Sie eine neue Verbindung einrichten. Sie müssen dann die Werte der einzelnen Spalten des Senders mit Signal und dem Empfänger mit Slot aus einer Liste auswählen (siehe Abbildung 17.93).

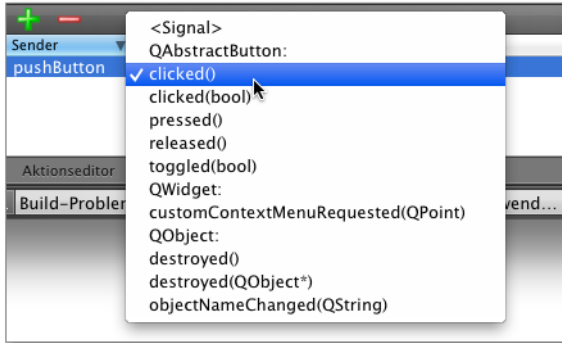


Abbildung 17.93 Signal-Slot-Verbindungen über den Signale-und-Slots-Editor ändern bzw. anpassen

Tabulatorenreihenfolge anpassen

Ein weiteres Tool vom Designer ist die Überprüfung der Tab-Reihenfolge. Viele Anwender verwenden bei Dialogen gerne die Tabulatortaste , um in das nächste Widget zu wechseln. Gerade wenn man mit dem Designer sein Layout zusammenklickt und das eine oder andere Widget hinzufügt, stimmt die Reihenfolge oft nicht mehr. Den Überblick verschaffen Sie sich mit dem Menü BEARBEITEN • TABULATORENREIHENFOLGE BEARBEITEN (siehe Abbildung 17.94).

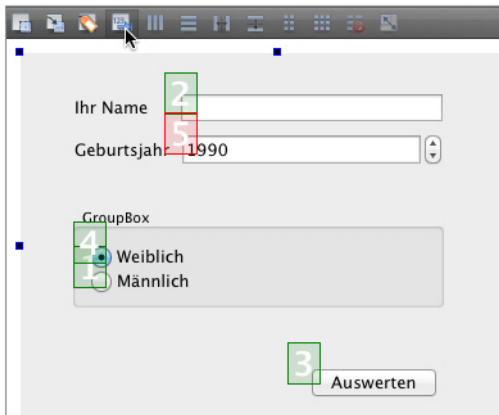


Abbildung 17.94 Die Tabulatorenreihenfolge passt so vermutlich nicht.

Die Tabulatorenreihenfolge in Abbildung 17.94 dürfte wohl nicht das sein, was der Programmierer wollte. Zuerst wird der Radiobutton MÄNNLICH angesprungen. Nach einem  geht es zu IHR NAME und einem weiteren  zur Schaltfläche AUSWERTEN usw.

Um jetzt die gewünschte Reihenfolge einzurichten, klicken Sie einfach so lange auf die Zahlen, bis der gewünschte Wert darin erscheint. Den Vorgang wiederholen Sie

mit den anderen Widgets so lange, bis Sie die gewünschte Reihenfolge erhalten (siehe Abbildung 17.95).



Abbildung 17.95 Jetzt passt die Tabulatorenreihenfolge.

Buddies bearbeiten

Auch die Tastaturkürzel und Buddies können Sie über das Menü BEARBEITEN • BUDDIES BEARBEITEN beim Designer einrichten. Hierbei geht man ähnlich wie beim Einrichten von Signal-Slot-Verbindungen vor. Wählen Sie in diesem Beispiel das Label, das Sie verwenden wollen, und ziehen Sie dieses per Drag & Drop auf das zugehörige Widget (Buddy) (siehe Abbildung 17.96).



Abbildung 17.96 Einzeiliges Eingabefeld als Buddy vom Label »Ihr Name«

Jetzt müssen Sie noch mit dem Eigenschaft-Editor vor dem Buchstaben (bei der Eigenschaft »text«), mit dem Sie auf das Zeilenfeld springen wollen, ein Ampersandzeichen (&) setzen, wodurch dieser Buchstabe dann im Label unterstrichen ist. Mit »Ihr &Name« wird bspw. der Buchstabe »N« unterstrichen, und Sie können mit der Tastenkombination **Alt** + **N** direkt auf das Zeilenfeld springen (Abbildung 17.97).

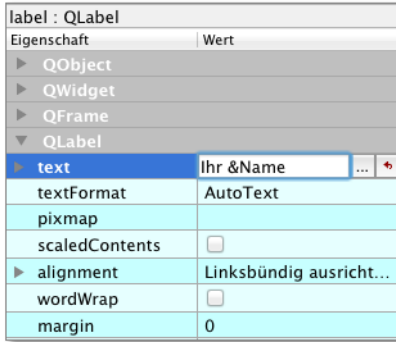


Abbildung 17.97 Mit »&« vor »Name« kann künftig per Tastatur direkt auf das entsprechende Editierfeld gesprungen werden.

Erstellten Dialog in einer Qt-Anwendung verwenden

Dass Sie das komplette Projekt abgespeichert haben, versteht sich von selbst. Wenn Sie mit dem Modus DESIGN fertig sind, können Sie wieder in den Modus EDITIEREN wechseln, wo Sie in der Projektverwaltung alle nötigen Dateien vorfinden. Unser Hauptaugenmerk liegt hierbei auf der Formdatei mit der Endung *.ui*. Im Beispiel wurde diese Datei unter dem Namen *dialog.ui* im Projektverzeichnis gespeichert. Diese Datei können Sie sich auch im Modus EDITIEREN ansehen. Dabei werden Sie feststellen, dass es sich hierbei um nichts anderes als um eine Datei im XML-Format handelt. In der Projektdatei (*.pro*) wird diese Datei wie folgt hinzugefügt:

```
FORMS += \
    dialog.ui
```

Weitere Designer-Dateien (*.ui*) können Sie getrennt durch eine Leerzeile hinten anfügen. Wenn Sie *qmake* ausführen und das Projekt erstellen (oder einfach ALLES ERSTELLEN) wird diese Form-Datei von einem UI-Compiler (*User Interface Compiler*, kurz UIC) verwendet. Dieser speziell für UI-Dateien benötigte Compiler erstellt eine entsprechende Headerdatei mit dem entsprechenden Code. Der Name dieser Datei hängt vom Namen der gespeicherten UI-Datei ab. In unserem Fall wird aus der UI-Datei *dialog.ui* die Headerdatei mit dem Namen *ui_dialog.h* hinter den Kulissen vom UI-Compiler erstellt. Also immer wird aus *NAME.ui* der Headername *ui_NAME.h* generiert.

Logischerweise müssen Sie im Quellcode, wo Sie den Dialog verwenden wollen, die entsprechende Headerdatei mit `#include` (hier *ui_dialog.h*) hinzufügen.

Wenn Sie den Anweisungen im Buch gefolgt sind, dann sind alle Dateien schon für Sie vorbereitet. Das Grundgerüst der Klasse finden Sie in unserem Beispiel in *dialog.h* und den Code, der das GUI erzeugt, in *dialog.cpp*. Wenn Sie sich die Datei ansehen,

finden Sie dort mit `setupUi()` eine Methode, welche den kompletten Widget-Baum (also das komplette GUI) für das Eltern-Widget aus der Form-Datei erzeugt.

Wo ist `ui_name.h`?

Die Datei, die vom UI-Compiler aus der Form-Datei erzeugt wurde, wird nicht zum Projekt bzw. zur *.pro*-Datei hinzugefügt, befindet sich aber im Build-Verzeichnis des Projekts.

Der Klassenname (hier `Dialog`) wird aus dem »objectName«-Attribut des Dialogs gesetzt, welchen Sie im Designer verwendet haben (siehe Abbildung 17.98).

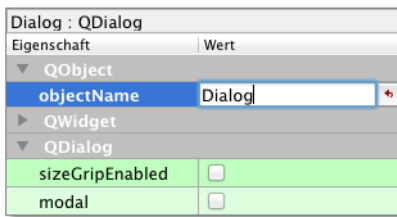


Abbildung 17.98 »objectName«-Attribut der Klasse »QObject« legt den Klassennamen fest.

Sollten Sie den Namen nachträglich löschen oder ändern, würden Sie die vom Designer erstellte Headerdatei nicht verwenden können, weil Sie hiermit eine anonyme Klasse ohne Namen erzeugt hätten. Der UI-Compiler fügt dem Klassennamen dann noch ein `Ui_` vorne an, so dass aus dem Objektnamen `Dialog` im Beispiel die Klasse `Ui_Dialog` wird.

Um mit einem Bezeichner auf die vom Designer verwendeten GUI-Elemente (Widgets) zugreifen zu können, wird ein eigener Namensraum `Ui` generiert und die Klasse `Dialog` öffentlich von `Ui_Dialog` abgeleitet. In der Praxis bedeutet das, dass Sie folgendermaßen ein Objekt des vom Designer entworfenen Dialogs erzeugen können:

```
Ui::Dialog myUi;
```

Die vom Designer generierten Klassen sind somit immer über `Ui::Klassenname` verfügbar.

Jetzt wollen wir die vom Designer erzeugte Klasse in einem einfachen Programm verwenden. Zunächst das Grundgerüst:

```
00 // kapitel17/uiexample/form.h
01 #ifndef FORM_H
02 #define FORM_H
03 #include <QtGui>
04 #include <QDialog>
```

```

05 #include <QMessageBox>
06 #include "ui_dialog.h"

07 class myDialog : public QDialog {
08     Q_OBJECT
09 public:
10     myDialog();
11     ~myDialog() {}
12 private:
13     Ui::Dialog ui;
14 private slots:
15     void analyseDialog();
16 };
17 #endif

```

Hier finden Sie in Zeile **13** die vom UI-Compiler generierte Klasse als privates Objekt. Natürlich darf die entsprechende Headerdatei (Zeile **06**) nicht fehlen, in der diese Klasse auch definiert ist. Jetzt die Implementierung des Codes:

```

00 // kapitel17/uiexample/form.cpp
01 #include "form.h"

02 myDialog::myDialog(){
03     ui.setupUi(this);
04     connect( ui.pushButton, SIGNAL( clicked() ),
              this, SLOT(analyseDialog()) );
05 }

06 void myDialog::analyseDialog() {
07     QString str;
08     if( ui.lineEdit->text().isEmpty())
09         str.append("Name: keine Angaben\n");
10     else {
11         str.append("Name: ");
12         str.append(ui.lineEdit->text());
13         str.append("\n");
14     }
15     if( ui.spinBox->cleanText().isEmpty())
16         str.append("Geburtsdatum: keine Angaben\n");
17     else {
18         str.append("Geburtsdatum: ");
19         str.append(ui.spinBox->cleanText());
20         str.append("\n");

```

```

21     }
22     if( ui.radioButton->isChecked())
23         str.append("Geschlecht: weiblich\n");
24     else
25         str.append("Geschlecht: männlich\n");
26     QMessageBox::information(0,tr("Die Auswertung"), str);
27 }

```

Das Wichtigste und Erste, das wir im Konstruktor in Zeile 03 erledigen, ist es, `setupUi()` aufzurufen, um überhaupt die UI-Klasse verwenden zu können. Sollten Sie `setupUi()` zu spät aufrufen und das Programm greift bereits auf eine Mitgliedsvariable der vom Designer generierten Oberfläche zurück, stürzt das Programm ab, weil auf einen undefinierten Speicherbereich zugegriffen wird.

In Zeile 04 richten wir dann einen Signal-Slot-Handler ein, der auf das Signal `clicked()` von `ui.pushButton` mit dem Slot `analyseDialog()` reagiert. Der Slot, der unsere Eingabe im Dialog auswertet, wurde in den Zeilen 06 bis 27 definiert.

Zugriff auf die Bezeichner der Form-Datei

Auch hier gilt für den Zugriff auf die einzelnen Bezeichner im Designer, dass diese aus der Eigenschaft `objectName` generiert werden. Ein Blick in `ui_DATEiname.h` oder den Objekt-Inspektor im Modus DESIGN hilft aus, falls man die Bezeichner der einzelnen Widgets nicht mehr genau weiß oder man es dem Designer überlassen hat, wie diese lauten.

Jetzt fehlt uns nur noch eine Hauptfunktion, um unser GUI auszuführen:

```

00 // kapitel17/uiexample/main.cpp
01 #include <QApplication>
02 #include "form.h"

03 int main(int argc, char *argv[]) {
04     QApplication app(argc, argv);
05     myDialog dlg;
06     dlg.show();
07     return app.exec();
08 }

```

In Abbildung 17.99 finden Sie nochmals einen kurzen Überblick über alle Dateien in unserem Projekt. Wenn Sie auf der Suche nach `ui_dialog.h` sind, werden Sie gewöhnlich im Build-Verzeichnis fündig, wo sich auch die ausführbare Anwendung befindet.

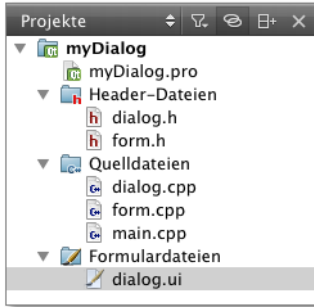


Abbildung 17.99 Alle Dateien des Projekts im Überblick

Fehlende Bibliotheken in der .pro-Datei

Und auch hier erneut zur Erinnerung: Bei Erstellen eines Qt-Projekts kann es passieren, dass qmake die nötigen Qt-Module nicht hinzulinkt. In unserem Beispiel würden hierzu die Module *Qt Core*, *Qt GUI* und *Qt Widgets* benötigt. Sollten Sie Probleme mit dem Übersetzen haben, brauchen Sie lediglich in der .pro-Datei folgende Zeile einzufügen:

```
QT += core gui widgets
```

Abbildung 17.100 und Abbildung 17.101 zeigen das Programm bei der Ausführung.



Abbildung 17.100 Der Dialog bei der Ausführung



Abbildung 17.101 Die Auswertung des Dialogs

Auf UI-Elemente ohne Hilfsvariable zugreifen

Neben der eben gezeigten Möglichkeit, können Sie die vom Designer generierte Klasse auch durch eine Mehrfachvererbung in einer eigenen Subklasse verwenden. Der Vorteil ist dabei, dass Sie die Widgets der vom UI-Compiler generierten Klasse ohne Hilfsvariable (`Ui::Klassenname`) und ohne Umwege verwenden können. Hierzu das Grundgerüst:

```
00 // kapitel17/uiexample2/form.h
01 #ifndef FORM_H
02 #define FORM_H
03 #include <QtGui>
04 #include <QDialog>
05 #include <QMessageBox>
06 #include "ui_Dialog.h"

07 class myDialog : public QDialog, private Ui::Dialog {
08     Q_OBJECT
09 public:
10     myDialog();
11     ~myDialog() {}
12 private slots:
13     void analyseDialog();
14 };
15 #endif
```

Hier kommt in Zeile **07** am Ende eine private Vererbung unserer Designerklasse in Aktion. Dadurch, dass die Klasse `privat` vererbt wird, bleibt auch der OOP-Ansatz mit seiner Verkapselung erhalten. Jetzt können Sie innerhalb der Klasse `myDialog` ohne Umwege auf die Mitglieder der Designerklasse zugreifen. Hier die Implementierung:

```
00 // kapitel17/uiexample2/form.cpp
01 #include "form.h"

02 myDialog::myDialog(){
03     setupUi(this);
04     connect( pushButton, SIGNAL( clicked() ),
              this, SLOT(analyseDialog()) );
05 }

06 void myDialog::analyseDialog() {
07     QString str;
08     if( lineEdit->text().isEmpty())
09         str.append("Name: keine Angaben\n");
```

```

10     else {
11         str.append("Name: ");
12         str.append(lineEdit->text());
13         str.append("\n");
14     }
...

```

Reihenfolge beachten

Damit diese Mehrfachvererbung hier auch funktioniert und keine Probleme mit dem Meta-Compiler auftreten, müssen Sie die Reihenfolge einhalten. Zuerst muss hier die Klasse von `QDialog` und dann von der Designerklasse erben.

17.10 Grafische Bedienelemente von Qt (Qt-Widgets)

Nachdem jetzt die Dialoge von Qt behandelt wurden, können Sie mit den grundlegenden Qt-Widgets fortfahren. Um den Rahmen des Kapitels nicht zu sprengen, werden hier nur die gängigeren Widgets kurz vorgestellt. Für mehr Details (und mehr Widgets) sei wie immer der Assistent von Qt empfohlen.

Quellcode auf der Buch-CD

Die Listings in diesem Beispiel sind häufig extrem gekürzt und zeigen nur den Ausschnitt, der nötig ist, um die Bedienelemente zu beschreiben. Den vollen Quellcode zu den Beispielen finden Sie auf der Buch-CD.

17.10.1 Schaltflächen (Basisklasse `QAbstractButton`)

Die Basisklasse `QAbstractButton` (ableitet von `QWidget`) ist die abstrakte Basisklasse der Widgets für Schaltflächen (Buttons) und stellt die grundlegenden Methoden, Signale und Slots für alle anderen Schaltflächenarten zur Verfügung (Abbildung 17.102).

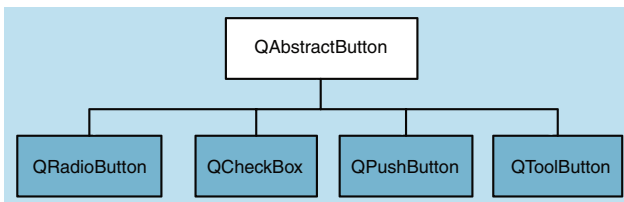


Abbildung 17.102 Basisklasse »`QAbstractButton`«

`QAbstractButton` unterstützt neben den gewöhnlichen Schaltflächen (`QPushButton`) auch Auswahlkästchen (`QCheckBox`) und Radioschaltflächen (`QRadioButton`). Auswahlkästchen und Radioschaltflächen werden gewöhnlich zu einer Gruppe zusammengefasst, innerhalb der sie ausgewählt werden können. Während bei den Auswahlkästchen mehrere Auswahlen in einer Gruppe erlaubt sind, wird bei den Radioschaltflächen in der Regel nur ein Element in der Gruppe ausgewählt.

Zu den gewöhnlichen Schaltflächen (`QPushButton`) gehört auch die Klasse `QToolButton`. Beide unterstützen auch den niedergedrückten bzw. angewählten Zustand (engl. *toggled*).

`QAbstractButton` vergibt den Text und das Icon an die Schaltflächen. Der Text wird mit `setText()` gesetzt (es sei denn, dieser wird beim Erzeugen mit dem Konstruktor schon vergeben) und das Icon mit `setIcon()`. Beispielsweise lässt sich folgendermaßen ein Icon auf die Schaltfläche setzen:

```
button->setIcon(QIcon("images/icon.png"));
```

In diesem Beispiel wird das Icon *icon.png*, welches sich im Verzeichnis *images* des laufenden Programms befindet, der Schaltfläche hinzugefügt. Ist das Icon größer als die Schaltfläche, wird die Größe automatisch angepasst. Alle Schaltflächen in Qt unterstützen sowohl Text als auch die Anzeige von Icons (Abbildung 17.103).

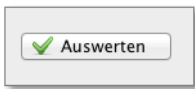


Abbildung 17.103 Ein Icon wurde der Schaltfläche hinzugefügt.

Wollen Sie dem Button ein Tastaturkürzel zuweisen, so stehen Ihnen gleich zwei Möglichkeiten zur Verfügung. Entweder verwenden Sie das Ampersandzeichen (&), und `QAbstractButton` erzeugt automatisch ein Tastaturkürzel für das Zeichen vor dem Ampersandzeichen, bspw.:

```
button = new QPushButton ("A&uswerten");
```

Hiermit können Sie mit dem Tastaturkürzel `[Alt]+[U]` die Schaltfläche auswählen. Dadurch wird auch gleich der Slot `animateClick()` ausgeführt. Oder Sie können ein solches Tastaturkürzel mit der Methode `setShortcut()` setzen, bspw.:

```
button->setShortcut(tr("Alt+M"));
```

Damit kann die Schaltfläche mit der Tastenkombination `[Alt]+[M]` betätigt (`clicked()`) werden.

Weitere Methoden

QAbstractButton definiert noch weitere gängige Methoden, welche somit auch für QPushButton, QRadioButton, QCheckBox und QToolButton zur Verfügung stehen. Schlagen Sie hierzu bitte bei Bedarf im Assistant von Qt nach.

Da QAbstractButton die gängigen Signale und Slots für alle darunterliegenden Q*Button-Klassen definiert, sollen hier zumindest die gängigen Signale in Tabelle 17.4 kurz aufgelistet werden.

Signal	Beschreibung
void clicked (bool checked = false);	Dieses Signal wird ausgelöst, wenn der Button betätigt wurde (bspw. niedergedrückt und losgelassen, wenn der Mauscursor innerhalb des Buttons war).
void pressed ();	Dieses Signal wird ausgelöst, wenn der Button niedergedrückt wurde.
void released ();	Dieses Signal wird erst ausgelöst, wenn der Button losgelassen wurden.
void toggled (bool checked);	Das Signal wird ausgelöst, wenn ein ankreuzbarer oder dauerhaft niederdrückbarer Button seinen Zustand verändert. checked ist true, wenn der Button angekreuzt oder niedergedrückt ist. Das Gegenteil bei false.

Tabelle 17.4 Signale von »QAbstractButton«

Einfache Schaltflächen mit QPushButton

Ein Beispiel zu QPushButton können wir uns an dieser Stelle sparen, weil Sie davon schon mehr als genug gesehen haben und in fast jedem weiteren Beispiel auch noch verwendet werden. Die grundlegende Verwendung ist meistens, auf das Signal clicked() zu reagieren, um irgendwelche Aktionen, wie bspw. eine Auswertung, zu starten.

Ebenso kann mit der Methode setCheckable() auch eine auswählbare Schaltfläche verwendet werden, welche Sie durch Anklicken niederdrücken und erneutes Anklicken wieder loslösen können – ähnlich wie ein Auswahlkästchen mit QCheckBox (Abbildung 17.104). Wenn sich eine solche auswählbare Schaltfläche geändert hat, kann man mit dem Signal toggled(bool) darauf reagieren und mit einem entsprechenden Slot überprüfen, ob die Schaltfläche jetzt niedergedrückt (true) ist oder nicht (false). Folgender Codeausschnitt soll diese dauerhaft auswählbare Schaltfläche kurz demonstrieren:

```

00 // kapitel17/mypushbuttondialog/mypushbuttondialog.cpp
...
01 button01 = new QPushButton("Linux");
...
02 button01->setCheckable(true);
03 button01->setChecked(true);
...
04 connect( button01, SIGNAL( toggled(bool) ),
           this, SLOT( checktoggled(bool) ) );
...
05 void MyQPushDialog::checktoggled(bool iftoggled) {
06     if( iftoggled )
07         // Schaltfläche ist niedergedrückt
08     else
09         // Schaltfläche ist nicht niedergedrückt
10 }
...

```

Um einer niedergedrückten Schaltfläche die Eigenschaft zu geben, setzen wir diese Eigenschaft (Zeile **02**) mit der Methode `setCheckable(true)`. Standardmäßig ist eine solche Schaltfläche nicht niedergedrückt. Mit der Methode `setChecked(true)` ist diese Schaltfläche jetzt auch beim Programmstart gleich niedergedrückt. In Zeile **04** richten wir eine Signal-Slot-Verbindung dazu ein, welche bei einer Änderung das Signal `toggled(bool)` erhält und den übergebenen Parameter (`true` oder `false`) an den Slot `checktoggled(bool)` übergibt. Diesen Slot haben wir in den Zeilen **05** bis **10** definiert.

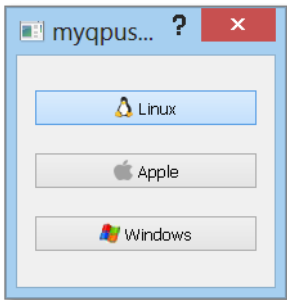


Abbildung 17.104 Eine auswählbare Schaltfläche, welche durch Anklicken jederzeit aktiviert und wieder deaktiviert werden kann

Auswahlkästchen mit `QCheckBox`

Die Klasse `QCheckBox` stellt ein Auswahlkästchen dar, welches mit einem Häkchen oder einem Kreuz aktiviert oder deaktiviert werden kann. Diese werden gerne eingesetzt, um bestimmte Optionen zu (de-)aktivieren.

Wird der Zustand eines Auswahlkästchens verändert, wird das Signal `stateChanged` (`int`) ausgelöst. Ansonsten enthält `QCheckBox` keine eigenen Signale und Slots. Allerdings kann auch hierbei bei Bedarf wieder auf die Signale (und Slots) von `QAbstractButton` zurückgegriffen werden. Die gängigsten Methoden von `QCheckBox` finden Sie in Tabelle 17.5 aufgelistet.

Methode	Beschreibung
<code>Qt::CheckState checkState() const;</code>	Gibt den Zustand des Auswahlkästchens zurück (siehe Tabelle 17.6).
<code>bool isTristate() const;</code>	Überprüft, ob das Auswahlkästchen drei Zustände erlaubt (siehe <code>setTristate()</code>).
<code>void setCheckState(Qt::CheckState state);</code>	Setzt das Auswahlkästchen auf den Zustand <code>state</code> (siehe Tabelle 17.6).
<code>void setTristate(bool y = true);</code>	Damit ist ein dritter Zustand für das Auswahlkästchen möglich, in dem Sie diesen in einen ausgegrauten Zustand setzen. Diesen ausgegrauten Zustand können Sie entweder aktivieren oder eben nicht. Dies soll anzeigen, dass dieser Check-Button nicht mehr zu ändern ist.

Tabelle 17.5 Methoden von `QCheckBox`

Die möglichen Werte für `Qt::CheckState` zeigt Tabelle 17.6 auf.

Konstante	Beschreibung
<code>Qt::Unchecked</code>	Das Element ist deaktiviert (nicht angekreuzt bzw. abgehakt).
<code>Qt::PartiallyChecked</code>	Das Element ist teilweise aktiviert.
<code>Qt::Checked</code>	Das Element ist aktiviert (angekreuzt bzw. abgehakt).

Tabelle 17.6 Zustände eines Check-Buttons

Hierzu folgt nun ein einfaches Listing, welches `QCheckBox` in der Praxis demonstrieren soll. Abgedruckt wird hier nur eine gekürzte Form (ohne das Layout) des Dialogs, der den Code rund um die Auswahlkästchen enthält. Den kompletten Quellcode mitsamt Headerdatei und `main`-Funktion finden Sie auf der Buch-CD.

```
00 // kapitel17/qcheckboxbutton/mydialog.cpp
...

```

```

01 MyQCheckBoxDialog::MyQCheckBoxDialog() {
...
02   check01 = new QCheckBox("Label 1");
03   check02 = new QCheckBox("Label 2");
04   check03 = new QCheckBox("Label 3");
05   button01 = new QPushButton("Auswerten");

06   check02->setCheckState(Qt::Checked);
07   check03->setTristate();
08   check03->setCheckState(Qt::PartiallyChecked);
...
09   connect( check01, SIGNAL( stateChanged ( int ) ),
           this, SLOT( isChecked( int ) ) );
10   connect( check02, SIGNAL( stateChanged ( int ) ),
           this, SLOT( isChecked( int ) ) );
11   connect( check03, SIGNAL( clicked() ),
           this, SLOT( cantChecked( ) ) );
12   connect( button01, SIGNAL( clicked() ),
           this, SLOT( myResult( ) ) );
13 }

14 // Zustand eines Buttons hat sich verändert
15 void MyQCheckBoxDialog::isChecked( int state ) {
16     if( state == Qt::Checked )
17         QMessageBox::information(this, "isChecked",
            "Eine Checkbox wurde aktiviert",QMessageBox::Ok);
18     else if( state == Qt::Unchecked )
19         QMessageBox::information(this, "isChecked",
            "Eine Checkbox wurde deaktiviert",
            QMessageBox::Ok);
20 }

21 // Zustand eines Buttons hat sich verändert
22 void MyQCheckBoxDialog::cantChecked( ) {
23     QMessageBox::warning(this, "Zugriff verweigert",
        "Sie können diese Checkbox nicht ändern",
        QMessageBox::Ok) ;
24   check03->setCheckState(Qt::PartiallyChecked);
25 }

26 // Alle Checkboxes auswerten
27 void MyQCheckBoxDialog::myResult() {
28     QString score = "Aktiv waren: ";

```

```

29     if (check01->checkState() == Qt::Checked )
30         score.append(" [Label 1] ");
31     if (check02->checkState() == Qt::Checked )
32         score.append(" [Label 2] ");
33     score.append(" [Label 3] "); // immer aktiv
34     QMessageBox::information(
35         this, "Auswertung", score, QMessageBox::Ok );
36 }

```

Abbildung 17.105 zeigt das Programm bei der Ausführung.



Abbildung 17.105 Das Programm mit den Auswahlkästchen (»QCheckBox«) bei der Ausführung

Bei dem Auswahlkästchen mit dem Text-Label »Label 1« aus Zeile 02 wurde nichts gemacht. Das Auswahlkästchen »Label 2« wurde in Zeile 06 mit der Methode `setCheckState()` mit `Qt::Checked` abgehakt. Das letzte Auswahlkästchen »Label 3« kann durch `setTristate()` in Zeile 07 einen dritten Zustand annehmen. Diesen Zustand setzen wir gleich in Zeile 08 mit `setCheckState()` auf `Qt::PartiallyChecked` als vorausgewählt und in unserem Fall als nicht mehr veränderbar.

Wenn sich bei den ersten beiden Auswahlkästchen etwas verändert, wird das Signal `stateChanged()` ausgelöst, welches wir in diesem Beispiel mit unserem eigenen Slot `isChecked()` in den Zeilen 09 und 10 mit `connect()` verbunden haben. Der Slot `isChecked()` (Zeile 14 bis 20) macht nichts anderes, als zu überprüfen, ob der Zustand ausgewählt (`Qt::Checked`) oder nicht ausgewählt (`Qt::Unchecked`) ist. Dementsprechend gibt dieser Slot eine Nachrichtenbox aus (siehe Abbildung 17.106).

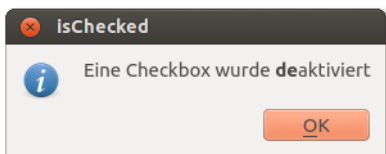


Abbildung 17.106 Der Slot »isChecked()« bei der Ausführung

Wird hingegen versucht, das Auswahlkästchen »Label 3« zu verändern, wird zwar auch das Signal `stateChanged()` abgefangen, aber darauf haben wir hier nicht reagiert. Stattdessen richten wir eine Signal-Slot-Verbindung in Zeile 11 ein, wo wir nur daran interessiert sind, ob jemand diese Kästchen angeklickt (`clicked()`) hat, und reagieren darauf mit dem eigenen Slot `cantChecked()`, welchen wir in den Zeilen 22 bis 25 definiert haben. Dieser Slot macht wiederum nichts anderes, als dem Anwender mitzuteilen, dass der Zustand des Kästchens nicht verändert werden kann (Abbildung 17.107). Da ja der Zustand im Grunde doch verändert werden kann, setzen wir am Ende des Slots in Zeile 24 diesen wieder mit `setCheckState()` auf `Qt::PartiallyChecked`.

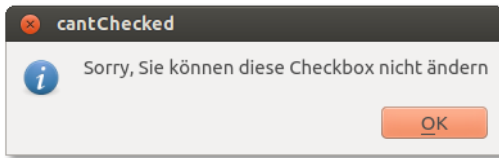


Abbildung 17.107 Der Slot »`cantChecked()`« bei der Ausführung

Für die Schaltfläche AUSWERTEN haben wir in Zeile 12 eine Signal-Slot-Verbindung eingerichtet, wo beim Anklicken der Schaltfläche auf das Signal `clicked()` mit dem Slot `myResult()` reagiert wird. Den Slot haben wir in den Zeilen 27 bis 35 definiert. Dieser macht nichts anderes, als darauf zu prüfen, welche Auswahlkästchen aktuell ausgewählt sind, und gibt eine Information zu allen gesetzten Auswahlkästchen in einer Nachrichtbox zurück (siehe Abbildung 17.108).



Abbildung 17.108 Der Slot »`myResult()`« bei der Ausführung

Radioschaltflächen mit `QRadioButton`

Die Klasse `QRadioButton` ist der Klasse `QCheckBox` recht ähnlich, nur dass hierbei gewöhnlich statt mehrerer nur eine Auswahl aus einer Gruppe von Radioschaltflächen getroffen wird. Die Radioschaltflächen werden in der Praxis auch ein wenig anders als die Auswahlkästchen der Klasse `QCheckBox` visualisiert. Statt eines Hakens oder eines Kreuzes (abhängig vom Fenstermanager des Systems) wird hierbei gewöhnlich ein Punkt verwendet. Dies ist allerdings auch wieder abhängig vom Betriebssystem und dem verwendeten Fenstermanager (Desktop).

Radioschaltflächen sind per Voreinstellung immer `autoExclusive`. Dies bedeutet, Sie können aus mehreren `RadioButtons` innerhalb einer Gruppe, wie bspw. `QButtonGroup`

oder `QGroupBox`, immer nur einen auswählen. Sollten Sie mehrere Gruppen von Radioschaltflächen benötigen, so müssen Sie diese in `QButtonGroup` oder `QGroupBox` stecken.

Die Klasse `QRadioButton` besitzt keine eigenen Signale, Slots oder Methoden. Wird eine Radioschaltfläche ausgewählt, wird das Signal `toggled()` (von der Basisklasse `QAbstractButton`) ausgelöst. Wollen Sie überprüfen, ob ein Button angewählt wurde oder nicht, wird die Methode `isChecked()` verwendet. Vorbelegen können Sie eine Radioschaltfläche in der Gruppe mit der Methode `setChecked(true)`.

Hierzu ein einfaches Beispiel, wie Sie Radioschaltflächen verwenden und auswerten können. Auch hier wird wieder nur eine gekürzte Form abgedruckt. Den kompletten Quellcode mit allen Dateien finden Sie wie immer auf der Buch-CD.

```
00 // kapitel17/qcradiobutton/mydialog.cpp
01 ...
02 // Konstruktor - abgeleitet von QDialog
03 MyQRadioButtonDialog::MyQRadioButtonDialog() {
04 ...
05     radio01 = new QRadioButton("Option 1");
06     radio02 = new QRadioButton("Option 2");
07     radio03 = new QRadioButton("Option 3");
08     button01 = new QPushButton("Auswerten");
09     // radio02 vorbelegen
10     radio02->setChecked(true);
11
12     vbox = new QVBoxLayout;
13     vbox->addWidget(radio01);
14 ...
15     groupBox = new QGroupBox("Radiobutton-Demo");
16     groupBox->setLayout(vbox);
17     QVBoxLayout* myBox = new QVBoxLayout;
18     myBox->addWidget(groupBox);
19     setLayout(myBox);
20
21     connect( radio01, SIGNAL( toggled(bool) ),
22             this, SLOT( changeRadio(bool) ) );
23     connect( radio02, SIGNAL( toggled(bool) ),
24             this, SLOT( changeRadio(bool) ) );
25     connect( radio03, SIGNAL( toggled(bool) ),
26             this, SLOT( changeRadio(bool) ) );
27     connect( button01, SIGNAL( clicked() ),
28             this, SLOT( myResult( ) ) );
29 }
30
31 // Zustand eines Buttons hat sich verändert
```

```

22 void MyQRadioButtonDialog::changeRadio(bool status) {
23     if( status )
24         QMessageBox::information(this, "changeRadio",
25             "Ein Radiobutton wurde geändert",QMessageBox::Ok );
26 }

26 // Radiobuttons auswerten
27 void MyQRadioButtonDialog::myResult() {
28     QString score = "Aktiver Radiobutton: ";
29     if( radio01->isChecked()) score.append("1");
30     if( radio02->isChecked()) score.append("2");
31     if( radio03->isChecked()) score.append("3");
32     QMessageBox::information(
33         this, "Ihre Auswahl", score, QMessageBox::Ok );
34 }

```

Abbildung 17.109 zeigt das Programm bei der Ausführung.



Abbildung 17.109 Die Radioschaltflächen mit »QRadioButton« im Einsatz

In Zeile 08 belegen wir die Radioschaltfläche `radio02` als aktive Schaltfläche beim Programmstart. In den Zeilen 16 bis 18 haben wir für jede Radioschaltfläche das Signal `toggled(bool)`, welches ausgelöst wird, wenn eine andere Radioschaltfläche ausgewählt wurde, mit dem eigenen Slot `changeRadio(bool)` verbunden. Dieser Slot (Zeile 22 bis 25) gibt nur eine Meldung aus, dass sich eine Radioschaltfläche geändert hat (Abbildung 17.110).



Abbildung 17.110 Der Slot »changeRadio(bool)« bei der Ausführung

Für die Schaltfläche AUSWERTEN haben wir eine Signal-Slot-Verbindung in Zeile 19 eingerichtet. Wird hier das Signal `clicked()` ausgelöst, wird der Slot `myResult()` ausgeführt, den wir in den Zeilen 27 bis 33 definiert haben und der uns zurückgibt, welche der Radioschaltflächen aktiv (`isChecked()`) ist (Abbildung 17.111).



Abbildung 17.111 Der Slot »myResult()« gibt aus, welche Radioschaltfläche aktiv ausgewählt ist.

QToolButton

Die Klasse `QToolButton` ist eine weitere Klasse für Schaltflächen, welche gewöhnlich innerhalb einer Werkzeugleiste (`QToolBar`) verwendet wird. Im Gegensatz zu normalen Schaltflächen enthalten solche Tool-Buttons in der Regel kein Text-Label, sondern ein Icon.

17.10.2 Container-Widgets (Behälter-Widgets)

Als Container-Widget werden hier Klassen bezeichnet, die andere Widgets als »Behälter« verwalten. Im Grunde könnte man ja auch `QDialog` und `QWidget` als ein solches Container-Widget bezeichnen, da die hier beschriebenen Container-Widgets nicht ohne die Klassen `QDialog` oder `QWidget` sichtbar sind. Somit sind diese Container-Widgets weitere Sammelbehälter innerhalb von `QDialog` oder `QWidget` (Abbildung 17.112).

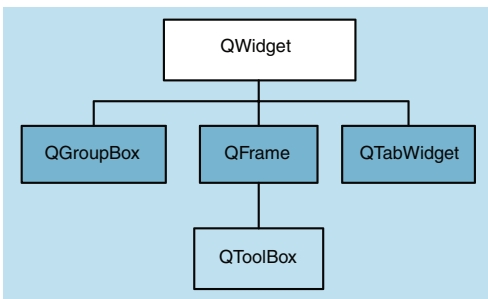


Abbildung 17.112 Klassenhierarchie der Container-Widgets

Verschiedene Widgets in einer Gruppe zusammenfassen mit QGroupBox

Die Klasse `QGroupBox`, welche Sie ja schon des Öfteren verwendet haben, ist ein Widget, das einen Rahmen mit Titel um eine Gruppe von Widgets besitzt. Zusätzlich kann auch hier mit dem Ampersandzeichen ein Tastaturschnellzugriff eingerichtet werden, so dass der Fokus auf eines der Kind-Widgets in der Box fällt. Zwar wurde `QGroupBox` bereits bei den Radioschaltflächen gezeigt, doch gibt es noch eine weitere Option, die hier noch nicht gezeigt wurde. Dabei ist es möglich, diese Box zu aktivieren und wieder zu deaktivieren. Die Option wird mit `setCheckable()` mit `true` aktiviert und mit `false` deaktiviert.

```
00 // kapitel17/groupbox/mywidget.cpp
01 #include "mywidget.h"
...
02 MyWidget::MyWidget( QWidget *parent): QWidget(parent) {
03     // Radio-Elemente des Widgets erzeugen
04     rbutton1 = new QRadioButton ("Radio 1");
05     rbutton2 = new QRadioButton ("Radio 2");
06     rbutton3 = new QRadioButton ("Radio 1a");
07     rbutton4 = new QRadioButton ("Radio 2a");
08     // Attribute setzen
09     rbutton1->setChecked(true);
10     rbutton4->setChecked(true);
11     // Zwei neue Group-Boxen erzeugen
12     groupBox1 = new QGroupBox("Radio-Auswahl &1");
13     groupBox2 = new QGroupBox("Radio-Auswahl &2");
14     // Attribute setzen
15     groupBox2->setCheckable(true);
16     groupBox2->setChecked(false);
17     // Layout erzeugen
18     vBox1 = new QVBoxLayout;
19     vBox2 = new QVBoxLayout;
20     // Elemente ins Layout einfügen
21     vBox1->addWidget(rbutton1);
22     vBox1->addWidget(rbutton2);
23     vBox2->addWidget(rbutton3);
24     vBox2->addWidget(rbutton4);
...
25     // Layout den Group-Boxen hinzufügen
26     groupBox1->setLayout(vBox1);
27     groupBox2->setLayout(vBox2);
28     // Noch ein Layout für die Group-Boxen
29     QVBoxLayout* layout = new QVBoxLayout;
30     layout->addWidget(groupBox1);
```

```

31 layout->addWidget(groupBox2);
32 setLayout(layout);
33 }

```

In den Zeilen 04 bis 10 erzeugen wir Radioschaltflächen. Zwei neue `QGroupBox-Container` werden in den Zeilen 11 und 12 erzeugt. Die erste Box kann hier mit `[Alt]+[1]` und die zweite mit `[Alt]+[2]` angesprungen werden (dank des Ampersandzeichens vor `&1` bzw. `&2` im Text-Label). In Zeile 15 setzen wir die zweite Box mit der Methode `setCheckable()` auf `true`. In der nächsten Zeile setzen wir dann gleich den Wert für diese Box auf `false`, so dass die Box zunächst deaktiviert ist (ausgegraut).

In den nächsten Zeilen wird wieder das typische Layout des Widgets erstellt. Zunächst stecken wir jeweils zwei Radioschaltflächen in ein `QVBoxLayout`, und dann verpacken wir jeweils ein `QVBoxLayout` in einen `QGroupBox-Container`. Zum Schluss packen wir nochmals die beiden `QGroupBox-Container` in ein `QVBoxLayout`.

Abbildung 17.113 und Abbildung 17.114 zeigen das Programm bei der Ausführung.



Abbildung 17.113 Das Programm bei der Ausführung mit deaktivierter »Radio-Auswahl 2«



Abbildung 17.114 Hier wurde die »Radio-Auswahl 2« aktiviert.

Signale und Slot

Für diese Kästchen zum Abhaken gibt es auch zwei Signale, die ausgelöst werden. Da ist zum einen `clicked(bool)`, das ausgelöst wird, wenn das Kästchen betätigt wurde, und zum anderen wird `toggled(bool)` ausgelöst, wenn das Kästchen aktiviert wurde. Die hier vorgestellte Methode `setCheckable()` können Sie außerdem auch als Slot verwenden.

Natürlich bietet das Container-Widget `QGroupBox` noch weitere Methoden und auch Möglichkeiten zum Ausrichten des Titels an. Aber für mehr Details sei wie immer der Qt Assistant empfohlen.

Registerkarten mit `QTabWidget`

Die Klasse `QTabWidget` stellt ein Registerkarten-Widget zur Verfügung. Das Prinzip ist einem Ringordner recht ähnlich, in dem sich häufig einzelne Trennblätter befinden, mit denen man versucht, das Blätter-Chaos zu sortieren. Ähnlich ist dies mit dem Tab-Widget, womit man gegebenenfalls ein Widget-Chaos vermeiden kann.

In der Praxis dürften Sie solche Reiter (engl. *Tabs*) vom Tabbed Browsing her kennen, bei dem jedes Dokument bzw. jede Webseite innerhalb des gemeinsamen Programmfensters über eine Schaltfläche, gewöhnlich am oberen Rand, ausgewählt werden kann.

Standardmäßig wird die Tab-Leiste oben in einem Seitenbereich angezeigt, was aber geändert werden kann. Natürlich kann immer nur ein Seitenbereich, dessen Tab-Leiste angewählt wurde, auf einmal gezeigt werden. Hierbei kann wiederum ein Tastaturkürzel mit dem Ampersandzeichen eingerichtet werden, so dass mit `[Alt] + Zeichen` auf die Tab-Leiste und somit den Seitenbereich zugegriffen wird.

Das `QTabWidget` hat eine beachtliche Anzahl an Methoden. Das folgende Beispiel soll daher nur ein paar grundlegende Funktionen zeigen. Signal gibt es hier nur eines mit `currentChanged(int)`, welches ausgelöst wird, wenn sich der aktuelle Seitenindex verändert hat. Der Parameter enthält den neuen angezeigten Seitenindex. Der Seitenindex wird bei 0 angefangen zu zählen. Auf der anderen Seite gibt es mit `setCurrentIndex(int)` und `setCurrentWidget(QWidget*)` zwei Slots, mit denen zum entsprechenden Seitenindex bzw. zum Seitenbereich mit dem Widget gewechselt wird.

```
00 // kapitel17/qtabwidget/mywidget.cpp
01 #include "mywidget.h"
...
02 MyWidget::MyWidget( QWidget *parent): QWidget(parent) {
03     // Zwei Check-Buttons erzeugen
```

```

04  cbutton2 = new QCheckBox ("Zweites Tab deaktivieren");
05  cbutton3 = new QCheckBox ("Tooltips verwenden");
06  // Vier normale Buttons erzeugen ...
07  pbutton1 = new QPushButton("Norden");
08  pbutton2 = new QPushButton("Süden");
09  pbutton3 = new QPushButton("Westen");
10  pbutton4 = new QPushButton("Osten");
11  // ... und daraus Toggle Buttons machen
12  pbutton1->setCheckable(true);
13  pbutton2->setCheckable(true);
14  pbutton3->setCheckable(true);
15  pbutton4->setCheckable(true);
16  // Nur ein Button darf niedergedrückt sein
17  pbutton1->setAutoExclusive(true);
18  pbutton2->setAutoExclusive(true);
19  pbutton3->setAutoExclusive(true);
20  pbutton4->setAutoExclusive(true);
21  // pbutton1 wird beim Start gesetzt
22  pbutton1->setChecked(true);
23  // Zwei Group-Boxen mit Label erzeugen
24  groupBox2 = new QGroupBox("Attribute");
25  groupBox3 = new QGroupBox("Position");
26  // Zwei vertikale Layout-Boxen erzeugen
27  vbox2 = new QVBoxLayout;
28  vbox3 = new QVBoxLayout;
29  // Check-Buttons in Layoutbox vbox2 packen
30  vbox2->addWidget(cbutton2);
31  vbox2->addWidget(cbutton3);
32  // Toggle Buttons in Layoutbox vbox3 packen
33  vbox3->addWidget(pbutton1);
34  vbox3->addWidget(pbutton2);
...
35  // Layout der einzelnen Group-Boxen setzen
36  groupBox2->setLayout(vbox2);
37  groupBox3->setLayout(vbox3);
38  // Neue Tab-Leiste erzeugen
39  tab = new QTabWidget;
40  // Check-Buttons hinzufügen
41  tab->addTab(groupBox2, "Attribute" );
42  // Groupbox mit Toggle Buttons hinzufügen
43  tab->addTab(groupBox3, "Position" );
44  // Signal-Slot-Verbindung für Check-Buttons
45  connect( cbutton2, SIGNAL( stateChanged(int) ),

```

```

        this, SLOT( setTabAttribute(int) ) );
46 connect( cbutton3, SIGNAL( stateChanged(int) ),
        this, SLOT( setTabAttribute(int) ) );
47 // Signal-Slot-Verbindung für Toggle Buttons
48 connect( pbutton1, SIGNAL( toggled(bool) ),
        this, SLOT( setPosition(bool) ) );
...
49 // Signal-Slot-Verbindung für die Tab-Leiste
50 connect( tab, SIGNAL( currentChanged( int ) ),
        this, SLOT( getCurrentPosition( int ) ) );
51 // Layout-Widget für das Tab-Widget
52 QVBoxLayout* layout = new QVBoxLayout;
53 layout->addWidget(tab);
54 setLayout(layout);
55 setWindowTitle("QTabWidget- Demo");
56 }

57 // Slot: Check-Buttons auswerten
58 void MyWidget::setTabAttribute(int a) {
59     if( cbutton2->isChecked() ) {
60         tab->setTabEnabled(1, false );
61     }
62     else if( ! cbutton2->isChecked() ) {
63         tab->setTabEnabled(1, true );
64     }
65     if( cbutton3->isChecked() ) {
66         tab->setTabToolTip(0, "Gängige Attribute hinzufügen");
67         tab->setTabToolTip(1, "Position vom Tab verändern");
68     }
69     else if ( !cbutton3->isChecked() ){
70         tab->setTabToolTip ( 0, "" );
71         tab->setTabToolTip ( 1, "" );
72     }
73 }

74 // Slot: Toggle Buttons auswerten
75 void MyWidget::setPosition(bool b) {
76     if( pbutton1->isChecked() )
77         tab->setTabPosition(QTabWidget::North);
78     else if( pbutton2->isChecked() )
79         tab->setTabPosition(QTabWidget::South);
80     else if( pbutton3->isChecked() )
81         tab->setTabPosition(QTabWidget::West);

```

```

82     else if( pbutton4->isChecked() )
83         tab->setTabPosition(QTabWidget::East);
84     }

85 // Slot: Tab wurde gewechselt
86 void MyWidget::getCurrentPosition(int pos) {
87     QString msg = QString("Zu Tab %1 gewechselt").arg(
        tab->tabText(pos) );
88     QMessageBox::information(
        this, "Tab gewechselt", msg, QMessageBox::Ok);
89 }

```

Zunächst legen wir in den Zeilen **04** und **05** zwei Schaltflächen zum Abhaken an und dann in den Zeilen **07** bis **10** vier einfache Schaltflächen, welche wir in den Zeilen **12** bis **15** zu niederdrückbaren (*toggled*) Schaltflächen machen. Damit nur eine der Schaltflächen ausgewählt werden kann, machen wir diese in den Zeilen **17** bis **20** `AutoExclusive`.

Als Nächstes erfolgt das Layout wieder, wo wir zwei `QGroupBox`-Elemente anlegen. Dann legen wir zwei `QVBoxLayout`-Elemente an, wo wir in einem die Schaltflächen zum Abhaken und im anderen die Toggle-Schaltflächen hineinlegen. Die `QVBoxLayout`-Elemente wiederum packen wir jetzt jeweils in eine `QGroupBox`, die wir bereits angelegt haben.

In Zeile **39** erzeugen wir jetzt ein neues `QTabWidget`, wo wir in Zeile **41** eine Seite mit dem Titel `ATTRIBUTE` hinzufügen mit dem ersten `QGroupBox`-Element, in dem sich die Schaltflächen zum Abhaken befinden. In Zeile **43** fügen wir eine zweite Seite mit dem Titel `POSITION` dem `QTabWidget` mit den Toggle-Schaltflächen hinzu, die wir ja zuvor auch in eine `QGroupBox` gepackt haben. Wie das Ganze später aussieht, sehen Sie in Abbildung 17.115 und Abbildung 17.116.



Abbildung 17.115 Die erste Seite des »QTabWidget« mit den Schaltflächen zum Abhaken

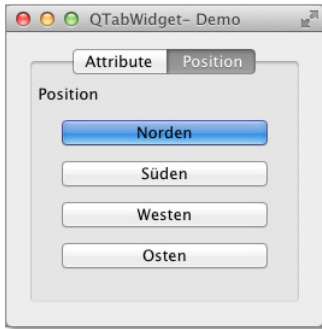


Abbildung 17.116 Die zweite Seite mit niederdrückbaren Schaltflächen

Jetzt folgen die Signal-Slot-Verbindungen. Für beide ankreuzbaren Schaltflächen richten wir eine Signal-Slot-Verbindung in den Zeilen 45 und 46 ein, wo der Slot `setTabAttribute()` aufgerufen wird, wenn sich der Zustand der Schaltflächen geändert hat. Den Slot `setTabAttribute()` haben wir in den Zeilen 57 bis 73 definiert. Aktivieren Sie bspw. ZWEITES TAB DEAKTIVIEREN, wird in Zeile 60 mit der Methode `setTabEnabled()` das zweite Tab POSITION deaktiviert, so dass es nicht mehr ausgewählt werden kann. Entfernen Sie das Häkchen wieder, wird in Zeile 63 das Tab wieder mit derselben Methode aktiviert. Aktivieren Sie hingegen TOOLTIPS VERWENDEN, werden diese in den Zeilen 66 und 67 gesetzt und angezeigt, sobald Sie mit dem Mauscursor über einem Tab stehen bleiben (Abbildung 17.117). Entfernen Sie das Häkchen wieder, so werden diese Tooltips auch in den Zeilen 70 und 71 entfernt.



Abbildung 17.117 Hier wurde das Tab »Position« deaktiviert und die Tooltip-Option aktiviert. Der Tooltip wird angezeigt, wenn Sie mit dem Mauszeiger über einem Tab stehen bleiben.

In Zeile 48 richten wir auch eine Signal-Slot-Verbindung ein, wenn eine der Schaltflächen auf der Seite POSITION niedergedrückt wurde. Gleiches wurde auch für die anderen drei Schaltflächen eingerichtet, nur hier nicht abgedruckt. Wurde eine Schaltfläche niedergedrückt, wird der Slot `setPosition()` aufgerufen, den wir in den Zeilen 74 bis 84 definiert haben, wo wir entsprechend der niedergedrückten Schaltflä-

che die Tabs mit der Methode `setTabPosition()` und den Werten `North`, `South`, `West` oder `East` ausrichten (Abbildung 17.118).

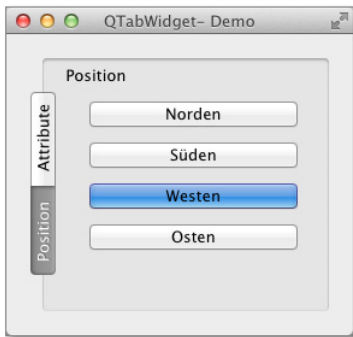


Abbildung 17.118 Hier wurde die Position des Tabs auf »QTabWidget::West« geändert.

Am Ende in Zeile 50 wurde noch eine Signal-Slot-Verbindung eingerichtet, wo das Signal `currentChanged()` ausgelöst wird, sobald die Tabs gewechselt werden. Im konkreten Fall wird lediglich eine Nachrichtbox angezeigt, in welches Tab gewechselt wurde (Abbildung 17.119). Der entsprechende Slot wurde in den Zeilen 86 bis 89 definiert.



Abbildung 17.119 Die Nachrichtbox wird im Beispiel angezeigt, wenn ein Tab gewechselt wurde (das Signal »`currentChanged()`« ausgelöst).

Verschiedene Rahmen mit QFrame

Die von `QWidget` abgeleitete Klasse `QFrame` ist eine Basisklasse für Widgets mit einem Rahmen. Der Rahmen um die Widgets kann dabei verschiedene Formen annehmen. `QFrame` kann (und wird) auch als reiner Platzhalter ohne Inhalt verwendet werden. Dieser Qt-Container bietet wieder eine Menge Methoden. Die wichtigsten sind die Methoden `setFrameShadow()`, womit Sie den Schatten des Rahmens für einen 3D-Effekt, und die Methode `setFrameShape()`, womit Sie den Typ des Rahmens festlegen können. Für beide Methoden stellt `QFrame` einige `enum`-Werte zur Verfügung, für die Stärke der Linie z.B. die Methode `setLineWidth(int)`.

Das Listing auf der Buch-CD in *kapitel17/qframe* demonstriert Ihnen die unterschiedlichen Rahmen- und Schattentypen durch Auswählen der entsprechenden Option (siehe Abbildung 17.120).

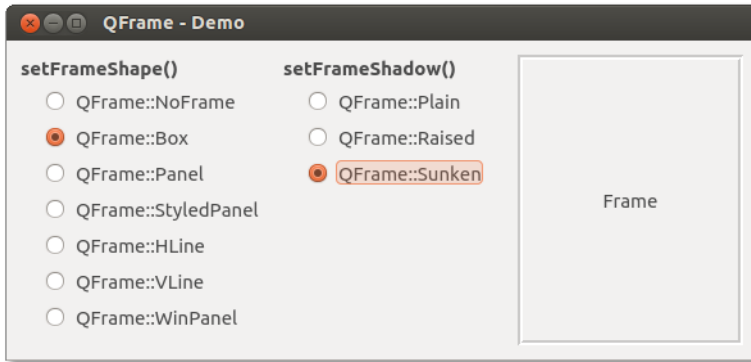


Abbildung 17.120 Der Rahmen um den Text »Frame« auf der rechten Seite wurde hier mit den Optionen »QFrame::Box« und »QFrame::Sunken« realisiert.

Andere Form von Registerkarten mit QToolBox

Die von `QFrame` abgeleitete Klasse ist der Klasse `QTabWidget` recht ähnlich, nur funktioniert diese eben in vertikaler Richtung. Diese Klasse ist eine Spalte mit Tab-Widgets-Elementen, in der immer nur der Seitenbereich eines Tab-Widgets auf einmal angezeigt wird. Da diese Klasse von `QFrame` abgeleitet wurde, kann sie praktisch überall mit eingebaut werden. Auch die Methoden sind denen von `QTabWidget` recht ähnlich, nur dass hierbei, statt dem der Teilbezeichnung »Tab« die Teilbezeichnung »Item« verwendet wird. Zusätzlich können Sie mit dieser Klasse auch noch auf alle Methoden von `QFrame` zurückgreifen. An dieser Stelle soll nicht mehr genauer auf `QToolBox` eingegangen werden. Sollten Sie mehr Informationen zu diesem Container-Widget benötigen, empfehle ich Ihnen wie immer den Qt Assistant. Auf der Buch-CD finden Sie ein Beispiel in *kapitel17/qtoolbox* (siehe Abbildung 17.121).

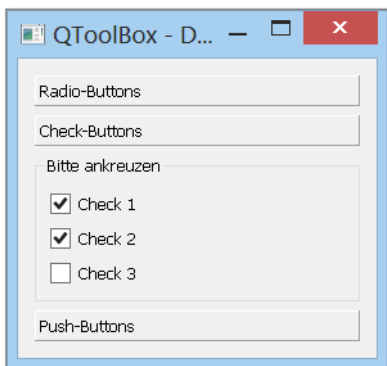


Abbildung 17.121 »QToolBox« beim Einsatz kann wie »QTabWidget« verwendet werden, nur eben in vertikaler Richtung.

17.10.3 Widgets zur Zustandsanzeige

In der Kategorie »Zustandsanzeige« stecken alle Widgets, die einen bestimmten Vorgang bzw. Zustand auf dem Bildschirm anzeigen können, allerdings selbst keinerlei Interaktion anbieten.

Einfachen Text (und Grafik) anzeigen mit QLabel

Die von `QFrame` abgeleitete Klasse `QLabel` wurde ja bereits bei den ersten Qt-Programmen verwendet und dient dazu, einen Text oder ein Bild anzuzeigen ohne jede Anwender-Interaktionen. In diese Klasse können Sie folgende Typen »stecken«:

- ▶ einfachen Text als `QString` mit der Methode `setText()`
- ▶ Rich-Text (RTF) als `QString`, ebenfalls mit der Methode `setText()`
- ▶ Pixmap als `QPixmap` mit der Methode `setPixmap()`
- ▶ Movie (Film) als `QMovie` mit der Methode `setMovie()`
- ▶ eine Zahl als `int` oder `double` mit der Methode `setNum()`



Abbildung 17.122 Mit einem `QLabel` lassen sich auch Grafiken wie hier ein animiertes GIF darstellen. Außerdem lässt sich hiermit auch die Buddy-Funktion erstellen, womit Sie ein `QLabel` mit einem anderen Widget »befreunden« und so mit einem Tastenkürzel ansprechen können, wenn Sie das Ampersandzeichen & vor einen Buchstaben eines Strings im `QLabel` setzen.

Sie sehen, hinter `QLabel` steckt also durchaus ein wenig mehr als ein einfaches Text-Widget. Wenn Sie die Methode `setText()` verwenden, versucht `QLabel` zu ermitteln,

ob es den Text als einfachen (Plain-)Text oder als Rich-Text interpretieren soll. Wollen Sie hierbei sicherstellen, dass tatsächlich Plain- oder Rich-Text verwendet werden, sollten Sie die Methode `setTextFormat()` stattdessen mit dem entsprechenden `Qt::TextFormat` verwenden.

Unterstützte Grafikformate

Welche Formate unterstützt werden, müssen Sie dem Verzeichnis *plugins/imageformats* der Qt-Distribution entnehmen. Darin finden Sie die Librarys bzw. DLLs, die dazu benötigt werden. Hierbei können jederzeit weitere Formate nachinstalliert werden. Wichtig ist auf jeden Fall, dass Sie die Grafikdatei in ein Verzeichnis *imageformats* legen, welches im selben Verzeichnis sein muss, wie die auszuführende Datei (siehe auch das Beispiel im Verzeichnis *kapitel17/qlabel* auf der Buch-CD). Im Beispiel in Abbildung 17.122 wurde ein animiertes GIF verwendet.

Die gängige Anwendung von `QLabel` kennen Sie ja bereits, daher wird hier auf ein Beispiel verzichtet. Wie Sie eine grafische Animation als Label (und ein wenig mehr) verwenden können, finden Sie auf der Buch-CD in dem Beispiel *kapitel17/qlabel*.

LCD-ähnliche Ziffern mit `QLCDNumber`

Mit der Klasse `QLCDNumber` (abgeleitet von `QFrame`) können Sie LCD-ähnliche Ziffern anzeigen lassen. Diese Nummern können sowohl in dezimaler, hexadezimaler, oktal- als auch in binärer Form ausgegeben werden. Das Widget ist schon ziemlich alt, aber kann recht nützlich sein, um fortlaufende oder sich ändernde Werte auf dem Bildschirm in einer klassischen LCD-Anzeige darzustellen.

Die Verwendung ist einfach. Nachdem Sie ein `QLCDNumber` angelegt haben, können Sie den Slot `display()` verwenden, um Werte anzuzeigen. Der Slot liegt mehrfach überladen vor und kann mit `QString`, `int` und `double` umgehen. Die Zahlenbasis stellen Sie mit `setMode()` ein. Standard ist die dezimale Zahlenbasis. Wird ein Wertebereich überschritten, löst `QLCDNumber` das Signal `overflow()` aus.

Sie finden auf der Buch-CD ein Beispiel in *kapitel17/qlcdnumber*, in dem das Datum mitsamt Uhrzeit und der aktuelle Zeitstempel (Timestamp, Sekunden seit dem 1.1.1970) mithilfe von `QLCDNumber` angezeigt werden (siehe Abbildung 17.123).

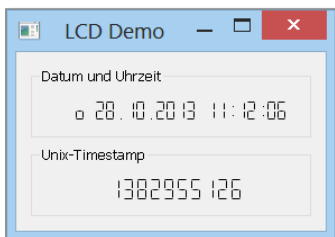


Abbildung 17.123 »QLCDNumber« bei der Ausführung

Fortschrittsbalken

Benötigen Sie einen Fortschrittsbalken, der Anwender darüber informiert, wie weit eine Operation (Kopieren, Verschieben, Herunterladen etc.) fortgeschritten ist, so steht Ihnen ein fertiger Dialog `QProgressDialog` als Klasse zur Verfügung. Der folgende Codeausschnitt soll den Dialog `QProgressDialog` demonstrieren, das Ergebnis sehen Sie in Abbildung 17.124:

```
00 // kapitel17/qprogressdialog/main.cpp
...
01 QProgressDialog progress(
    "Aktueller Fortschritt ...", "Abbrechen", 0, 10);
02 progress.setWindowTitle("QProgressDialog");
03 progress.show();
04 for (int i = 0; i < 10; i++) {
05     progress.setValue(i);
06     if (progress.wasCanceled())
07         break;
08     // ... Arbeit simulieren --- Anfang
09     QTime dieTime= QTime::currentTime().addSecs(1);
10     while( QTime::currentTime() < dieTime )
11         QApplication::processEvents(
            QEventLoop::AllEvents, 100 );
12     // ... Arbeit simulieren --- Ende
13 }
14 progress.setValue(10);
...
```

In Zeile **01** legen wir einen `QProgressDialog` mit dem Label **AKTUELLER FORTSCHRITT** und der Schaltfläche **ABBRECHEN** an. Der minimale Wert beträgt 0 und der maximale 10. Würde bspw. der Wert 5 verwendet, wäre der Fortschrittsbalken bei 50%. In Zeile **02** setzen wir die Titel des Dialogs, und in Zeile **03** wird der Dialog angezeigt (angefangen bei 0%). In Zeile **04** starten wir jetzt die Schleife von 0 bis 10, wie die Arbeiten ausgeführt werden. Mit jedem Schleifendurchlauf wird der Fortschrittsbalken mit `setValue(i)` in Zeile **05** auf einen neuen Wert gesetzt. Ebenfalls wird bei jedem Durchlauf in Zeile **06** geprüft, ob der Vorgang über die Schaltfläche **ABBRECHEN** abgebrochen wurde, also ob `wasCanceled()` gleich `true` ist. In unserem Fall brechen wir dann trocken mit `break` in Zeile **07** den Vorgang ab. Hier können Sie noch eine »Sind Sie sicher?«-Nachrichtbox einbauen. In den Zeilen **08** bis **12** simulieren wir einen Arbeitsvorgang. In unserem Beispiel halten wir jeden der zehn Schleifendurchläufe für 1 Sekunde an. Am Ende in Zeile **14** wird auch noch der maximale Wert gesetzt, damit der Fortschrittsbalken auch 100% erreicht.

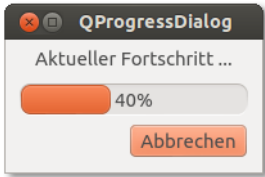


Abbildung 17.124 »QProgressDialog« bei der Ausführung

QProgressBar

Reicht Ihnen die Gestaltungs- und Anwendungsmöglichkeit von QProgressDialog nicht aus, finden Sie mit QProgressBar eine noch flexiblere Lösung, einen Fortschrittsbalken zu erstellen.

17.10.4 Widgets zur Eingabe

Bei den jetzt folgenden Klassen handelt es sich um die gängigen Widgets, die für die Anwender-Interaktion benötigt werden, ohne die keine grafische Oberfläche auskommen würde.

Verschiedene Schieberegler

Ausgehend von der Basisklasse QAbstractSlider bietet Qt mit QSlider und QDial zwei grundlegende Schieberegler an. Wenn Sie sich also durch die Dokumentation des Qt Assistenten durcharbeiten wollen, müssen Sie bei der Basisklasse QAbstractSlider anfangen, welche viele grundlegende Methoden, Slots und Signale für QSlider und QDial anbietet. Der folgende Codeausschnitt soll die grundlegende Verwendung von beiden Schiebereglern in der Praxis demonstrieren:

```
00 // kapitel17/qslider/mywidget.cpp
...
01 slider = new QSlider(Qt::Horizontal);
02 slider->setTickInterval( 10 );
03 slider->setRange ( 0, 100 );
04 slider->setTickPosition(QSlider::TicksAbove);
05 slider->setSliderPosition( 50 );

06 dial = new QDial;
07 dial->setNotchesVisible(true);
08 dial->setRange ( 0, 100 );
09 dial->setNotchTarget ( 7.4 );
10 dial->setWrapping( true );
```

```

...
11  connect( slider, SIGNAL( valueChanged( int ) ),
        this, SLOT( setLabel( int ) ) );
12  connect( dial, SIGNAL( valueChanged( int ) ),
        this, SLOT( setLabel2( int ) ) );
13  connect( but1, SIGNAL(clicked()),
        this, SLOT(calc()));
...
14  void MyWidget::setLabel( int val ) {
15    text->setText(QString("Aktueller Wert: %1").arg(val));
16  }

17  void MyWidget::setLabel2( int val ) {
18    text2->setText(QString("Aktueller Wert: %1").arg(val));
19  }

20  void MyWidget::calc() {
21    QString str("Wert von QSlider: ");
22    str.append(QString("%1").arg(slider->value()));
23    str.append("\nWert von QDial: ");
24    str.append(QString("%1").arg(dial->value()));
25    QMessageBox::information(this, tr("My Application"),
                             str, QMessageBox::Ok);
26  }

```

In Zeile **01** erzeugen wir einen neuen horizontalen Schieberegler (Abbildung 17.125 oben). Natürlich können Sie auch einen vertikalen Schieberegler erzeugen, indem Sie `Qt::Vertical` statt `Qt::Horizontal` verwenden. In Zeile **02** legen Sie den Intervall der einzelnen Striche an. Würden Sie hier 1 angeben und verwenden einen Wertebereich von 0 bis 100 hätten Sie 100 Striche am Schieberegler. Den Wertebereich legen Sie hier in Zeile **03** fest, und wie und wo die einzelnen Striche auf dem Schieberegler sind, geben Sie in Zeile **04** an. Die Position des Schiebereglers legen wir in Zeile **05** fest (hier in der Mitte mit 50).

In Zeile **06** erzeugen Sie einen etwas anderen »Regler«, eine kreisrunde Skala, ähnlich wie bei einem Lautstärkeregler (Abbildung 17.125 unten). Abgesehen vom Aussehen ist die Verwendung von `QDial` ähnlich wie von `QSlider`. Mit Zeile **07** legen Sie fest, dass die Kerben um die Skala sichtbar sind. Den Wertebereich legen Sie auch hier mit der Methode `setRange()` fest. Den Abstand der einzelnen Kerben geben Sie mit Zeile **09** an. Der Standardwert wäre hier 3.7. Normalerweise ist der Regler nicht »durchdrehend« so dass dieser nach einer Umdrehung wieder von vorne anfängt. Mit der Methode `setWrapping(true)` sorgen Sie aber dafür, dass die Skala durchgehend ist.

In den Zeilen 11 und 12 richten wir für beide Regler eine Signal-Slot-Verbindung ein und reagieren auf das Signal `valueChanged(int)`, das ausgelöst wird, sobald sich der Wert der Regler verändert hat. Der Slot `setLabel(int)` setzt dann den Wert des entsprechenden Text-Labels auf den aktuellen Wert. Die Signal-Slot-Verbindung in Zeile 13 hingegen wertet den aktuellen Zustand beider Regler aus und gibt die Werte in einer Nachrichtenbox aus (Abbildung 17.126).

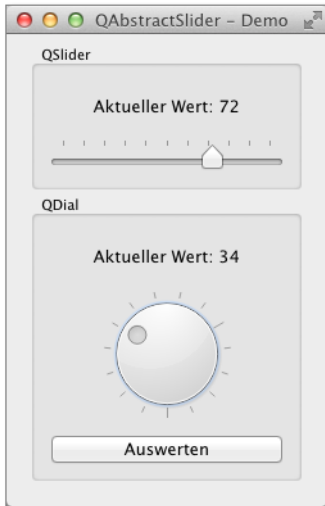


Abbildung 17.125 »QSlider« und »QDial« bei der Ausführung

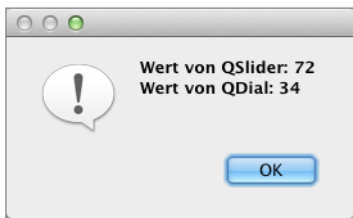


Abbildung 17.126 Die Schaltfläche »Auswerten« wurde betätigt.

QScrollBar

Die Klasse `QScrollBar` stellt eine vertikale oder horizontale Scroll-Leiste zur Verfügung. Diese Klasse wird gewöhnlich dort eingesetzt, wo ein Widget länger bzw. breiter ist, als dies auf dem Bildschirm dargestellt werden kann. Damit kann der Anwender dann zum nicht sichtbaren Bereich navigieren. Wenn Sie ein Scrolling auf ein anderes Widget benötigen sollten, ist es besser, wenn Sie hierzu die Klasse `QScrollArea` verwenden. Diese Klasse sei hier nur vollständigkeitshalber erwähnt.

Einzeiliges und mehrzeiliges Textfeld

Für die Eingabe von Text finden Sie mit `QLineEdit` ein einzeiliges Texteingabefeld vor. `QLineEdit` bietet viele tolle Funktionen auch jenseits vom Kopieren, Ausschneiden, Einfügen und dem Drag & Drop von gewöhnlichem Text. Für mehr Details empfehle ich Ihnen auch hierfür wieder, den Assistenten von Qt zu verwenden.

```
00 // kapitel17/qlineedit/mywidget.h
...
01 MyWidget::MyWidget( QWidget *parent): QWidget(parent) {
02     edit1 = new QLineEdit;
03     // Autovervollständigung verwenden
04     QStringList wordList;
05     wordList << "Galileo" << "Wolf"
           << "Auto" << "Automobil";
06     QCompleter *completer = new QCompleter(wordList,this);
07     completer->setCaseSensitivity(Qt::CaseInsensitive);
08     edit1->setCompleter(completer);
...
09     edit2 = new QLineEdit;
10     edit2->setAlignment(Qt::AlignHCenter);
11     // Integer-Validator verwenden
12     edit2->setValidator(new QIntValidator(edit2));
...
13     connect( radio01, SIGNAL( toggled(bool) ),
           this, SLOT( changeEcho(bool) ) );
14     // Gleiches für 9 weitere Radioschaltflächen
...
15 }

16 void MyWidget::changeEcho( bool b ) {
17     if( radio01->isChecked())
18         edit1->setEchoMode(QLineEdit::Normal);
19     if( radio02->isChecked())
20         edit1->setEchoMode(QLineEdit::Password);
21     if( radio03->isChecked())
22         edit1->setEchoMode(QLineEdit::PasswordEchoOnEdit);
23     if( radio04->isChecked())
24         edit1->setEchoMode(QLineEdit::NoEcho);
25 }

26 // Neue Maske bzw. Validator setzen
27 void MyWidget::setValidMask( bool b ) {
28     edit2->setInputMask("");
```

```

29  edit2->setReadOnly(false);
30  if( radio05->isChecked())
31      edit2->setValidator(new QIntValidator(edit2));
32  else if( radio06->isChecked())
33      edit2->setValidator(new QDoubleValidator(edit2));
34  else if( radio07->isChecked()) {
35      // Integer-Wert zwischen -9999 und 9999
36      QRegExp rx("-?[1-9]\\d{0,3}");
37      QValidator *validator=new QRegExpValidator(rx,this);
38      edit2->setValidator(validator);
39  }
40  else if( radio08->isChecked())
41      edit2->setInputMask("00-00-0000");
42  else if( radio09->isChecked())
43      edit2->setInputMask("000.000.000.000;_");
44  else if( radio10->isChecked())
45      edit2->setReadOnly(true);
46  }

```

In Zeile 02 erzeugen wir ein neues einzeliges Textfeld. Als Nächstes legen wir in Zeile 04 eine String-Liste mit vier Worten an, die wir dann in eine `QCompleter`-Klasse packen (eine Autovervollständigungsklasse). Mit Zeile 07 legen wir fest, dass uns die Groß- und Kleinschreibung egal ist, bevor wir die Autovervollständigung für unser Textfeld in Zeile 08 mit `setCompleter()` setzen (siehe Abbildung 17.127).

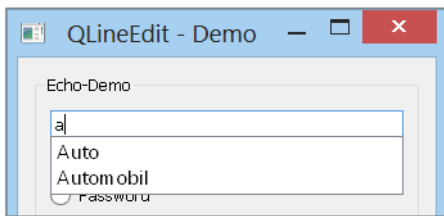


Abbildung 17.127 Die eingerichtete Autovervollständigung im Einsatz

In Zeile 09 legen wir ein zweites Eingabefeld an. In Zeile 10 stellen wir die Textausrichtung auf mittig. In Zeile 12 legen wir mit `setValidator()` fest, was in der Zeile eingegeben werden darf. Hier legen wir mit `QIntValidator` fest, dass nur gültige Integer-Zahlen eingegeben werden dürfen. Es gibt auch noch `QDoubleValidator` (für `double`), `QRegExpValidator` (für reguläre Ausdrücke) oder `QValidator` für eigene Anwendungen.

Zur Demonstration wurden Signal-Slot-Verbindungen für Radioschaltflächen eingerichtet, wie Sie in Zeile 13 sehen, die Sie unterhalb der beiden Textfelder (siehe Abbildung 17.128) auswählen und dann entsprechend im Eingabefeld testen können. Der Slot in den Zeilen 16 bis 25 demonstriert, wie Sie die Ausgabe im ersten Eingabefeld

auf verschiedene Modi setzen können. Der zweite Slot in den Zeilen 26 bis 46 zeigt, wie Sie verschiedene Validatoren (mit `setValidator()`) oder Eingabemasken (mit `setInputMask()`) setzen können.

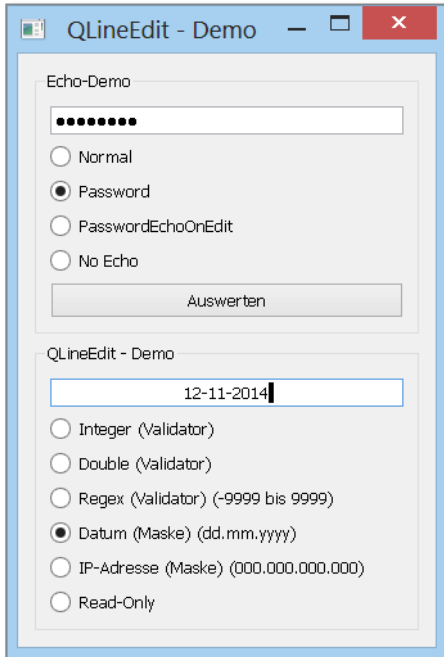


Abbildung 17.128 »QLineEdit« mit verschiedenen Eingabemasken- und Modi im Einsatz

Die Klasse `QTextEdit` ist ein ziemlich fortgeschrittener Text-Betrachter bzw. -Editor, der das Rich-Text-Format (kurz RFT) mit HTML-Formatierung unterstützt (Abbildung 17.129). Auch umfangreiche Texte werden unterstützt (und das ist bei den Text-Widgets anderer GUIs nicht immer selbstverständlich). Die HTML-Unterstützung macht die Formatierung des Textes einfach. Hierzu können einfache HTML-Tags verwendet werden. Setzen Sie bspw. einen Text zwischen `<b>Text</b>`, so wird dieser Text in fetter Schrift dargestellt. Es werden eine Menge Tags wie Bilder, Listen, Tabellen usw. unterstützt. Natürlich können Sie auch HTML-Dateien sowie normalen Text laden.

Die Klasse `QTextEdit` sollte allerdings nicht als Webbrowser verstanden werden. Das Widget will auch kein Ersatz für die klassischen Office-Editoren sein, sondern einfach ein kleiner, schneller und portabler Editor, um sich Hilfen und kleinere Dokumente anzusehen bzw. diese zu editieren. Des Weiteren wurden in diesem Texteditor auch die üblichen Tastenkombinationen zum Editieren des Textes eingebaut. Beispielsweise kopieren Sie mit `[Strg] + [C]` einen markierten Text in die Zwischenablage, und mit `[Strg] + [V]` fügen Sie einen Text aus der Zwischenablage wieder in den Editor ein.

QTextBrowser

Wenn Ihnen jetzt vorschwebt, aus der Klasse `QTextEdit` einen einfachen Text-Browser mit Hypertext-Navigation zu verwenden, dann finden Sie hierfür etwas viel Besseres mit der Klasse `QTextBrowser`. Die Klasse `QTextBrowser` wurde von der Klasse `QTextEdit` als nur lesbar (Read-only-Mode) abgeleitet, und es wurden einige Navigationsmethoden wie vorwärts, rückwärts, Startseite usw. hinzugefügt.

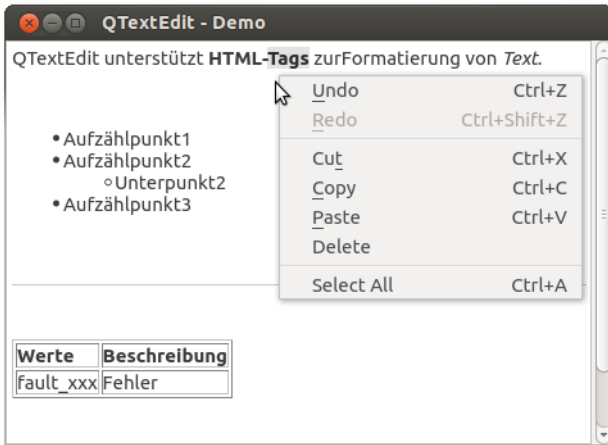


Abbildung 17.129 »QTextEdit« bei der Ausführung

Umfangreiches Beispiel zu QTextEdit

Die Klasse `QTextEdit` wird noch in einem umfangreicheren Beispiel in den Abschnitten 17.11, »Anwendungen in einem Hauptfenster«, und 17.13, »Anwendungen in einem Hauptfenster mit dem Qt Designer«, verwendet. Das Listing für die Abbildung 17.129 hingegen finden Sie auf der Buch-CD im Ordner *kapitel17/qtextedit*.

Eine Combobox zur Auswahl aus einer Liste

Die Klasse `QComboBox` (abgeleitet von `QWidget`) ist eine Kombination aus einem Button und einer Liste. Aus dieser Liste kann der Anwender eine Option auswählen, die dann auch angezeigt wird. Dabei nimmt diese Liste relativ wenig Platz ein, da unbetätigt immer nur ein Element der Liste angezeigt wird. Klickt der Anwender auf diese Box, wird gewöhnlich eine Liste mit den möglichen auszuwählenden Elementen angezeigt. Es ist dabei auch möglich, dass der Anwender den Inhalt einer solchen Box wie bei einem Textfeld editieren kann.

```

00 // kapitel17/qcombobox/mywidget.cpp
...
01 MyWidget::MyWidget( QWidget *parent): QWidget(parent) {
02     combo1 = new QComboBox;
03     combo1->setSizeAdjustPolicy(QComboBox::AdjustToContents );
04     // Ein paar Daten
05     QStringList wordList;
06     wordList << "Data 1" << "Data 2"
           << "Data 3" << "Data 4";
07     QVariant data(wordList);
08     combo1->addItem( " --- Bitte auswählen --- ");
09     combo1->addItem("String-Liste", data );
10     combo1->addItem( "Programmstart",
           QVariant(QTime::currentTime()) );
11     combo1->addItem("Datum",QVariant(QDate::currentDate()));
...
12     combo2 = new QComboBox;
13     // Elemente editierbar
14     combo2->setEditable( true );
15     // Elemente alphabetisch hinzufügen
16     combo2->setInsertPolicy(QComboBox::InsertAlphabetically);
17     but1 = new QPushButton("Hinzufügen");
...
18     connect( combo1, SIGNAL( activated ( int ) ),
           this, SLOT( checkCombo( int ) ) );
19     connect( but1, SIGNAL( clicked() ),
           this, SLOT( addToCombo() ) );
...
20 }

21 // Combo-Box auswerten
22 void MyWidget::checkCombo( int var ) {
...
23 }

24 void MyWidget::addToCombo() {
...
25 }

```

In Zeile **02** erstellen wir eine neue leere Combobox. In Zeile **08** fügen wir der Combobox einen leeren String hinzu. Es können aber auch einem Text mehrere Daten mit `QVariant` übergeben werden, wie dies in Zeile **09** mit einer Liste von Strings demonstriert wird. Gleiches wird auch nochmals mit den Zeilen **10** und **11** gemacht, nur dass

hier neben Text weitere Informationen wie die aktuelle Uhrzeit oder das aktuelle Datum mitgegeben werden.

QVariant

QVariant wird kurz in Abschnitt 17.16.4, »QVariant«, beschrieben. Ebenso findet QTime kurz in Abschnitt 17.16.5, »Typen für Datum und Uhrzeit«, Erwähnung. Aber auch hier gilt, dass Sie die Dokumentation von Qt für mehr Informationen verwenden sollten.

Die zweite Combobox, die wir in Zeile 12 erzeugen, machen wir in Zeile 14 editierbar. Im konkreten Fall soll es möglich sein, eigene Einträge hinzuzufügen (Abbildung 17.130). Mit Zeile 16 sorgen wir dafür, dass die Einträge in alphabetischer Reihenfolge hinzugefügt werden, wenn die Schaltfläche aus Zeile 17 betätigt wurde.



Abbildung 17.130 Das Beispiel mit »QComboBox« bei der Ausführung

Beide Comboboxen sind für sich alleine nutzlos, ohne dass eine Signal-Slot-Verbindung eingerichtet wird. In Zeile 18 reagieren wir auf die erste Combobox, wenn das Signal `activated(int)` ausgelöst wurde, mit dem selbst geschriebenen Slot `check-Combo(int)`, wo die QVariant-Daten ausgewertet und in einer Nachrichtenbox angezeigt werden (Abbildung 17.131).



Abbildung 17.131 Das Datum in »Combo-Box 1« wurde ausgewählt.

Bei der Signal-Slot-Verbindung in Zeile 19 sollten Sie einen Text in der zweiten Combobox eingeben und diesen mit der Schaltfläche HINZUFÜGEN der Combobox hinzufügen. Hierfür reagieren wir auf die Schaltfläche, wenn diese das Signal `clicked()`

auslöst, mit dem selbst geschriebenen Slot `addToCombo()`, womit der Text der Combo-box hinzugefügt wird (Abbildung 17.132). Der Quellcode zu den Slots wurde hier nicht mehr abgedruckt. Aber Sie finden das komplette Listing dazu wie immer auf der Buch-CD in *kapitel17/qcombobox*.



Abbildung 17.132 Ein neuer String wurde hier der »Combo-Box 2« hinzugefügt.

Verschiedene Spin-Boxen

Die von `QWidget` abgeleitete Klasse `QAbstractSpinBox` ist eine Mischung aus einem Textfeld und einem Button mit Pfeilen (oder Plus/Minus), womit Werte angezeigt und verändert werden können. Die Werte können hierbei wie bei einem gewöhnlichen Textfeld durch Eingabe oder eben über die Buttons verändert werden.

Die Klasse `QAbstractSpinBox` ist die Superklasse für die davon abgeleiteten Klassen `QSpinBox`, `QDateTimeEdit` und `QDoubleSpinBox`. Die Klasse `QDateTimeEdit` wieder besitzt mit `QDateEdit` und `QTimeEdit` weitere zwei Unterklassen, die wiederum davon abgeleitet wurden. Eine Suche nach Methoden, Signalen und Slots der darunterliegenden Klassen sollte daher immer bei der Basisklasse `QAbstractSpinBox` erfolgen, wenn Sie in den darunterliegenden Klassen nicht fündig geworden sind.

- `QSpinBox`: Die Klasse `QSpinBox` ist die klassische Integer-Version der Spin-Box. Sollten Sie einen Gleitpunktwert benötigen, können Sie die Klasse `QDoubleSpinBox` stattdessen verwenden.
- `QDoubleSpinBox`: Die Klasse `QDoubleSpinBox` entspricht von der Funktion her exakt der Klasse `QSpinBox`. Selbst die Methoden, Signale und Slots sind die gleichen, die eben bei der Klasse `QSpinBox` beschrieben wurden. Einzig müssen Sie, wo eben ein Integer als Wert gesetzt bzw. zurückgegeben wurde, jetzt den Typ `double` verwenden.
- `QDateTimeEdit`: Die Klasse `QDateTimeEdit` ist die Spin-Box-Version, mit der Sie Datum und Uhrzeit auswählen können. Hiermit ist auch alles möglich, das man von Spin-Boxen her kennt. Datum und Uhrzeit lassen sich kopieren, ziehen (*drag*) und auch gegebenenfalls editieren. Natürlich können Sie hierbei auch das gewünschte Format von Datum und/oder Uhrzeit vorgeben.

- **QTimeEdit und QDateEdit:** Die von QDateTimeEdit abgeleiteten Klassen QTimeEdit und QDateEdit haben keine eigenen Methoden, Signale und Slots. Und diese sind im Grunde ja auch gar nicht nötig, weil die Basisklasse QDateTimeEdit alles zur Verfügung stellt. Im Grunde werden diese beiden Klassen eigentlich gar nicht benötigt, da man ja alles mit QDateTimeEdit machen kann (was diese beiden Klassen intern ja auch machen).

Hierzu ein Codeausschnitt, welcher diese Spin-Boxen bei der Ausführung demonstrieren soll:

```
00 // kapitel17/spinbox/mywidget.cpp
...
01 MyWidget::MyWidget( QWidget *parent): QWidget(parent) {
02     // QSpinBox
03     QLabel *spinLabel = new QLabel(
        "Wert von -100% bis 100% auswählen\n"
        "Bestätigen mit Enter" );
04     spin1 = new QSpinBox;
05     spin1->setRange(-100, 100);
06     spin1->setSingleStep(10);
07     spin1->setValue(0);
08     spin1->setSuffix("%");
09     spin1->setSpecialValueText("Minimum");
10     spin1->setButtonSymbols(QAbstractSpinBox::PlusMinus);
11     spin1->setWrapping(true);
12     spin1->setAccelerated ( true );

13     QLabel *spinLabel2 = new QLabel(
        "Genauigkeit nach dem Komma des"
        " double-Wertes auswählen" );
14     spin2 = new QSpinBox;
15     spin2->setRange(0, 13);
16     spin2->setValue(2);
17     // QDoubleSpinBox
18     QLabel *doubleLabel1 = new QLabel(
        "Einen Wert von 0.0 bis 10.0 auswählen" );
19     dspin = new QDoubleSpinBox;
20     dspin->setRange(0.0, 10.0);
21     dspin->setSingleStep(0.25);
22     dspin->setValue(5.0);
23     dspin->setSpecialValueText("Kein Wert");
24     dspin->setSuffix("°");
25     dspin->setAccelerated ( true );
```



```

26 // QDateTimeEdit
27 QLabel *dtlabel = new QLabel(
    "Bitte ein Datum und eine Zeit auswählen:" );
28 dtspin=new QDateTimeEdit(QDateTime::currentDateTime());
29 dtspin->setCalendarPopup ( true );

30 QLabel *formatLabel = new QLabel(
    "Bitte den Format-String auswählen:" );
31 QComboBox *formatComboBox = new QComboBox;
32 formatComboBox->addItem("yyyy-MM-dd / hh:mm:ss");
33 formatComboBox->addItem("hh:mm:ss MM/dd/yyyy");
34 formatComboBox->addItem("hh:mm:ss dd/MM/yyyy");
35 formatComboBox->addItem("dd.MMM.yyyy - hh:mm:ss");
...
36 connect( spin1, SIGNAL(valueChanged(const QString) ),
    this, SLOT( checkSpin( const QString ) ) );
37 connect( spin2, SIGNAL(valueChanged(int)),
    this, SLOT(changeDecimal(int)));
38 connect( formatComboBox,
    SIGNAL(activated(const QString&)),
    this,SLOT(setFormatString(const QString&)));
39 }

40 void MyWidget::checkSpin( const QString strVal ) {
41     QString svalue = QString(
        "Wert in QSpinBox: %1").arg(strVal);
42     QMessageBox::information( this, "Auswertung",
        svalue, QMessageBox::Ok );
43 }

44 void MyWidget::changeDecimal(int decimals) {
45     dspin->setDecimals(decimals);
46 }

47 void MyWidget::setFormatString(
48     const QString &formatString ){
49     dtspin->setDisplayFormat(formatString);
50 }

```

In Zeile 04 erzeugen wir eine neue Spin-Box und setzen in Zeile 05 den erlaubten Wertebereich von -100 bis 100. Mit `setSingleStep()` setzen wir den Wert, der inkrementiert bzw. dekrementiert wird, wenn die kleinen Schaltflächen der Spin-Box betätigt werden. Standardmäßig ist dieser Wert 1, wir verwenden hier 10er-Schritte. Den

Standardwert beim Programmstart legen wir in Zeile 07 fest. Mit `setSuffix()` in Zeile 08 setzen wir einen String oder ein Zeichen, das hinter dem Wert in der Spin-Box angezeigt wird. Dieses Suffix ist nicht Teil des Wertes, sondern dient nur der Anzeige. Mit `setSpecialValueText()` können Sie einen Text setzen, der angezeigt wird, wenn der minimalste Wert erreicht ist. Mit `setWrapping(true)` in Zeile 11 stellen Sie ein, dass dieser fortfährt, wenn der aktuelle Wert `maximum()` oder `minimum()` erreicht hat. Standardmäßig ist dieser Wert auf `false` gesetzt. Mit `setAccelerate()` in Zeile 12 aktivieren Sie mit `true`, dass sich die Frequenz der einzelnen Schritte beschleunigt, je länger die Schaltflächen der Spin-Box niedergedrückt wurden. In den Zeilen 19 bis 25 machen wir erneut dasselbe, nur dass wir hier eine `QDoubleSpinBox` für Gleitkommazahlen verwenden.

In Zeile 28 erzeugen wir eine neue Spin-Box vom Typ `QDateTimeEdit` und setzen auch gleich das aktuelle Datum und die Uhrzeit. In Zeile 29 schalten wir das Kalender-Popup ein, womit ein Popup-Fenster für die Auswahl des Datums angezeigt wird (Abbildung 17.133). Standardmäßig ist dieser Wert auf `false` gesetzt.

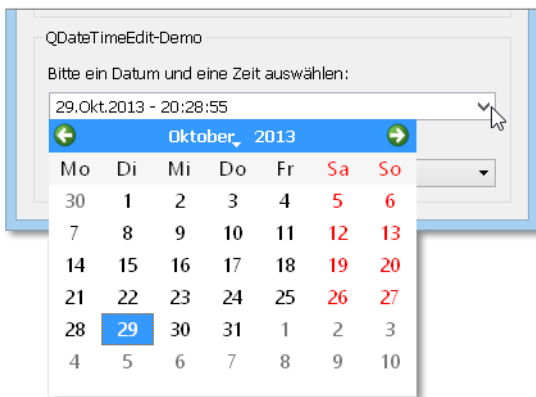


Abbildung 17.133 Das Kalender-Popup-Fenster wird dank »`setCalendarPopup(true)`« angezeigt. Hier können Sie das Datum auswählen.

In den Zeilen 31 bis 35 richten wir eine Combobox ein, womit Sie das Format von `QDateTimeEdit` ändern können. Hierfür wird eine Signal-Slot-Verbindung in Zeile 38 eingerichtet, wo beim Auslösen von `activated(QString)` der selbst geschriebene Slot `setFormatString()` aufgerufen wird. Im Slot, der in den Zeilen 47 bis 50 definiert wurde, setzen wir das Datums- und Zeitformat mit der Methode `setDisplayFormat()` auf das in der Combobox ausgewählte Format.

Mit der Signal-Slot-Verbindung in Zeile 36 hingegen wird die Spin-Box von den Zeilen 04 bis 12 ausgewertet und der Wert in einer Nachrichtenbox angezeigt, was der selbst geschriebene Slot `checkSpin()` für uns macht. Die Signal-Slot-Verbindung in Zeile 37 hingegen wird aktiviert, wenn die Spin-Box der Zeilen 14 bis 16 aktiviert

wurde, und setzt dabei die Anzahl der Nachkommastellen von `QDoubleSpinBox` aus den Zeilen 19 bis 25 mithilfe des Slots `changeDecimal()` in den Zeilen 44 bis 46 auf den dort verwendeten Wert.

Abbildung 17.134 gibt Ihnen einen Eindruck von `QSpinBox`, `QDoubleSpinBox` und `QDateTimeEdit` bei der Ausführung.

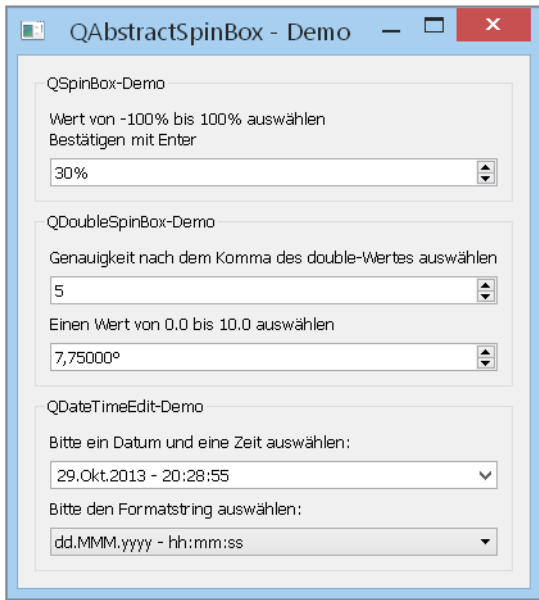


Abbildung 17.134 Verschiedene Spin-Boxen von Qt bei der Ausführung

17.10.5 Online-Hilfen

Dem Anwender Hilfe zum Programm mit an die Hand zu geben, ist das A und O einer guten Anwendung. Dies wird von vielen Programmierern gerne vernachlässigt. Häufig gehen Entwickler davon aus, dass etwas schon selbstsprechend ist. Aus Erfahrung kann ich aber sagen, dass dies leider nicht immer der Fall ist. Häufig muss man nicht mal ganze Dokumentationen erstellen, sondern es reicht aus, kleine Tipps und Direkthilfen bereitzustellen.

Statuszeilentipp

Den Tipp in der Statuszeile werden Sie im nächsten Abschnitt (Abschnitt 17.11, »Anwendungen in einem Hauptfenster«) noch des Öfteren verwenden (Abbildung 17.135). Hierbei hat es oft den Anschein, dass ein solcher Statuszeilentipp nur den Menüelementen und Werkzeugleisten-Buttons vorbehalten ist. Da aber diese Methode auch mit `QWidget::setStatusTip()` vertreten ist, kann man praktisch jedem beliebigen Widget einen Statusstipp hinzufügen (sofern dies Sinn macht).



Abbildung 17.135 Einen Tipp in der Statuszeile anzeigen

Tooltips

Einen Tooltip können Sie ebenfalls bei einem beliebigen Widget verwenden (`QWidget::setToolTip()`), auch wenn dies in der Praxis vorwiegend bei Schaltflächen ohne einen Text, nur mit einem Icon verwendet wird. Ein solcher Tipp erscheint, wenn Sie mit einem Mauszeiger eine bestimmte Zeit über einem Widget verweilen und Sie einen Tooltip mit der Methode `setToolTip()` gesetzt haben. Etwa folgende Zeilen:

```
edit = new QTextEdit;
edit->setToolTip("Hier einen Text eingeben");
```

Verweilt man jetzt mit dem Mauszeiger auf dem Widget `QLineEdit`, erscheint der Tooltip (Abbildung 17.136).

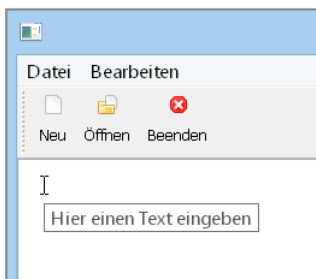


Abbildung 17.136 Ein Tooltip in Textfeld

Man sollte es allerdings nicht übertreiben mit den Tooltips und sie nur wirklich sinnvoll einsetzen. Wie schon erwähnt, werden die Tooltips am meisten mit Schaltflä-

chen oder Widgets verwendet, die nur grafische Elemente anzeigen. Dabei ist es häufig nicht gleich ersichtlich, was passiert wenn man auf eine Schaltfläche mit dieser Grafik klickt.

Direkthilfen

Wenn Sie mehr als nur einen Tooltip anzeigen wollen und dies vor allem ein wenig übersichtlicher machen, dann bietet sich der typische »Was ist das...?«-Dialog an. Diese Direkthilfe kann verwendet werden, wenn sich das Fenster in einem Direkt-hilfe-Modus befindet. Um in diesen Modus zu gelangen, muss man entweder in der Titelleiste auf das Fragezeichen klicken oder die Tastenkombination  + .

Einrichten können Sie eine solche Direkthilfe für jedes Widget mit der Methode `QWidget::setWhatsThis()`. Ein Beispiel:

```
edit = new QLineEdit;
edit->setWhatsThis(
    "Dieser Text kann mit (Shift)+(F1) angezeigt werden."
    "Auch eine <b>HTML-Formatierung</b> ist möglich." );
```

Klicken Sie jetzt diese Direkthilfe an oder betätigen Sie die entsprechende Tastenkombination, bekommen Sie die Hilfe wie in Abbildung 17.137 angezeigt.

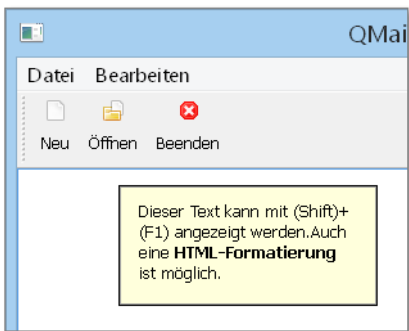


Abbildung 17.137 Diese Direkthilfe wurde per Tastatur aufgerufen.

Am Beispiel können Sie erkennen, dass sich diese Direkthilfe auch mit HTML formatieren lässt, natürlich nur mit den von Qt unterstützten Tags (siehe hierzu auch die Dokumentation von »Richtext«, und zwar den Abschnitt »Supported HTML Subset«).

In einem Hauptfenster wird diese Direkthilfe gewöhnlich über das Menü HILFE implementiert. Dazu bietet die Klasse `QWhatsThis` einige interessante statische Funktionen, wie bspw. `QWhatsThis::createAction()` an.

HTML geführte Dokumentation mit QTextBrowser

Sollten Sie eine etwas umfangreichere Anleitung oder Dokumentation für Ihre Anwendung erstellen wollen, so würde sich hierzu die von `QTextEdit` abgeleitete Klasse `QTextBrowser` hervorragend eignen. Die Klasse `QTextBrowser` liefert einen Rich-Text-Browser mit einer Hypertext-Navigation. `QTextBrowser` ist im Grunde nur eine Erweiterung zur `QTextEdit`-Klasse im Read-only-Modus. Zusätzlich findet man in dieser Klasse einige Navigationsfunktionalitäten, womit der Anwender Links von HTML-Seiten folgen kann.

17.11 Anwendungen in einem Hauptfenster

Als Grundlage für ein Qt-Hauptfenster dient die Klasse `QMainWindow`. An dieser Klasse lassen sich dann ohne großen Aufwand weitere Widgets wie ein Menü, eine Statusleiste, eine Toolbar (dt. Werkzeugleiste) oder ein Dock-Widget anbringen (Abbildung 17.138). Die Hauptrolle des Hauptfensters spielt das Zentral-Widget. Das Zentral-Widget ist einfach das `QMainWindow`-Widget ohne das Menü, ohne eine Statusleiste, ohne eine Toolbar und ohne das Dock-Widget, welches angezeigt wird, wenn man allein ein Objekt der Klasse `QMainWindow` erzeugt.

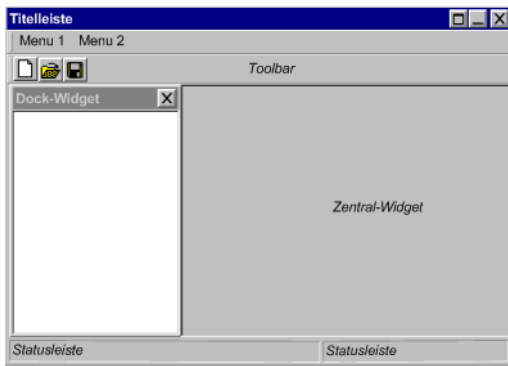


Abbildung 17.138 Grober Aufbau eines Hauptfensters. Natürlich ist die Anordnung abhängig vom System. So finden Sie beim Mac bspw. die Menüleiste im Desktop oberhalb des Bildschirms wieder. Die wirkliche Anordnung hängt auch vom verwendeten Fenstersystem ab.

17.11.1 Die Klasse für das Hauptfenster `QMainWindow`

Wie bereits zuvor erwähnt, ist die Klasse `QMainWindow` für das Hauptanwendungsfenster in Qt verantwortlich. Das Prinzip von `QMainWindow` entspricht wieder dem üblichen GUI-Prinzip, indem man von dieser Klasse eine eigene ableitet. Im Grunde ist eigentlich alles so, wie Sie es schon im Abschnitt zuvor mit dem Ableiten von `QWidget`

bzw. `QDialog` gemacht haben. Zur Demonstration ein einfaches Grundgerüst eines Hauptfensters:

```
00 // kapitel17/qmainwindow1/MyWindow.h
01 #ifndef MyWindow_H
02 #define MyWindow_H
03 #include <QWidget>
04 #include <QMainWindow>
05 #include <QTextEdit>

06 class MyWindow : public QMainWindow {
07     Q_OBJECT
08 public:
09     MyWindow( QMainWindow *parent = 0,
               Qt::WindowFlags flags = 0 );
10     QTextEdit* editor;
11 };
12 #endif
```

In Zeile **06** wird unsere Klasse `MyWindow` von `QMainWindow` abgeleitet. Im Beispiel soll als Zentral-Widget die Klasse `QTextEdit` (Zeile **10**) verwendet werden. Hierzu die Implementierung des Codes:

```
00 // kapitel17/qmainwindow1/MyWindow.cpp
01 #include "MyWindow.h"
02 #include <QApplication>

03 // Neue Widget-Klasse vom eigentlichen Widget ableiten
04 MyWindow::MyWindow( QMainWindow *parent,
                     Qt::WindowFlags flags): QMainWindow(parent, flags) {
05     editor = new QTextEdit;
06     resize(320, 200);
07     setCentralWidget(editor);
08     setWindowTitle("QMainWindow - Demo");
09 }
```

Soweit bietet dieses Beispiel nichts Neues. Wichtig ist allerdings die Methode `setCentralWidget()` in Zeile **07**, mit der Sie das `QTextEdit`-Widget zum Zentral-Widget des Hauptfensters machen. `QTextEdit` ist also der Hauptgrund der Anwendung oder das, worum sich die Anwendung dreht. Hierzu benötigen Sie jetzt nur noch eine Hauptfunktion, um das Programm auszuführen:

```
00 // kapitel17/qmainwindow1/main.cpp
01 #include <QApplication>
```

```

02 #include "MyWindow.h"

03 int main(int argc, char *argv[]) {
04     QApplication app(argc, argv);
05     MyWindow window;
06     window.show();
07     return app.exec();
08 }

```

Abbildung 17.139 zeigt das Programm bei der Ausführung.



Abbildung 17.139 »QMainWindow« bei der Ausführung mit einem »QTextEdit« als Zentral-Widget

Flags für das Hauptfenster QMainWindow

Wie Sie im Listing gesehen haben, kann bei der Erzeugung eines Hauptfensters mit dem zweiten Parameter ein zusätzliches Flag vom Typ `Qt::WindowFlags` verwendet werden, womit einige Optionen um Rahmen, Titelleiste und Systemmenü eines Hauptfensters gesetzt bzw. verändert werden. Sofern es sinnvoll ist, lassen sich mehrere Werte mit dem bitweisen ODER verknüpfen. Einige dieser Flags sind systemabhängig. Für einen Überblick zu den einzelnen Flags verwenden Sie bitte wieder den Assistenten von Qt.

17.11.2 Eine Menüleiste für das Hauptfenster (QMenu und QMenuBar)

Quellcode von Buch-CD

Auf den folgenden Seiten finden Sie jetzt Codeausschnitte aus dem selbst erstellten Texteditor. Das komplette Listing dazu finden Sie auf der Buch-CD unter *kapitel17/JWTexteditor*.

Um typische Menüs für Hauptanwendungen zu verwenden, benötigt man im Grunde nur eine Menüleiste, welche mit der Klasse `QMenuBar` dargestellt wird, und

einzelne Menüelemente, die mit der Klasse `QMenu` realisiert werden. Der Vorgang ist mit der Klasse `QMainWindow` recht einfach zu lösen:

```
01 QMenu *fileMenu = new QMenu(tr("&Datei"), this);
02 menuBar()->addMenu(fileMenu);
```

Zunächst erzeugen Sie ein neues Menü mit der Klasse `QMenu`. In der nächsten Zeile hängen Sie das neue Menü an die Menüleiste von `QMainWindow` an. Dafür sorgt die Methode `menuBar()`. Falls noch keine Menüleiste im Hauptfenster existiert, erzeugt diese Methode eine neue und gibt diese zurück. Die Methode `addMenu()` ist wieder eine Methode der Klasse `QMenuBar`, womit ein neues Element der Menüleiste hinzugefügt wird. Hier wird das neue Objekt `fileMenu` mit dem String »Datei« hinzugefügt (siehe Abbildung 17.140).

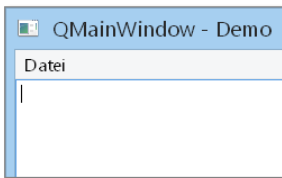


Abbildung 17.140 Ein Element der Menüleiste hinzugefügt

So werden im Grunde weitere Titel der Menüleiste hinzugefügt. Beispielsweise würde folgendermaßen ein weiteres Element »Bearbeiten« der Leiste hinzugefügt werden (Abbildung 17.141):

```
01 QMenu *workMenu = new QMenu(tr("&Bearbeiten"), this);
02 menuBar()->addMenu(workMenu);
```

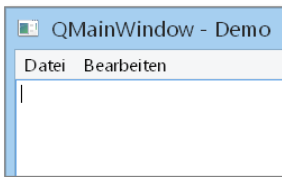


Abbildung 17.141 Ein weiteres Element wurde der Menüleiste hinzugefügt.

tr() für spätere Übersetzungen

Sicherlich ist Ihnen hier schon aufgefallen, dass wir jetzt einige Strings zwischen `tr()` (genauer: `QObject::tr()`) setzen. Diese statische Funktion wird uns später dabei helfen, unser Programm für mehrere Sprachen multilingual zu erstellen. Wie dies funktioniert, erfahren Sie in Abschnitt 17.12, »Anwendung lokalisieren mit Qt Linguistic«.

Nachdem Sie eine Menüleiste mit einem Menü hinzugefügt haben, werden Sie einzelne Menüelemente einfügen wollen. Dies wird gewöhnlich mit der Methode `addAction()` der Klasse `QMenu` erledigt:

```
...
01 QMenu *fileMenu = new QMenu(tr("&Datei"), this);
02 menuBar()->addMenu(fileMenu);
03 act1 = fileMenu->addAction(
    QIcon(QString("%1%2")
        .arg(QCoreApplication::applicationDirPath())
        .arg("/images/page_white.png")), tr("&Neu"),
    this, SLOT(newFile()),
    QKeySequence(tr("Ctrl+N", "Datei|Neu")) );
04 act1->setStatusTip(
    tr("Löscht den aktuellen Inhalt der Datei"));
05 act2 = fileMenu->addAction(
    QIcon(QString("%1%2")
        .arg(QCoreApplication::applicationDirPath())
        .arg("/images/folder_page_white.png")),
    tr("&Öffnen..."), this, SLOT(openFile() ),
    QKeySequence(tr("Ctrl+O", "Datei|Öffnen")));
06 act2->setStatusTip(
    tr("Öffnet eine Datei in den Texteditor"));
07 fileMenu->addSeparator();
    act3 = fileMenu->addAction(
        QIcon(QString("%1%2")
            .arg(QCoreApplication::applicationDirPath())
            .arg("/images/cancel.png")), tr("Be&enden"),
        qApp, SLOT(quit()),
        QKeySequence(tr("Ctrl+E", "Datei|Beenden")) );
08 act3->setStatusTip(tr("Programm beenden"));
...
```

Durch das Hinzufügen der Aktionen (der Klasse `QAction`) sieht das »aufgeklappte« Menü aus wie in Abbildung 17.142.

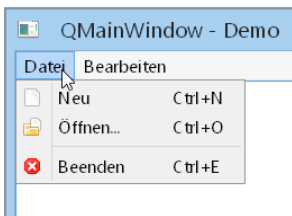


Abbildung 17.142 Einzelne Menüelemente mit Aktionen hinzugefügt

An diesem Punkt werden Sie mit drei Klassen konfrontiert, der Menüleiste (`QMenuBar`), dem Menüelement (`QMenu`) und der Aktion (`QAction`), welche das Menü ausführen soll. Daher sollen hier jetzt die einzelnen Klassen zunächst etwas erläutert werden. Für mehr Details ist hier das Lesen der Dokumentation mithilfe des Assistenten von Qt unverzichtbar.

Icons hinzufügen

Sicherlich sind Ihnen hier folgende Zeilen innerhalb von `addAction()` aufgefallen:

```
QIcon(QString("%1%2")
        .arg(QCoreApplication::applicationDirPath())
        .arg("/images/page_white.png"))
```

Hiermit soll ein Icon hinzugefügt werden. Der einfachere Weg wie

```
QIcon("images/page_white.png")
```

funktioniert in der Regel nur, wenn das Programm aus der Kommandozeile gestartet wird. Mehr dazu können Sie in Abschnitt 17.15.3, »Wenn keine Icons angezeigt werden ...«, nachlesen.

QMenuBar

Die Klasse `QMenuBar` wird für die horizontale Menüleiste verwendet. Selten wird diese Klasse beim Hauptfenster direkt verwendet. Meistens wird hierfür die Methode `menuBar()` von `QMainWindow` benutzt. Im Codeausschnitt haben wir bspw. mit `menuBar()->addMenu(fileMenu)` ganz einfach das Dateimenü der Menüleiste hinzugefügt. Die Methode `addMenu()` liegt mehrfach überladen vor und kann auch nur mit `QString` oder `QIcon` verwendet werden. Soll ein Eintrag irgendwo dazwischen eingefügt werden, können Sie die Methode `insertMenu()` verwenden. Signale, auf die hierbei reagiert werden kann, sind `hovered(QAction)`, das ausgelöst wird, wenn ein Eintrag in der Menüleiste markiert wurde, und das Signal `triggered(QAction)`, welches beim Auswählen einer Aktion im Menü ausgelöst wird.

QMenu

Die Klasse `QMenu` ist ein Menü-Widget für eine Menüleiste, ein Kontextmenü und andere Popup-Menüs. Ein Menü-Widget wird vom Anwender ausgewählt. Dabei kann es als Pulldown-Menü in einer Menüleiste (`QMenuBar`) verwendet werden oder in einem Kontextmenü.

Um ein Menü-Widget in einer Menüleiste zu verwenden, wird die Methode `QMenuBar::addMenu()` verwendet. Kontextmenüs hingegen werden entweder synchron mit `exec()` oder asynchron mit `popup()` aufgerufen. Ein Kontextmenü wird gewöhnlich

geöffnet, wenn der Anwender mit der rechten Maustaste in einem Hauptfensterbereich klickt. Beispielsweise wurde mit folgender `protected`-Methode ein Popup-Fenster im Hauptfenster realisiert:

```
void MyWindow::contextMenuEvent(QContextMenuEvent *event) {
    QMenu *menu = new QMenu(this);
    menu->addAction(act1);
    menu->addAction(act2);
    menu->addAction(act3);
    menu->exec(event->globalPos());
}
```

Für `act1`, `act2` und `act3` wurden Zeiger vom Typ `QAction` als Rückgabewert `fileMenu->addAction(...)` verwendet. Jetzt erscheint bei einem rechten Mausklick (außerhalb des Texteditors) das Popup-Menü, das Sie in Abbildung 17.143 sehen.

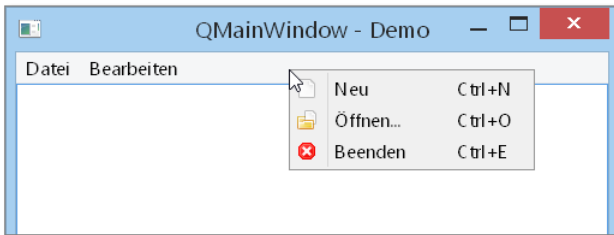


Abbildung 17.143 Popup-Menü mit »QMenu«

Wie schon `QMenuBar` enthält auch `QMenu` u. a. die Signale `hovered(QAction)` und `triggered(QAction)`.

QAction und QActionGroup

Häufig war in den Abschnitten zuvor die Rede von Aktionen der Klasse `QAction`, worauf jetzt in diesem Abschnitt eingegangen wird. Aktionen werden von der Klasse `QAction` repräsentiert und sind abstrakte Benutzerschnittstellen, die einem Widget hinzugefügt werden können.

In vielen Anwendungen werden gewöhnlich Kommandos über Menüs, Schaltflächen in der Werkzeugleiste und Tastaturkürzel eingefügt. Dank der Klasse `QAction` können alle Kommandos über denselben Weg realisiert werden. Das bedeutet auch, Aktionen die über die Klasse `QAction` realisiert werden, müssen nur einmal definiert werden und können dann sowohl für Menüs, Werkzeugleisten-Buttons als auch Tastaturkürzel verwendet werden.

Solche Aktionen können entweder unabhängig vom Objekt erzeugt werden oder werden Konstruktion von bspw. Menüs. Die Klasse `QMenu` (aus dem Abschnitt zuvor)

bspw. bietet hierzu komfortable Methoden an, solche Objekte in einem Schritt zu erzeugen und gleich dem Menü hinzuzufügen.

Ein Objekt der Klasse `QAction` kann Dinge enthalten wie ein Icon, einen Text (für das Menü), ein Tastaturkürzel, einen Text für die Statuszeile, einen »Was ist das?«-Text und einen Text für den Tooltip. Bei Menüs ist es außerdem noch möglich, eine individuelle Schriftart zu verwenden.

Bis jetzt kennen Sie nur den Vorgang mit der Methode `QMenu::addAction()`, um eine Aktion dem Menü hinzuzufügen:

```
01 QMenu *fileMenu = new QMenu(tr("&Datei"), this);
02 menuBar()->addMenu(fileMenu);
03 act1 = fileMenu->addAction(
    QIcon(QString("%1%2")
        .arg(QCoreApplication::applicationDirPath())
        .arg("/images/page_white.png")), tr("&Neu"),
    this, SLOT(newFile()),
    QKeySequence(tr("Ctrl+N", "Datei|Neu")) );
```

Dieser Vorgang hat allerdings den »Nachteil«, dass diese Aktion nicht so ohne Weiteres wiederverwendet werden kann (bspw. für eine gleichwertige Schaltfläche in der Werkzeugleiste). Für solche Fälle können Sie eigene Aktionen instanziiieren. Zum Beispiel:

```
...
01 QAction* underlineAct = new QAction(
    tr("&Unterstreichen"), this );
02 underlineAct->setCheckable(true);
03 underlineAct->setShortcut(tr("Ctrl+U"));
04 underlineAct->setStatusTip(tr("Text unterstreichen"));
05 underlineAct->setIcon(
    QIcon(QString("%1%2")
        .arg(QCoreApplication::applicationDirPath())
        .arg("/images/text_underline.png")));
06 connect( underlineAct, SIGNAL(triggered(bool)),
    editor, SLOT(setFontUnderline(bool)) );
...
```

Zunächst erzeugen Sie in Zeile **01** eine neue Aktion mit dem Text-Label »Unterstreichen«. In Zeile **02** setzen Sie die Aktion als »ankreuzbar«. Bei einer Schaltfläche bedeutet dies bspw., dass diese »toggled« ist und bei einem Menü ist eben ein Häkchen oder ein Kreuz dafür vorhanden. Natürlich geht es auch ohne. In Zeile **03** setzen Sie mit `[Strg]/[U]` ein Tastaturkürzel, womit die Aktion ebenfalls (oder nur) aufgerufen werden kann. Hat die Anwendung eine Statusleiste, wird in Zeile **04** der Text dafür angegeben, der angezeigt wird, wenn der Anwender diese Aktion markiert. In

Zeile 05 fügen wir auch gleich ein Icon der Aktion dazu. Zum Schluss wollen wir noch eine Signal-Slot-Verbindung für unsere Aktion einrichten. Das Signal `triggered()` wird ausgelöst, wenn der Anwender die Aktion betätigt hat. Je nachdem, welchen Wert `triggered()` im Parameter hat, wird beim Texteditor der Slot `setFontUnderline()` ausgeführt. Entweder wird eben der weitere bzw. markierte Text im Editor unterstrichen oder eben nicht unterstrichen ausgegeben.

Eine solche Aktion kann jetzt bspw. folgendermaßen dem Menü BEARBEITEN hinzugefügt werden:

```
01 QMenu *work = menuBar->addMenu(tr("&Bearbeiten"));
02 work->addAction(underlineAct);
```

Manchmal wird auch eine Art Radioschaltflächen-Funktion benötigt, wenn eben nur eine von mehreren Aktionen aktiv sein darf/soll. Für solche Zwecke bietet Qt gruppierte Aktionen mit der Klasse `QGroupAction` an. Hierzu legt man zunächst die einzelnen Aktionen vom Typ `QAction` an:

```
...
01 QAction *leftAlignAct = new QAction(
    tr("&Left Align"), this);
02 leftAlignAct->setCheckable(true);
03 leftAlignAct->setShortcut(tr("Ctrl+L"));
04 leftAlignAct->setStatusTip(tr("Text links ausrichten"));
05 leftAlignAct->setIcon(
    QIcon(QString("%1%2")
        .arg(QCoreApplication::applicationDirPath())
        .arg("/images/text_align_left.png")));
06 connect( leftAlignAct, SIGNAL(triggered()),
    this, SLOT(leftAlignment()));

07 QAction *rightAlignAct = new QAction(
    tr("&Right Align"), this);
08 rightAlignAct->setCheckable(true);
09 rightAlignAct->setShortcut(tr("Ctrl+R"));
10 rightAlignAct->setStatusTip(tr("Text rechts ausrichten"));
11 rightAlignAct->setIcon(
    QIcon(QString("%1%2")
        .arg(QCoreApplication::applicationDirPath())
        .arg("/images/text_align_right.png")));
12 connect( rightAlignAct, SIGNAL(triggered()),
    this, SLOT(rightAlignment()));

13 QAction *justifyAct = new QAction(tr("&Justify"), this);
14 justifyAct->setCheckable(true);
```

```

15 justifyAct->setShortcut(tr("Ctrl+J"));
16 justifyAct->setStatusTip(tr("Text bündig ausrichten"));
17 justifyAct->setIcon(
    QIcon(QString("%1%2")
        .arg(QCoreApplication::applicationDirPath())
        .arg("/images/text_align_justify.png")));
18 connect( justifyAct, SIGNAL(triggered()),
    this, SLOT(justifyAlignment()));

19 QAction *centerAct = new QAction(tr("&Center"), this);
20 centerAct->setCheckable(true);
21 centerAct->setShortcut(tr("Ctrl+E"));
22 centerAct->setStatusTip(tr("Text zentriert ausrichten"));
    centerAct->setIcon(
        QIcon(QString("%1%2")
            .arg(QCoreApplication::applicationDirPath())
            .arg("/images/text_align_center.png")));
23 connect( centerAct, SIGNAL(triggered()),
    this, SLOT(centerAlignment()));
...

```

Hier haben Sie vier verschiedene Aktionen zum Ausrichten des Textes im Texteditor. Da keine dieser Ausrichtungen gleichzeitig verwendet werden soll, benötigen Sie eine Zusammenfassung dieser vier Aktionen zu einer Gruppe. Dies lässt sich relativ einfach mit der Klasse `QActionGroup` realisieren:

```

01 QActionGroup* alignmentGroup = new QActionGroup( this );
02 alignmentGroup->addAction(leftAlignAct);
03 alignmentGroup->addAction(rightAlignAct);
04 alignmentGroup->addAction(justifyAct);
05 alignmentGroup->addAction(centerAct);
06 leftAlignAct->setChecked(true);

```

Zunächst erzeugen Sie ein `QActionGroup`-Objekt, wo Sie die vier einzelnen Aktionen hineinpacken. Die Aktion `leftAlignAct` wird beim Start des Programms als Standard angekreuzt.

Machen Sie aber jetzt nicht den Denkfehler, und versuchen Sie nur das `QActionGroup`-Objekt dem Menü hinzuzufügen. Die Klasse `QActionGroup` dient lediglich dazu, einzelne Aktionen zu gruppieren. Das Visuelle müssen Sie nach wie vor einzeln dem Menü hinzufügen:

```

...
01 QMenu *formatMenu = workMenu->addMenu(tr("&Format") );
02 formatMenu->addAction(underLineAct);

```

```

03 formatMenu->addSeparator(); // Trennlinie
04 formatMenu->addAction(leftAlignAct);
05 formatMenu->addAction(rightAlignAct);
06 formatMenu->addAction(justifyAct);
07 formatMenu->addAction(centerAct);
08 formatMenu->addSeparator(); // Trennlinie
...

```

In Zeile **01** fügen wir dem Menü **BEARBEITEN** (`workMenu`) einen weiteren Eintrag »Format« hinzu. Diesem Eintrag fügen wir jetzt in den Zeilen **02** bis **08** weitere Einträge, hier unsere zuvor erstellten Aktionen, mit der Methode `addAction()` hinzu. Bei der Ausführung sieht das Ganze dann aus wie in Abbildung 17.144.

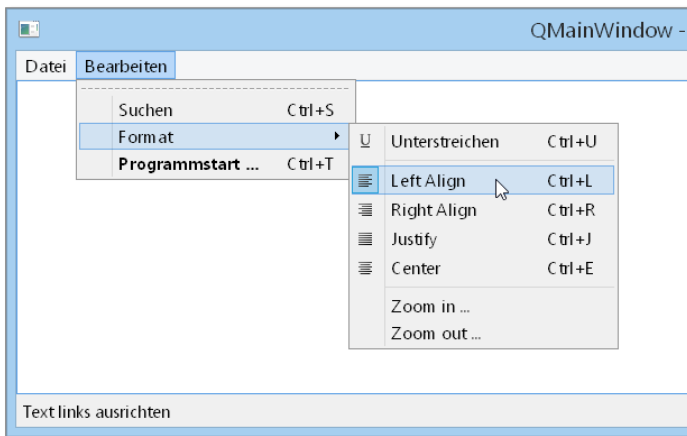


Abbildung 17.144 Gruppierete Aktionen dem Menü hinzugefügt

Menüleiste bei Mac OS X

Als Mac-Anwender ist man es ja gewohnt, dass die Menüleiste oberhalb vom Bildschirm angebracht ist. Wahrscheinlich hier überflüssig zu erwähnen, aber das gilt auch für Anwendungen, die mit Qt erstellt wurden. Auch hier finden Sie die Menüleiste, wie für Mac OS X eben üblich, am oberen Bildschirmrand (Abbildung 17.145).

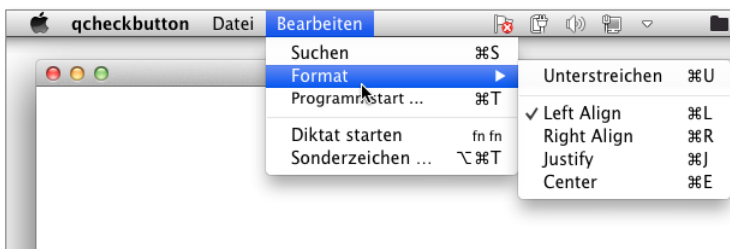


Abbildung 17.145 Natürlich sind auch bei Qt-Anwendungen die Menüleisten, wie für den Mac üblich, oben angebracht.

17.11.3 Eine Statusleiste mit QStatusBar

Die Statusleiste ist eine horizontale Leiste unterhalb des Hauptfensters, welche Informationen zum aktuellen Status der Anwendung ausgeben kann. Die Klasse, mit der man eine solche Leiste erzeugt, heißt `QStatusBar`. In der Praxis wird allerdings eher selten direkt über diese Klasse eine Instanz erzeugt. Gewöhnlich benutzt man hierfür die Methode `QMainWindow::statusBar()`, welche einen Zeiger auf ein von `QMainWindow` benutztes `QStatusBar`-Objekt zurückgibt. Sollte das Hauptfenster noch keine Statusleiste besitzen, wird eine erzeugt beim Aufruf von `statusBar()`. In einer `QMainWindow`-Instanz lässt sich somit mit folgender Zeile eine Statusleiste erzeugen:

```
MyWindow::MyWindow( ... ) {
...
    (void*)statusBar();
...
}
```

Über diese Methode, kann nun auf die einzelnen Methoden, Slots und Signale der Statusleiste (`QStatusBar`) zugegriffen werden:

```
statusBar()->methodNamevonQStatusBar();
```

Bevor hier auf die einzelnen Methoden der Klasse `QStatusBar` eingegangen wird, sollen schon mal vorweg die drei verschiedenen Möglichkeiten erläutert werden, wie man Meldungen an die Statusleiste ausgeben kann.

Temporäre Nachricht in der Statusleiste anzeigen

Temporäre Meldungen können jederzeit in der Statusleiste mit den Slots `QStatusBar::showMessage()` und `QStatusBar::clearMessage()` angezeigt bzw. wieder entfernt werden, bspw. beim erfolgreichen Öffnen einer Datei:

```
// Datei öffnen und im Editor anzeigen
void MyWindow::openFile() {
...
    // Datei erfolgreich geöffnet; Meldung an Statusleiste
    statusBar()->showMessage(tr("Datei erfolgreich geladen"));
...
}
```

In der Praxis sieht dies aus wie in Abbildung 17.146.



Abbildung 17.146 Temporärer Text wird in der Statusleiste angezeigt.

Diese Meldung bleibt jetzt so lange angezeigt, bis Sie sie explizit mit `QStatusBar::clearMessage()` löschen oder bis eine neue Meldung in der Statusleiste gesetzt wird.

```
// Text in der Statusleiste löschen
statusBar()->clearMessage();
```

Mit `QStatusBar::showMessage()` haben Sie allerdings auch die Möglichkeit, einen Text nur für eine bestimmte Zeit anzuzeigen. Dieser Slot besitzt noch einen zweiten Parameter, womit Sie die Zeit in Millisekunden angeben, solange der Text in der Statusleiste angezeigt werden soll, bspw. bleibt der folgende Text für 3 Sekunden in der Statusleiste sichtbar:

```
// Text in der Statusleiste für 3 Sekunden anzeigen
statusBar()->showMessage(tr("Datei erfolgreich geladen"),3000);
```

Für temporäre Meldungen verwendet man gewöhnlich Informationen, die nur zeitlich begrenzt sichtbar sein sollen.

Normale Nachrichten in der Statusleiste

Normale Meldungen werden immer angezeigt. Dies sind Meldungen, die sichtbar werden, wenn Sie bspw. mit der Maus über einem Menü-Widget oder einer Schaltfläche einer Werkzeugleiste stehen (*Hovered*) und hierfür etwa mit der Methode `setStatusTip()` von `QAction` einen Text gesetzt haben, bspw.:

```
...
QMenu *workMenu = new QMenu(tr("&Bearbeiten"), this);
menuBar()->addMenu(workMenu);
```

```
QAction* act4 = workMenu->addAction(
    tr("&Suchen"), this, SLOT(search()),
    QKeySequence(tr("Ctrl+S", "Bearbeiten|Suchen")));
// Den Text für die Statusleiste setzen
act4->setStatusTip(tr("Nach einer Stringfolge suchen"));
```

Dies sieht in der Praxis dann z.B. aus wie in Abbildung 17.147.

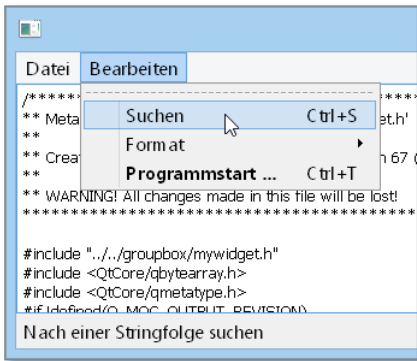


Abbildung 17.147 Normale Nachricht in der Statusleiste

In der Praxis bedeutet dies allerdings nicht, dass Sie auf normalen Text in der Statusleiste beschränkt sind. Sie können sich hierbei, sofern sinnvoll (bspw. Fortschrittsbalken), auch andere Widgets mit der Methode `QStatusBar::addWidget()` reinpacken und mit `QStatusBar::removeWidget()` wieder entfernen.

Temporäre Nachricht hat Vorrang

Wenn eine temporäre Nachricht ausgeführt wird, wird die normale Nachricht überdeckt.

Dauerhafte Nachrichten in der Statusleiste

Benötigen Sie Meldungen in der Statusleiste, die permanent angezeigt werden und garantiert nicht von einer temporären oder normalen Meldung überdeckt werden, dann müssen Sie die Methode `QStatusBar::addPermanentWidget()` verwenden. Permanente Meldungen erscheinen immer außen rechts. Wird eine weitere permanente Meldung hinzugefügt, wird die äußerste (rechts außen) Meldung um eine Position nach innen verschoben. Hier ein Beispiel einer solchen permanenten Meldung:

```
MyWindow::MyWindow(QMainWindow *parent, Qt::WindowFlags flags):
    QMainWindow(parent, flags) {
    ...
```

```

(void*) statusBar();
sLabel = new QLabel;
sLabel->setFrameStyle(QFrame::Panel | QFrame::Sunken);
sLabel->setLineWidth(2);
statusBar()->addPermanentWidget(sLabel);
connect( editor, SIGNAL( textChanged() ),
        this, SLOT( updateStatusBar() ) );
updateStatusBar();
...
}

```

Hier wird ein Label als permanentes Widget der Statusleiste hinzugefügt. In diesem Zusammenhang richten wir eine Signal-Slot-Verbindung zu unserem QTextEdit-Objekt und dem Hauptfenster ein. Wurde der Text verändert (`textChanged()`), wird der Slot `updateStatusBar()` ausgeführt. Dieser wird bei der Ausführung des Konstruktors beim ersten Mal auch manuell ausgeführt. Hierzu der Quellcode des Slots `updateStatusBar()`:

```

void MyWindow::updateStatusBar() {
    QString str = editor->toPlainText();
    int count_char = str.length();
    int count_words = str.count(" ");
    int count_lines = str.count("\n");
    QString label = tr("Zeichen: %1 Wörter: %2 Zeilen: %3")
        .arg(count_char).arg(count_words).arg(count_lines+1);
    sLabel->setText(label);
}

```

Damit haben Sie im Grunde nichts anderes als ein weiteres Feld rechts in der Statusleiste hinzugefügt, in dem die Anzahl der Zeichen, Wörter und Zeilen des Editors angezeigt und ständig (sobald der Text geändert wurde) aktualisiert werden.

Natürlich kann man auch hier andere Widgets als nur `QLabel` verwenden, bspw. lässt sich eine einfache Datums- und Uhrzeitanzeige mit dem LCD-Widget (`QLCDNumber`) folgendermaßen realisieren:

```

// Konstruktor
MyWindow::MyWindow(QMainWindow *parent,
                  Qt::WindowFlags flags):
    QMainWindow(parent, flags) {
...
    time = new QLCDNumber;
    time->setFrameStyle(QFrame::Panel | QFrame::Sunken);

```

```

time->setLineWidth(2);
time->setSegmentStyle(QLCDNumber::Flat);
statusBar()->addPermanentWidget(time);
QTimer *timer = new QTimer(this);
connect( timer, SIGNAL(timeout()),
        this, SLOT(updateTime()) );
// Jede Sekunde aktualisieren
timer->start(1000);
updateTime();
...
}
...
void MyWindow::updateTime() {
    // Aktuelles Datum und Uhrzeit
    QDateTime Time = QDateTime::currentDateTime();
    // Formatieren
    QString text = Time.toString("dd.MM.yyyy hh:mm:ss");
    // Anzahl der Ziffern setzen
    time->setNumDigits(text.size()+1);
    // Und alles anzeigen
    time->display(text);
}

```

In der Praxis würde unsere Statusleiste mit den permanenten Meldungen aussehen wie in Abbildung 17.148.

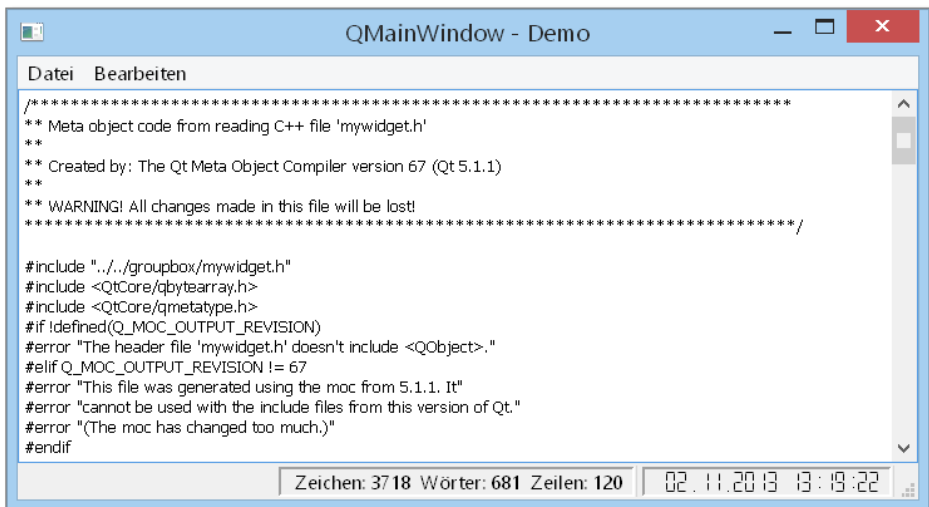


Abbildung 17.148 Permanente Meldungen in der Statusleiste

Natürlich wird auch hier wieder empfohlen, sich mithilfe des Assistenten von Qt tiefer in die Klasse von `QStatusBar` und deren Methoden, Signale und Slots einzulesen.

17.11.4 Eine Werkzeugleiste mit der Klasse `QToolBar`

Zur Werkzeugleiste mit Buttons braucht man eigentlich nicht allzu viel zu sagen, weil alles schon bei den Menü-Widgets erwähnt wurde. Sämtliche Aktionen (`QAction`) lassen sich auch in der Werkzeugleiste (wieder-)verwenden. Zusätzlich bietet eine Werkzeugleiste noch Optionen an, mit denen man die Leiste mitsamt Buttons verschieben kann.

Auch für das Hinzufügen einer Werkzeugleiste bietet das Hauptfenster mit `QMainWindow::addToolBar()` eine komfortable Methode an, ohne direkt eine Instanz von `QToolBar` zu erzeugen. So geht man in der Regel wie folgt vor:

```
// Neue Werkzeugleiste
toolFile = addToolBar(tr("Datei"));
toolFile->addAction(act1);
toolFile->addAction(act2);
toolFile->addAction(act3);
// Neue Werkzeugleiste
toolWork = addToolBar(tr("Bearbeiten"));
toolWork->addAction(underlineAct);
toolWork->addSeparator();
toolWork->addAction(leftAlignAct);
toolWork->addAction(rightAlignAct);
toolWork->addAction(justifyAct);
toolWork->addAction(centerAct);
toolWork->addSeparator();
// Font-Combobox zur Werkzeugleiste
QFontComboBox *cFont = new QFontComboBox;
toolWork->addWidget(cFont);
connect( cFont, SIGNAL(currentFontChanged ( const QFont& )),
        editor, SLOT(setCurrentFont ( const QFont&)));
```

Nachdem Sie eine Werkzeugleiste erzeugt und gleichzeitig dem Hauptfenster hinzugefügt haben, können Sie die bereits vorhandenen (oder gegebenenfalls neuen) Aktionen der Werkzeugleiste hinzufügen. Im Beispiel wurde auf die bereits im Menü-Widget erzeugten Aktionen zurückgegriffen. Zusätzlich wurde noch passend dazu eine Font-Combobox hinzugefügt und eine entsprechende Signal-Slot-Verbindung dazu eingerichtet. Im Beispiel wurden der Aktion im Gegensatz zum Menü-Widget außerdem noch ein paar Icons hinzugefügt. In der Praxis sieht dieses Beispiel wie in Abbildung 17.149 aus.



Abbildung 17.149 Das Hauptfenster, jetzt mit einer Werkzeugleiste

Die Werkzeugleisten lassen sich per Standard jetzt fast beliebig an jeder Seite im Hauptfenster verschieben, wie Abbildung 17.150 und Abbildung 17.151 zeigen sollen.

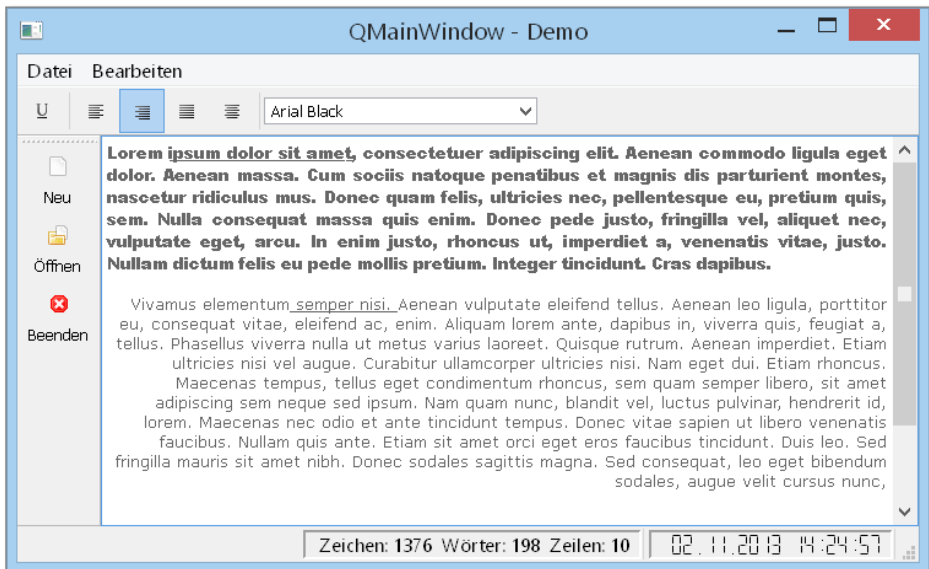


Abbildung 17.150 Die Werkzeugleiste ist dank »setAllowedAreas(Qt::LeftToolBarArea|Qt::TopToolBarArea)« nach links und nach oben verschiebbar.

Selbstverständlich können Sie solche »Schiebereien« des Anwenders mit der Methode `QToolBar::setMovable(false)` auch unterbinden. Die Methode `QMainWindow::addToolBar()` gibt es auch noch in einer zweiten Form (`QMainWindow::addToolBar(Qt::ToolBarArea area, QToolBar* toolbar)`), womit Sie die Position der Werkzeugleiste festlegen. Mögliche Werte für `Qt::ToolBarArea` entnehmen Sie wie immer dem Assistenten von Qt.

Wollen Sie bspw. dass eine Werkzeugleiste beim Programmstart unten erscheint (Abbildung 17.151), müssen Sie dies nur folgendermaßen machen:

```
toolWork = addToolBar(tr("Bearbeiten"));
// Werkzeugleiste toolWork unten anzeigen
addToolBar(Qt::BottomToolBarAreas, toolWork );
```



Abbildung 17.151 Hier wurde die Werkzeugleiste unten angebracht.

17.11.5 An- und abdockbare Widgets im Hauptfenster mit QDockWidget

Ein Widget der Klasse `QDockWidget` kann dazu verwendet werden, ein weiteres Fenster mit beliebigen Widgets innerhalb des Hauptfensters (`QMainWindow`) zu platzieren. Dieses Dock-Fenster kann entweder fest im Hauptfenster integriert oder als verschiebbares Top-Level-Fenster auf dem Desktop platziert werden. Dock-Widgets sind so etwas wie ein zweites Fenster im Hauptfenster (`QMainWindow`). Ebenso können auch mehrere Dock-Widgets auf einmal verwendet werden. Platziert werden diese Dock-Widgets immer um das Zentral-Widget von `QMainWindow`.

Ein Dock-Fenster lässt sich von einem Bereich in einen anderen verschieben oder auch komplett vom Hauptfenster lösen. Ebenso kann, wie auch bei einer Werkzeugleiste, festgelegt werden, wohin ein Dock-Fenster verschoben werden darf oder ob man es auch schließen kann.

Grundlegend enthält ein Dock-Fenster eine Titelleiste und einen Bereich mit einem Inhalt. Zusätzlich befindet sich in der Titelleiste ein Button zum Schließen und Löschen des Dock-Fensters. Allerdings sind diese Optionen auch anpassbar.

Ein einfaches Beispiel hierzu. Mit den folgenden wenigen Zeilen wurde ein einfaches Dock-Fenster auf der rechten Seite dem Hauptfenster hinzugefügt. Dabei handelt sich um einen einfachen WYSIWYG-Editor (dt. so viel wie »Was du siehst, ist, was du bekommst«). Das heißt, Sie können im linken Zentral-Widget nach wie vor einen Text eingeben. Verwenden Sie hierzu bspw. HTML-Tags, dann wird dies im Dock-Fenster (worin ebenfalls ein Widget der Klasse `QTextEdit` haust) auch als HTML-formatierter Text angezeigt. Hierzu der entsprechende Quellcode:

```
// Konstruktor
MyWindow::MyWindow(QMainWindow *parent,
                  Qt::WindowFlags flags):
    QMainWindow(parent, flags) {
...
    // Ein neues Dock erzeugen
    QDockWidget *dock = new QDockWidget(
        tr("WYSIWYG-Editor"), this);
    // nur im rechten und linken Bereich erlauben
    dock->setAllowedAreas(
        Qt::LeftDockWidgetArea | Qt::RightDockWidgetArea);
    // Widget für das Dock-Widget erzeugen
    dock_editor = new QTextEdit;
    // Editor als Widget im Dock-Fenster verwenden
    dock->setWidget(dock_editor);
    // auf der rechten Seite dem Hauptfenster hinzufügen
    addDockWidget(Qt::RightDockWidgetArea, dock);
    connect( editor, SIGNAL(textChanged() ),
            this, SLOT( showHTML() ) );
...
}
...
// Slot showHTML()
void MyWindow::showHTML() {
    QString str = editor->toPlainText();
    dock_editor->setHtml(str);
}
```

Dem Hauptfenster QMainWindow wurde dieses Dock-Widget mit der Methode QMainWindow::addDockWidget() hinzugefügt. Bei der Ausführung sieht dieses an- bzw. abdockbare Widget aus wie in Abbildung 17.152 und Abbildung 17.153.

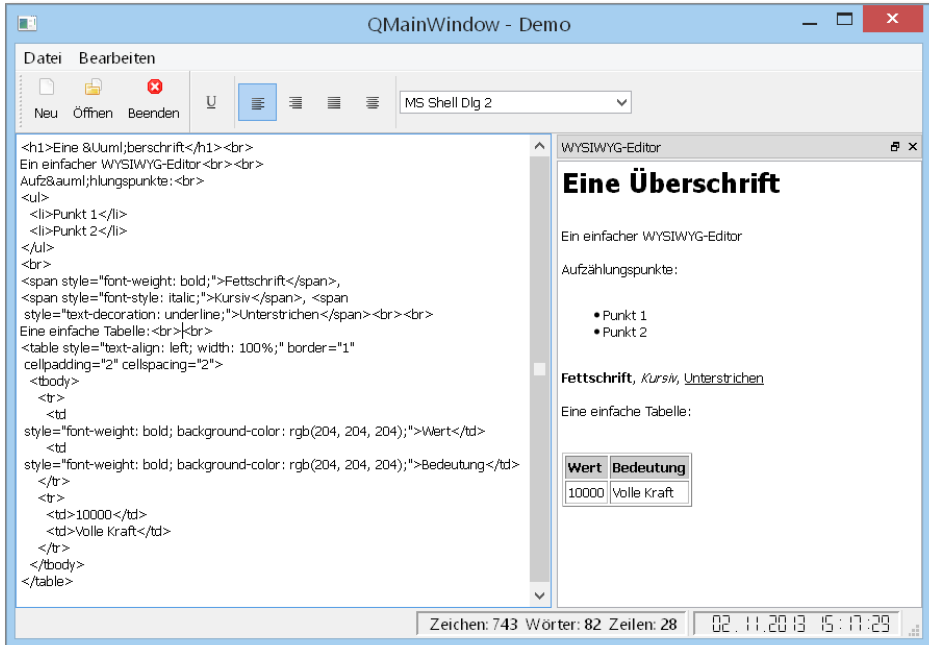


Abbildung 17.152 Der Texteditor jetzt mit einem an- und abdockbaren Widget



Abbildung 17.153 Hier wurde das Widget abgedockt.

17.11.6 Einstellungen sichern mit QSettings

Die Klasse `QSettings` wird verwendet, um diverse Einstellungen der Anwendung zu speichern. Sicherlich kann man so etwas auch per Hand mit sogenannten *config*- oder auch *ini*-Dateien machen. Aber Qt bietet hierzu eine Vollservice-Lösung an. Jeder, der schon mal umfangreichere Programme auf mehreren Systemen geschrieben hat, weiß, dass es ziemlich aufwendig werden kann, die Einstellungen an die Gegebenheiten der verschiedenen Systeme anzupassen. Da kommen Dinge wie systemweite und/oder benutzerdefinierte und/oder anwenderbezogene Einstellungen, Zugriffsrechte usw. vor.

Das Ziel der Klasse `QSettings` ist dabei ganz klar, ein portable Einstellung für die Qt-Anwendung zu ermöglichen. So werden Anwendungsdaten auf den unterschiedlichen Systemen gewöhnlich auch unterschiedlich gespeichert. MS-Windows benutzt hierfür bspw. die Registry, bei Mac OS X wird auf die XML-basierte Datei mit der Endung *.plist* zurückgegriffen, und unter Unix/Linux sind hierbei gleich mehrere Varianten (bspw. unter */etc/xdg* für systemweite oder *~/.config* für benutzerdefinierte Einstellungen) bekannt (teilweise abhängig von der Distribution). Um sich eben nicht mit solchen Dingen herumzuschlagen (eher zu ärgern), verwendet man einfach `QSettings`.

Zum Anlegen eines neuen `QSettings`-Objekts genügen im Grunde zwei Parameter:

```
QSettings* settings=new QSettings(
    tr("J.Wolf"),tr("JWTexteditor"));
```

Mit dem ersten Parameter gibt man die Firma oder den Namen an, und der zweite Parameter ist gewöhnlich für den Namen der Anwendung reserviert. Welcher Eintrag bei der Instanziierung auf den unterschiedlichen Systemen jeweils hinzugefügt würde, zeigen Abbildung 17.154 bis Abbildung 17.156.

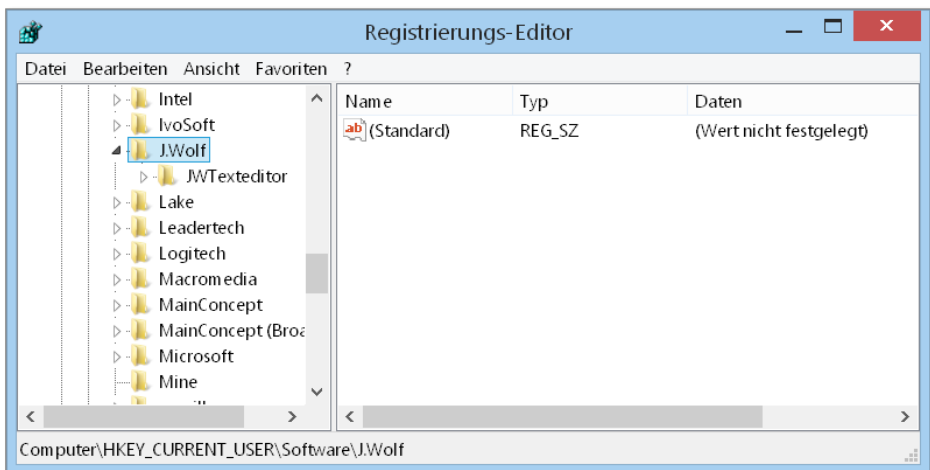


Abbildung 17.154 Neuer Registry-Eintrag unter MS-Windows in HKEY_CURRENT_USER\Software

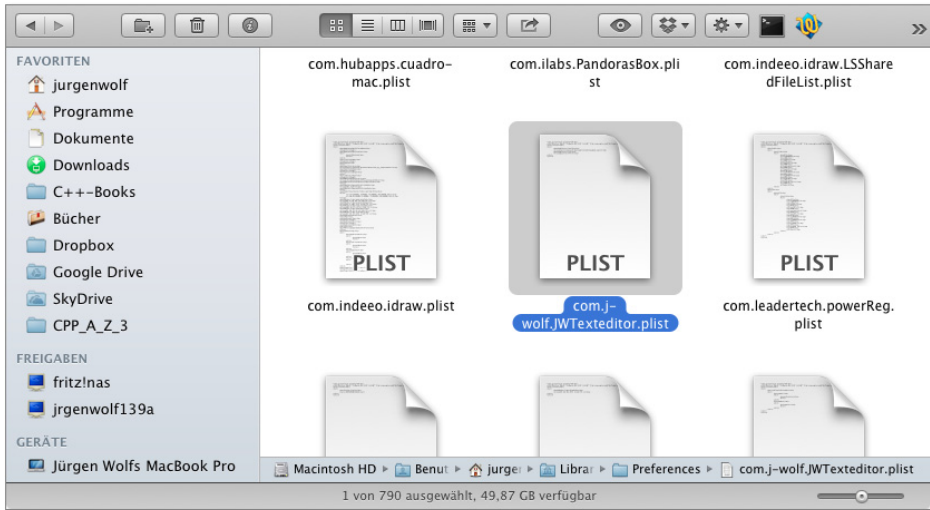


Abbildung 17.155 Neuer XML-Preference-Eintrag unter Mac OS X in `$HOME/Library/Preferences`

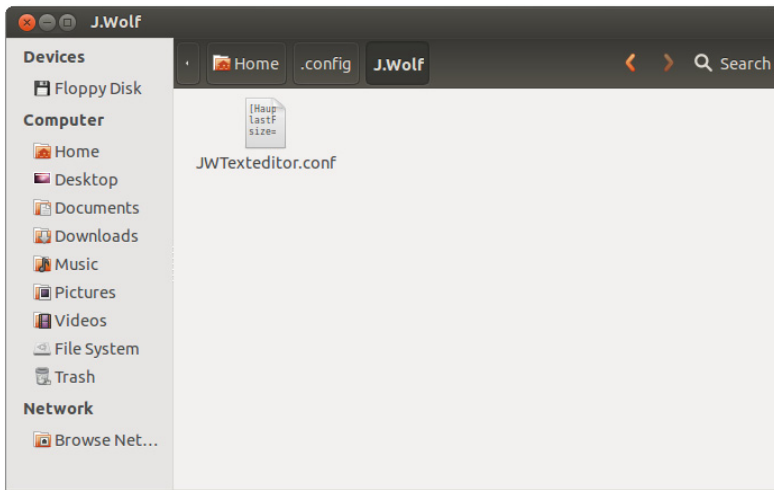


Abbildung 17.156 Neuer Eintrag unter Linux in `$HOME/.config`

Das sind hier nur gängige Möglichkeiten, wo QSettings seine Einträge speichert. Mehr Details dazu finden Sie wie immer in der Dokumentation zu QSettings mithilfe des Assistenten von Qt.

Standardmäßig werden unsere Einstellungen nur unter dem jeweiligen Benutzer entsprechend gespeichert, unter dem man eingeloggt ist (QSettings::UserScope). Benötigen Sie Einstellungen, die systemweit für jeden Benutzer zugänglich sind, also der kompletten Anwendung entsprechen (unter MS-Windows bspw. `HKEY_LOCAL_`

MACHINE\Software), kann man bei der Instanziierung des `QSettings`-Objekts die Konstante `QSettings::SystemScope` verwenden:

```
QSettings* settings = new QSettings(
    QSettings::SystemScope, tr("J.Wolf"), tr("JWTexteditor"));
```

Admin only

Bitte beachten Sie, dass Einstellungen mit `QSettings::SystemScope` Administratorrechte voraussetzen.

Es gibt noch eine weitere komfortable Möglichkeit, um nicht immer wieder ein `QSettings`-Objekt erzeugen zu müssen. Hierzu schreiben Sie (bspw. in der `main()`-Funktion):

```
QCoreApplication::setOrganizationName("J.Wolf");
QCoreApplication::setApplicationName("JWTexteditor");
```

Jetzt können Sie jederzeit eine parameterlose Instanziierung von `QSettings` verwenden. Daten werden bei `QSettings` mit der Methode `setValue()` geschrieben. Diese Methode erwartet neben dem Schlüssel auch einen Wert. Da der Wert von der Klasse `QVariant` ist, sind hierbei fast keine Grenzen gesetzt.

Im folgenden Beispiel wird z.B. beim Slot `MyWindow::openFile()` unseres selbst geschriebenen Texteditors die zuletzt geöffnete Datei gesetzt. Hierzu wird eine Gruppe namens »Hauptfenster« eingerichtet. Anschließend wird ein Schlüssel »last-File« mit dem Wert der geöffneten Datei gesetzt. Zum Schluss wird die neu angelegte Gruppe »Hauptfenster« wieder gelöscht:

```
...
QSettings* settings = new QSettings(
    tr("Pronix Inc."), tr("Qt-Texteditor"));
...
// Datei öffnen und im Editor anzeigen
void MyWindow::openFile() {
    QString fileName;
    fileName = QFileDialog::getOpenFileName(
        this, tr("Datei öffnen"), "",
        "C++ Dateien (*.cpp *.h);;"
        "Text-Dateien (*.txt);;"
        "Alle Dateien (*.*)" );
    if (!fileName.isEmpty()) {
        QFile file(fileName);
        if (file.open(QFile::ReadOnly | QFile::Text)) {
            editor->setPlainText(file.readAll());
            statusBar()->showMessage(
```

```

        tr("Datei erfolgreich geladen"));
    // Zuletzt geöffnete Datei setzen
    settings->beginGroup("HauptFenster");
    settings->setValue("lastFile", fileName );
    settings->endGroup();
}
}
}
...

```

Mit diesen Zeilen wird der neue Eintrag »lastFile« auf den verschiedenen Systemen hinzugefügt, wie in Abbildung 17.157 bis Abbildung 17.159 zu sehen.

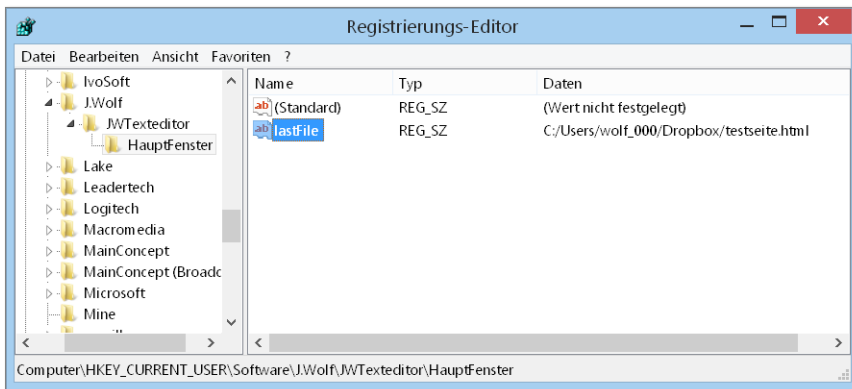


Abbildung 17.157 Neuer Eintrag mit »lastFile« in der Registry von Windows, wo die zuletzt geöffnete Datei gespeichert ist

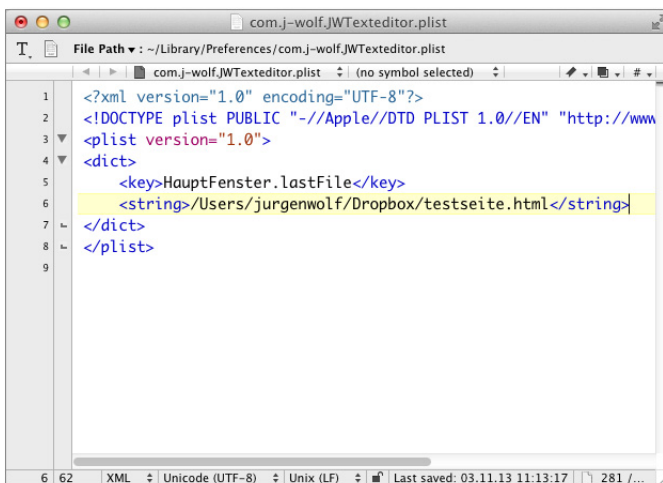


Abbildung 17.158 Der neue Eintrag »lastFile« als XML-Preference unter Mac OS X

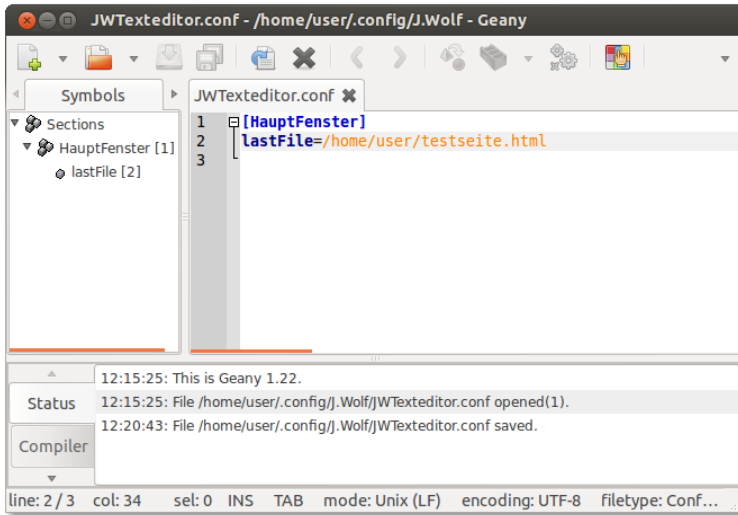


Abbildung 17.159 Der neue Eintrag »lastFile« unter Linux in .conf als typische Config-Datei

Selbstverständlich können Sie zu den Gruppen weitere Einstellungen speichern. Theoretisch ist es auch möglich, ohne Gruppe eine Einstellung zu machen. Sie können aber auch weitere Gruppen hinzufügen.

Auch ist es möglich, ohne Angabe der Gruppe mit der Methode `beginGroup()` auf eine Gruppe zuzugreifen. Hierbei können Sie im Grunde wie bei einer Pfadangabe über die Gruppe auf die entsprechende Einstellung zugreifen, bspw.:

```
settings->setValue("HauptFenster/lastFile", fileName );
```

Damit erreichen Sie das Gleiche, wie eben beschrieben wurde. Selbstverständlich sind Sie hierbei nicht nur auf Zeichenketten angewiesen (schließlich verwenden wir ja `QVariant`). Beispielsweise können Sie folgendermaßen einen Handler einrichten, der ausgeführt wird, wenn das Hauptfenster geschlossen wird:

```
...
void MyWindow::closeEvent(QCloseEvent *event) {
    settings->beginGroup("HauptFenster");
    settings->setValue("size", size() );
    settings->endGroup();
    event->accept();
}
...
```

Hierbei handelt es sich um eine `protected`-Methode der Klasse `QWidget` (`QWidget::closeEvent()`). Daher wird diese auch ausgeführt. Im Beispiel wird die Größe

des Fensters mithilfe der Methode `QWidget::size()` in den Schlüssel "size" geschrieben und als Eintrag der Gruppe "HauptFenster" hinzugefügt. In der Registry von MS-Windows ist dann folgender Eintrag aus Abbildung 17.160 enthalten.

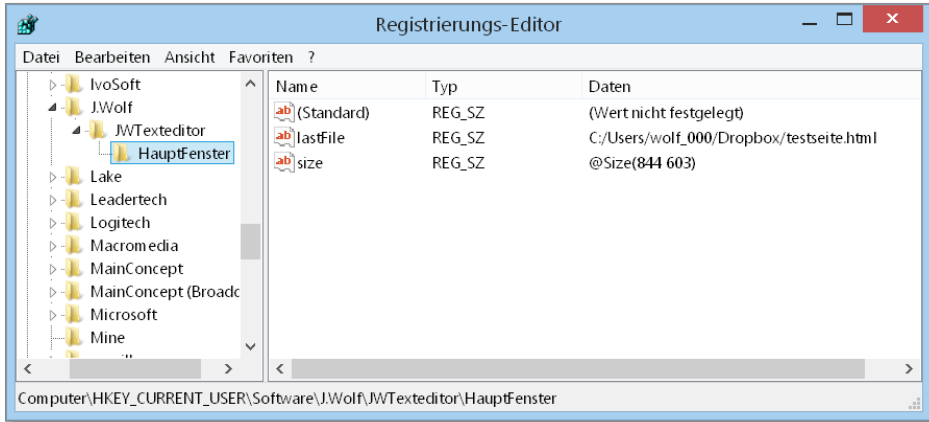


Abbildung 17.160 Beim Schließen des Hauptfensters wird die aktuelle Größe in der Registry gesichert, so dass beim erneuten Start der Anwendung derselbe Zustand wiederhergestellt werden kann.

Ähnlich wird auch ein Eintrag unter Mac OS X in der XML-Preference und bei Linux in der `.config`-Datei hinzugefügt.

Alle hier gesetzten Werte werden nun beim Start des Programms im Konstruktor überprüft und gegebenenfalls verwendet:

```
MyWindow::MyWindow(QMainWindow *parent,
                   Qt::WindowFlags flags):
    QMainWindow(parent, flags) {
...
    settings = new QSettings(
        tr("Pronix Inc."), tr("Qt-Texteditor"));
    settings->beginGroup("HauptFenster");
    // QVariant in einen QString konvertieren
    QString fileName = settings->value("lastFile").toString();
    // QVariant in ein QSize konvertieren
    QSize size = settings->value("size", sizeHint()).toSize();
    if (!fileName.isEmpty()) {
        QFile file(fileName);
        if (file.open(QFile::ReadOnly | QFile::Text)) {
            editor->setPlainText(file.readAll());
            statusBar()->showMessage(
                tr("Zuletzt geöffnete Datei "
                  "erfolgreich geladen") );
        }
    }
}
```



```

        // Wert wieder setzen
        settings->setValue("lastFile", fileName );
    }
    else // falls die Datei nicht mehr existiert
        settings->setValue("lastFile", "");
    }
    else // Keine Datei gesetzt
        settings->setValue("lastFile", "");
    settings->endGroup();
    // Fenstergröße in Einstellungen vorhanden?
    if( size.isNull() )
        resize(640, 480); // Nein
    else
        resize(size);      // Ja, dann setzen
...
}

```

17.11.7 Kompletter Quellcode des Texteditors

Das Listing zum hier erstellten Texteditor finden Sie auf der Buch-CD im Ordner *kapitel17/JWTexteditor*. Den fertigen Texteditor bei der Ausführung unter den verschiedenen Systemen zeigen Ihnen Abbildung 17.161 bis Abbildung 17.163.

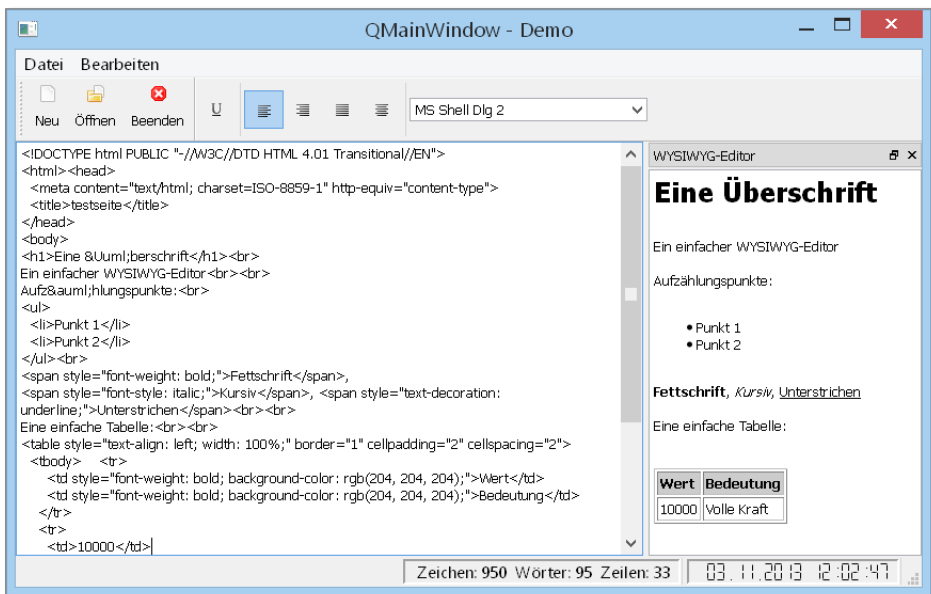


Abbildung 17.161 Der fertige Texteditor bei der Ausführung unter Windows

Wie Sie einen solchen Texteditor mit dem Designer von Qt zusammenklicken können, erfahren Sie in Abschnitt 17.13, »Anwendungen in einem Hauptfenster mit dem Qt Designer«.

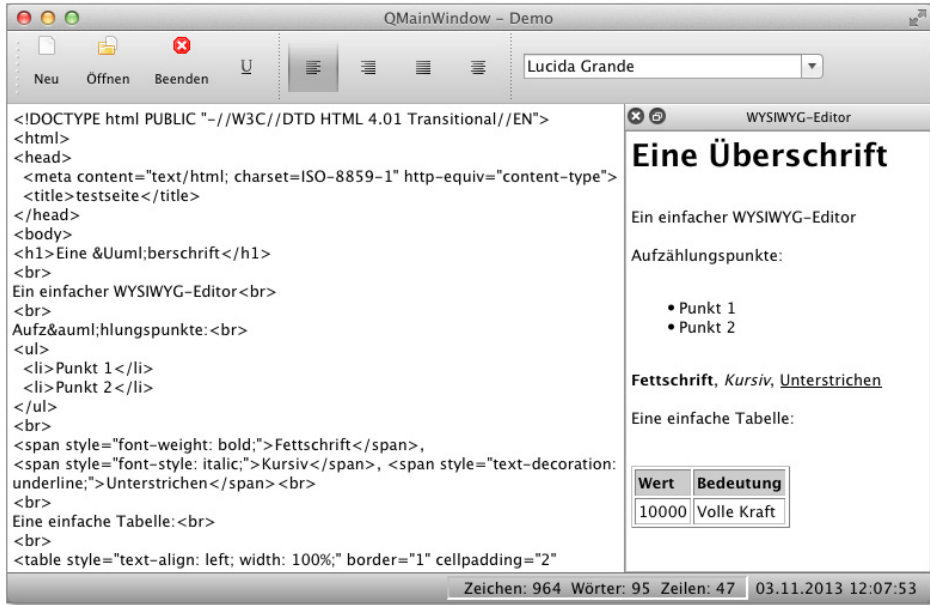


Abbildung 17.162 Der fertige Texteditor bei der Ausführung unter Mac OS X

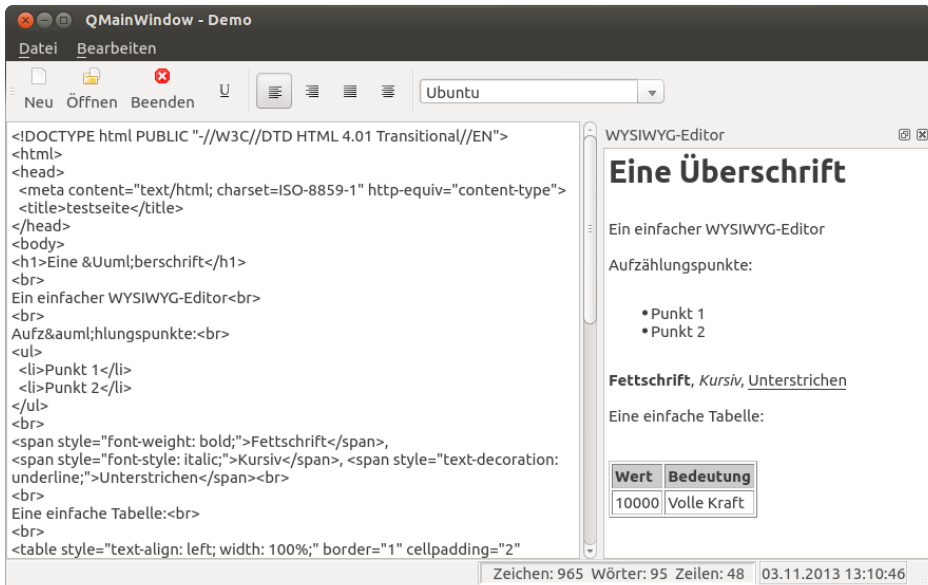


Abbildung 17.163 Der fertige Texteditor bei der Ausführung unter Linux

Fazit

Zugegeben, dies waren jetzt viele Informationen auf engstem Raum. Daher ist es unerlässlich, für die einzelnen Klassen den Assistenten von Qt zu verwenden. Das Beispiel hat aber sehr schön einzelne Komponenten einer typischen grafischen Oberfläche eines Hauptfensters demonstriert. In der Praxis werden Sie eine solche Anwendung eher selten komplett manuell im Editor schreiben und stattdessen in einer Kombination aus Designer und Editor Ihre Anwendungen entwickeln. Trotzdem ist es sehr wichtig, zu verstehen, welche Klassen da im Hintergrund arbeiten und wie Sie diese auch theoretisch manuell schreiben können.

17.12 Anwendung lokalisieren mit Qt Linguistic

Beim Listing im Texteditor zuvor wurde die statische Funktion `tr()` aus `QObject` für Strings verwendet, mit dem Hinweis, dass wir damit eine Anwendung in mehreren Sprachen lokalisieren können. Wie das funktioniert, soll in diesem Abschnitt anhand des Texteditors aus *kapitel17/JWTexteditor* gezeigt werden.

Zuerst müssen wir unsere Projektdatei (*.pro*) etwas anpassen bzw. um einen Eintrag `TRANSLATION` erweitern. Mit `TRANSLATION` können Sie die Übersetzungsdatei(en) als Argument eintragen. Im Beispiel wollen wir eine Übersetzungsdatei für die englische Sprache als Argument übergeben. Somit müssen Sie der Projektdatei folgenden Eintrag hinzufügen:

```
TRANSLATIONS = MyWindow_en.ts
```

Ebenso können Sie hierbei auch mehrere Übersetzungen auf einmal hinzufügen:

```
TRANSLATIONS = MyWindow_en.ts \
                MyWindow_it.ts \
                MyWindow_es.ts \
                MyWindow_fr.ts
```

Hier wurden der englischen Version noch zusätzlich eine Übersetzung für Italienisch, Spanisch und Französisch hinzugefügt.

In unserem Beispiel begnügen wir uns mit einer englischen Übersetzung. Daher sieht die Projektdatei (**.pro*) aus wie in Abbildung 17.164.

An dieser Stelle wurde allerdings noch keine Datei (hier *MyWindow_en.ts*) angelegt. Hierzu müssen Sie die Projektdatei mit einem Aufruf von `lupdate` aufrufen. Auf der Oberfläche von Qt Creator finden Sie diesen Befehl über **EXTRAS** **EXTERN** **LINGUIST** **ÜBERSETZUNG AKTUALISIEREN (LUPDATE)**.

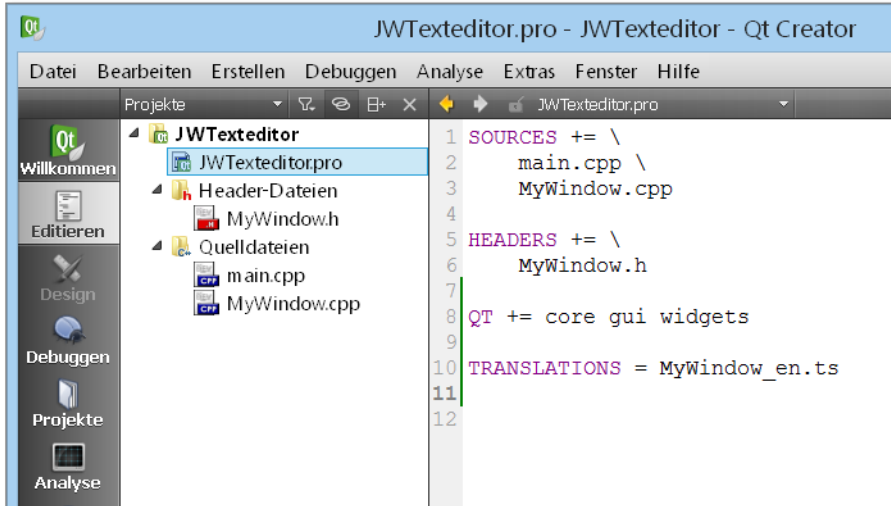


Abbildung 17.164 Hinzufügen einer Übersetzungsdatei

lupdate lässt sich nicht ausführen?

Wenn sich `lupdate` über das Menü **EXTRAS** nicht ausführen lässt, dann liegt das vermutlich an mangelnden Zugriffsrechten (wie das bspw. bei Windows der Fall sein kann). Für die Ausführung von `lupdate` sind Administratorrechte nötig. Abhilfe können Sie hier schaffen, indem Sie die Kommandozeile über Qt öffnen. Unter **Windows** finden Sie diese Kommandozeile über **START • ALLE PROGRAMME • QT** im selben Ordner vor wie den **Linguist** von Qt. Wechseln Sie dann mit `cd` in das entsprechende Verzeichnis, in der die Projektdatei liegt, und geben Sie in der Kommandozeile ein (*JWTexteditor.pro* sei die Projektdatei):

```
$> lupdate JWTexteditor.pro
```

Allerdings sind auch hierfür Administratorrechte nötig.

Jetzt finden Sie alle extrahierten Zeichenketten aus den Quelldateien, die Sie im Projekt (hier *JWTexteditor.pro*) zwischen `tr()` gestellt haben, in der Datei *MyWindow_en.ts* als XML-basiertes Format wieder. Hier ein Ausschnitt dieser Datei:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE TS>
<TS version="2.0" language="en_US">
<context>
  <name>MyWindow</name>
  <message>
    <location filename="MyWindow.cpp" line="22"/>
```

```

        <source>&Dati</source>
        <translation type="unfinished"></translation>
    </message>
    <message>
        <location filename="MyWindow.cpp" line="25"/>
        <source>&Neu</source>
        <translation type="unfinished"></translation>
    </message>
    ...

```

Sofern Sie jetzt dem Quellcode neue Strings mit `tr()` hinzufügen, dann müssen Sie, wie eben gezeigt, mittels `lupdate JWTexteditor.pro` bzw. über das Menü **EXTRAS EXTERN LINGUIST ÜBERSETZUNG AKTUALISIEREN (LUPDATE)** die Übersetzungsdatei immer wieder auf den neuesten Stand bringen.

Fehlervermeidung bei der Übersetzung

Jetzt haben Sie die Vorbereitungen für eine Übersetzung getroffen. Trotzdem muss ich Sie hierbei noch auf eine Fehlerquelle aufmerksam machen, die mit der statischen Methode `QObject::tr()` auftreten kann. Ein Problem bei der Verwendung von `QObject::tr()` kommt besonders gerne bei dynamischen Strings vor, wenn man nicht aufpasst. Zum Beispiel bei folgendem String:

```
QString msg = tr("File " + filename + " not found!");
```

Da sich der String abhängig von `filename` ändert, wird dieser von `tr()` nicht korrekt übersetzt – hinter `filename` kann der Übersetzer keinen Text platzieren. In solchen Fällen müssen Sie auf `QString::arg()` und einen Platzhalter zurückgreifen. In diesem Fall sieht die Lösung also wie folgt aus:

```
QString msg = tr("File %1 not found!").arg(filename);
```

Wenn Sie sich außerdem die Syntax von `tr()` angesehen haben, werden Sie festgestellt haben, dass neben dem eigentlichen String auch noch zusätzlich ein Kommentar verwendet werden kann. Dies kann man bspw. bei zweideutigen Wörtern nutzen. Es befinden sich zwei Strings "Size" im Programm zum Übersetzen. Dies kann zweideutig sein. Einmal ist die Größe des Hauptfensters gemeint und einmal die Größe einer Dialogbox. Hier empfiehlt es sich, die Kommentare zu verwenden:

```

QString size1 = tr("Size", "the main window");
QString size2 = tr("Size", "the dialog box");

```

Nach einem Update mit `lupdate` stehen auch in der Übersetzungsdatei die beiden Strings mit den jeweiligen Hinweisen.

Übersetzen mit Linguist

Um jetzt die Übersetzungsquelle zu bearbeiten, wird in der Praxis gewöhnlich auf Linguist von Qt zurückgegriffen, welcher ebenfalls mit dem Qt Creator SDK mitgeliefert wurde. Das Programm hat schon den Vorteil, dass auch Nicht-Programmierer bzw. professionelle Übersetzer die Strings übersetzen können.

Starten Sie also den Linguist von Qt, und öffnen Sie die Datei mit der *.ts*-Endung. In unserem Fall die Datei *MyWindow_en.ts* (Abbildung 17.165).

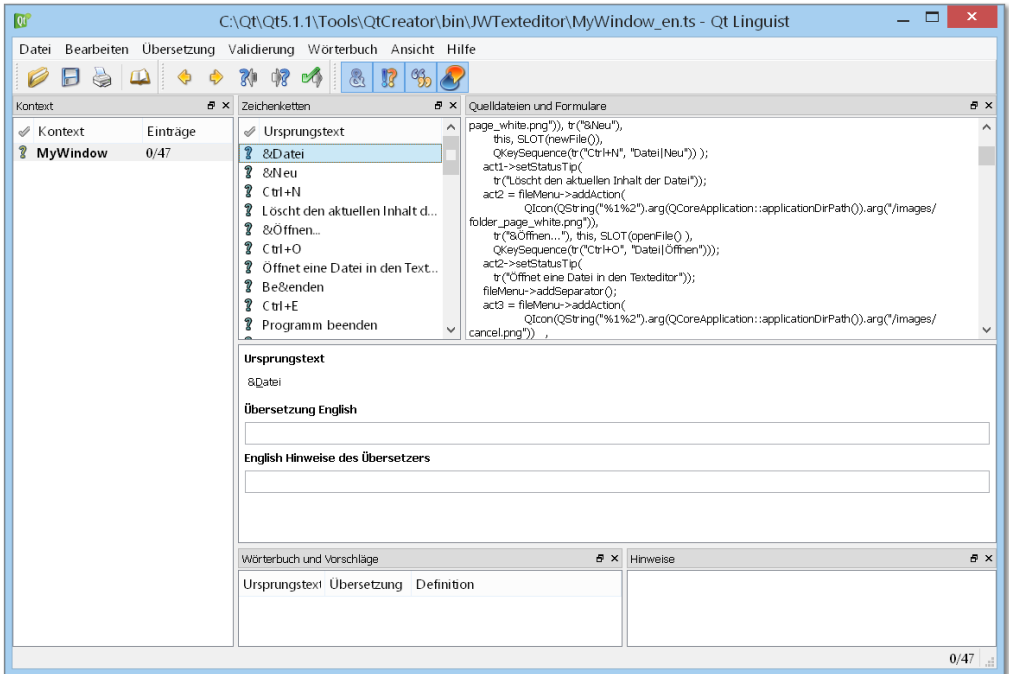


Abbildung 17.165 »MyWindow_en.ts« in den Linguist von Qt geladen

Im linken Dock-Fenster sehen Sie jetzt eine Übersicht des Übersetzungskontextes. Hier sehen Sie auch die Anzahl der Elemente und wie viele bisher übersetzt wurden. Im Beispiel haben wir 47 Strings und noch keinen davon übersetzt. Als KONTEXT wird gewöhnlich der Name der Klasse verwendet (hier MYWINDOW).

Wenn das Fragezeichen noch blau ist, bedeutet dies, dass der String noch nicht übersetzt wurde. Ein übersetzter String besitzt eine orange Farbe. Sollte etwas nicht ganz klar sein bei der Übersetzung (bspw. fehlende "... " oder &) markiert Linguist dieses Element mit einem roten Ausrufezeichen. Sind Sie mit der Übersetzung zufrieden, können Sie die orangefarbenen Fragezeichen anklicken, und es wird ein Häkchen daraus.

Um einen String zu übersetzen, müssen Sie diesen nur anklicken und können ihn auf der rechten Seite im Editfeld ÜBERSETZUNG eintippen. Der extrahierte Originaltext steht unter URSPRUNGSTEXT (siehe Abbildung 17.166).

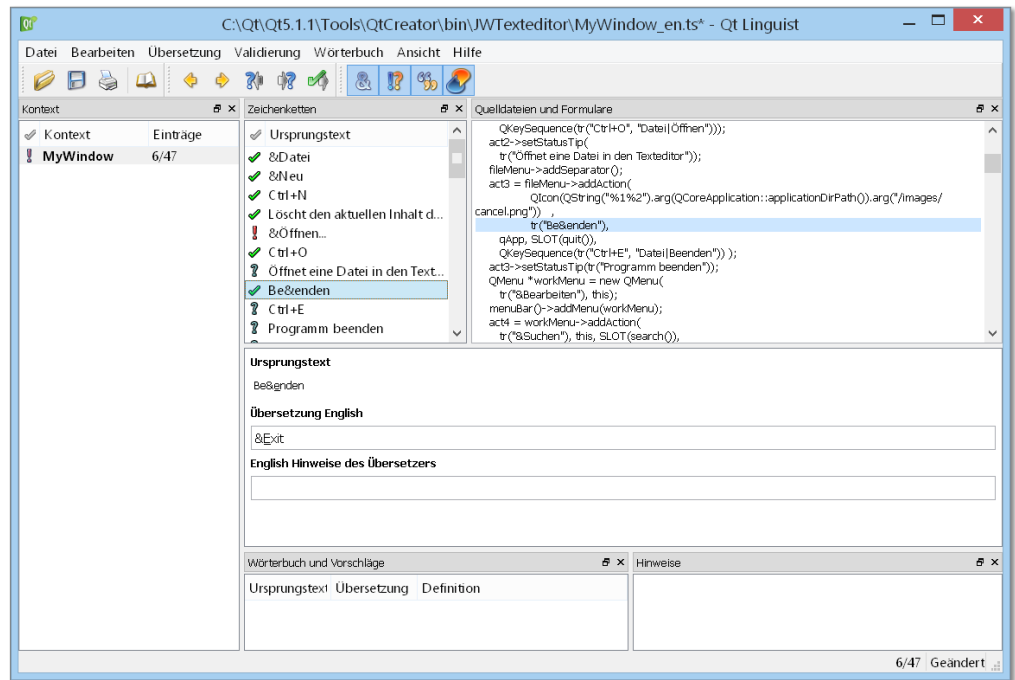


Abbildung 17.166 Einige übersetzte Elemente in Linguist

Natürlich hat der Linguist von Qt noch viel mehr Features, aber in diesem Abschnitt soll nur auf den Standardgebrauch eingegangen werden. Wenn Sie alle Übersetzungen gemacht haben, müssen Sie diese mit dem Linguist in der geöffneten *.ts*-Datei speichern.

Manuell übersetzen

Es ist auch möglich, eine Übersetzung ohne den Linguist von Qt zu machen. Hierzu müssen Sie dann den Quelltext von *MyWindow_en.ts* in einem Texteditor öffnen und die Übersetzung zwischen den Tags `<translation>...</translation>` selbst eingeben. Zum Beispiel:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE TS>
<TS version="2.0" language="en_US">
<context>
  <name>MyWindow</name>
  <message>
```

```

    <location filename="MyWindow.cpp" line="22"/>
    <source>&Datei</source>
    <translation>&File</translation>
</message>
<message>
    <location filename="MyWindow.cpp" line="25"/>
    <source>&Neu</source>
    <translation>&New</translation>
</message>
...

```

Zwischen den Tags `<source>...</source>` stehen die aus `tr()` extrahierten Strings, und zwischen `<translation>...</translation>` schreiben Sie die Übersetzung.

Binäre Übersetzungsdatei erstellen

Nachdem Sie die Datei `MyWindow_en.ts` mit dem Linguist oder manuell übersetzt und abgespeichert haben, müssen Sie aus der Datei eine Binärdatei erstellen, die Ihr Qt-Programm anschließend verwenden kann. Hierfür steht Ihnen das Kommando `lrelease` zur Verfügung, welches Sie über das Menü **EXTRAS • EXTERN • LINGUIST ÜBERSETZUNG FREIGEBEN (LRELEASE)** ausführen. Alternativ können Sie in der Kommandozeile Folgendes ausführen:

```

$> lrelease MyWindow.pro
Updating 'MyWindow_en.qm'...
    Generated 47 translations (47 finished and 0 unfinished)

```

Durch den Aufruf von `lrelease` finden Sie eine neue binäre Datei `MyWindow_en.qm` vor, welche jetzt alle Übersetzungen (hier auf Englisch) für unser Programm enthält. Diese Datei mit der Endung `*.qm` benötigen wir jetzt, um die Übersetzung in unserem Programm verwenden zu können.

Übersetzung verwenden

Um jetzt die Übersetzungen im Programm zu verwenden, müssen Sie auf die Klasse `QTranslator` zurückgreifen. Gewöhnlich erledigt man diesen Vorgang als Erstes in der `main()`-Funktion. Am besten sehen Sie sich die entsprechende `main()`-Funktion an:

```

00 // kapitel17/JWEditor_trans/main.cpp
01 #include <QApplication>
02 #include <QTranslator>
03 #include <QLocale>
04 #include "MyWindow.h"

```



```

05 int main(int argc, char *argv[]) {
06     QApplication app(argc, argv);
07     QTranslator editTranslator;
08     QString filename;
09     filename = QString("MyWindow_%1").arg(QLocale::system().name());
10     editTranslator.load(filename, qApp->applicationDirPath() );
11     app.installTranslator(&editTranslator);
12     MyWindow window;
13     window.show();
14     return app.exec();
15 }

```

Um in der Zeile 09 die entsprechende Übersetzungsdatei mit der Endung *.qm* zu finden, wird die statische Methode `QLocale::system()` verwendet, um Informationen über das lokale System zu erhalten. Zusätzlich mit der Methode `name()` erhalten wir dann einen String aus Sprachcode und Ländercode. Bei Deutschland wäre dies bspw. *de_DE*. In diesem Beispiel würde somit in `filename` der String *MyWindow_de_DE* stehen. Die Endung *.qm* wird von `QTranslator` automatisch angefügt.

In Zeile 10 wird versucht, die entsprechende Übersetzungsdatei `filename` zu laden. Als zweites Argument von `QTranslator::load()` geben wir das Verzeichnis an, wo sich die Übersetzungsdatei befindet. In diesem Fall im selben Verzeichnis wie die Anwendung. Der Algorithmus von `load()` gibt nicht gleich auf, wenn die Übersetzungsdatei *MyWindow_de_DE.qm* nicht vorhanden ist, sondern streicht von hinten zunächst die Endung und dann die Unterstriche raus bis zum letzten Unterstrich. Bspw. würden in diesem Fall nach folgenden Übersetzungsdateien der Reihe nach gesucht:

```

MyWindow_de_DE.qm
MyWindow_de_DE
MyWindow_de
MyWindow.qm
MyWindow

```

Sollte allerdings bis hierhin noch keine entsprechende Übersetzungsdatei gefunden worden sein, wird die im Quellcode benutzte Sprache verwendet. Wir wollen aber in unserem Fall ja eine englische Version aus der Deutschen machen. Hierzu könnten Sie bspw. die lokale Einstellung auf dem System ändern (auf *en_UK* oder *en_US*), oder aber Sie benennen für einen Test die Datei *MyWindow_en.qm* um in *MyWindow_de.qm*. Oder Sie schreiben einfach in Zeile 09 Folgendes:

```

09 filename = QString("MyWindow_en.qm");

```

Es ist immer schwer, einen Übersetzer für eine entsprechende Sprache zu finden, mit der man nichts am Hut hat. So kann es recht nützlich sein, wenn man dennoch Versionen für andere Sprachen wie Französisch, Spanisch, Italienisch etc. bereitstellt, indem man einfach eine Kopie der englischen Version macht und die Endung für den entsprechenden Sprachcode hinzufügt (bspw. *MyWindow_fr.qm* für Französisch). Sofern Sie mindestens eine englische Sprachversion verwenden, können Sie sicher sein, dass Ihr Programm in aller Welt verwendet werden kann.

Wenn möglich dann auf Englisch

In der Praxis sollte man im Quellcode am besten gleich Englisch verwenden und am Ende die extrahierten Strings ins Deutsche usw. übersetzen.

Nach dem Laden der entsprechenden Übersetzungsdatei müssen wir nur noch die Methode `installTranslator()` in Zeile 11 in unserer `QApplication`-Instanz aufrufen, und nach dem nächsten `show()`-Aufruf sollte die neu übersetzte Benutzeroberfläche angezeigt werden (siehe Abbildung 17.167).

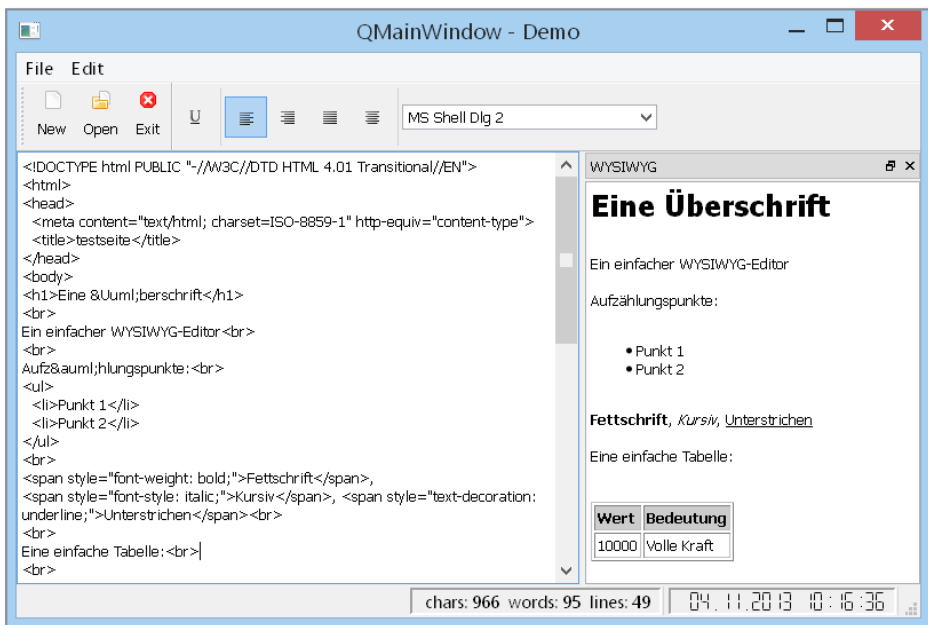


Abbildung 17.167 Ein in die englische Sprache übersetzter Editor

Verwendung von `tr()` erzwingen

Das Programm ist jetzt zwar übersetzt, aber meistens werden Programme immer weiterentwickelt, und schnell wird hierbei mal vergessen, einen String zwischen der statischen Methode `tr()` einzuschließen. Als Ergebnis erhält man dann oft ein halb

lokalisiertes Programm, welches zweierlei Sprachen aufweist. Es ist unschön, wenn im Menü ein Element wie »Ausschneiden« vorhanden ist und eine Zeile später ein Menüelement »Paste«. Um so etwas zu vermeiden, können Sie das Präprozessorsymbol `QT_NO_CAST_FROM_ASCII` verwenden. Dadurch erzwingen Sie, dass jeder String-Literal zwischen einem `tr()` gesetzt werden muss. Ist ein String nicht zwischen einem `tr()` gesetzt, gibt der Compiler einen Fehler zurück. Das Symbol setzen Sie in der Projektdatei (*.pro*) wie folgt:

```
DEFINES += QT_NO_CAST_FROM_ASCII
```

char-Arrays übersetzen

Wenn Sie in Ihrem Programm ein statisches `char`-Array angelegt haben, was ja keiner Klasse von Qt entspricht, so können Sie auch hier dafür sorgen, dass diese Zeichenketten extrahiert werden. Qt bietet hierzu das Makro `QT_TR_NOOP()` an. Zum Beispiel:

```
...
static const char* colors[] = {
    QT_TR_NOOP("red"),
    QT_TR_NOOP("green"),
    QT_TR_NOOP("blue"),
    0
};
...
```

Durch das Makro `QT_TR_NOOP()` wird einfach das Argument (also der String) dazwischen zurückgegeben. Dies reicht aber aus, dass ein Aufruf von `lupdate` auch die in `QT_TR_NOOP()` verpackten Strings extrahiert.

17.13 Anwendungen in einem Hauptfenster mit dem Qt Designer

Natürlich ist es auch möglich mit dem Designer ein Hauptfenster zu entwerfen, wie Sie dies in Abschnitt 17.9, »Eigenen Dialog mit dem Qt Designer erstellen«, mit einem Dialog gemacht haben. Auch will ich Ihnen diese Möglichkeit nicht vorenthalten und Ihnen demonstrieren, wie Sie einen kleinen Texteditor mit dem Designer als Hauptfenster zusammenklicken können.

1. Starten Sie ein neues leeres Qt-Projekt über das Menü `DATEI • NEU`, wie Sie dies in Abschnitt 17.1.2, »Ein einfaches Projekt erstellen«, in den Schritten 1 bis 5 gemacht haben.
2. Klicken Sie jetzt die Projektdatei mit der rechten Maustaste an, und wählen Sie im Kontextmenü `HINZUFÜGEN` aus (Abbildung 17.168).

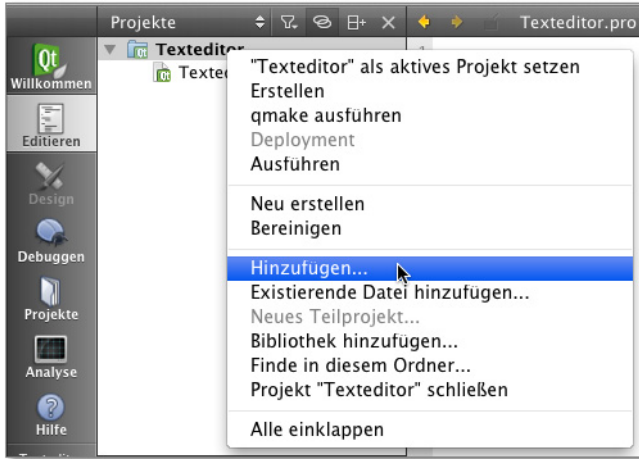


Abbildung 17.168 Etwas dem Projekt hinzufügen

3. Wählen Sie im nächsten Dialog unter den Vorlagen bei DATEIEN UND KLASSEN QT aus, so dass Sie in der Mitte QT-DESIGNER-FORMULARKLASSE auswählen können. Klicken Sie anschließend auf die Schaltfläche AUSWÄHLEN (siehe Abbildung 17.169).

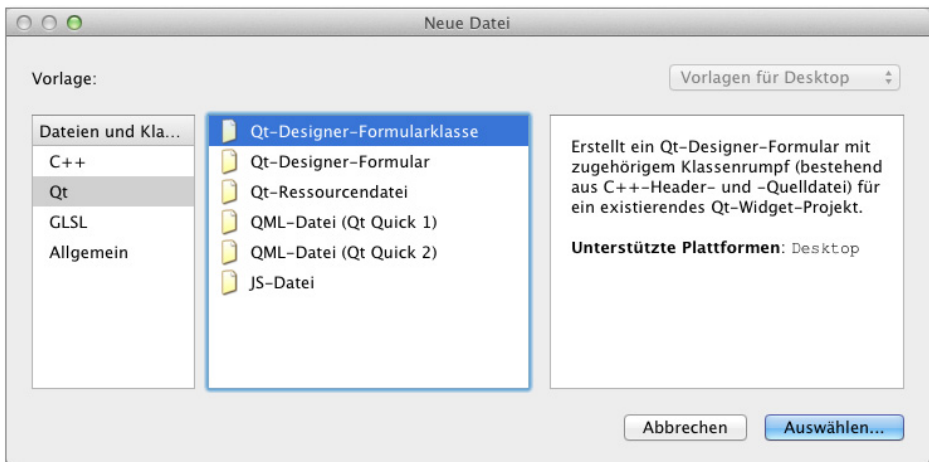


Abbildung 17.169 Eine neue Qt-Designer-Formularklasse erstellen

4. Im nächsten Schritt erscheint ein Fenster, in dem Sie auswählen können, was Sie erstellen wollen. In diesem Beispiel wollen wir ein Hauptfenster erstellen. Wählen Sie daher MAIN WINDOW aus. Klicken Sie dann auf die Schaltfläche WEITER (siehe Abbildung 17.170).

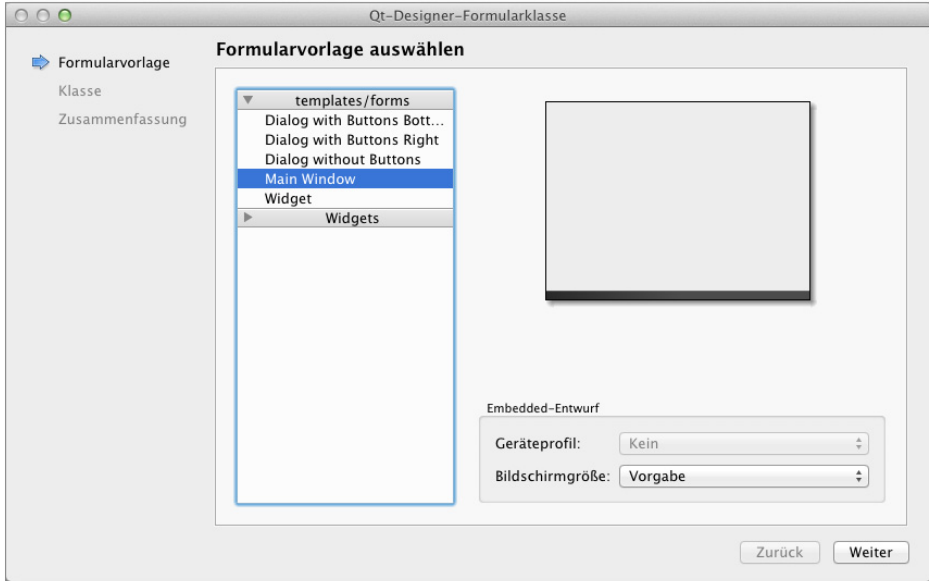


Abbildung 17.170 Ein Vorlage auswählen

5. Als Nächstes werden Sie nach dem Klassennamen, dem Namen der HEADER-DATEI, der QUELLDATEI, der FORM-DATEI (UI-Datei) und nach dem Speicherort für das künftige Dialogfenster gefragt (siehe Abbildung 17.171).



Abbildung 17.171 Alle Angaben für die künftige UI-Datei müssen hier ausgefüllt werden.

6. Am Ende erhalten Sie eine Zusammenfassung und können noch eine Projektverwaltung hinzufügen. Mit der Schaltfläche FERTIG sollten Sie eine Dialogbox im Modus DESIGN von Qt Creator vorfinden (siehe Abbildung 17.172).

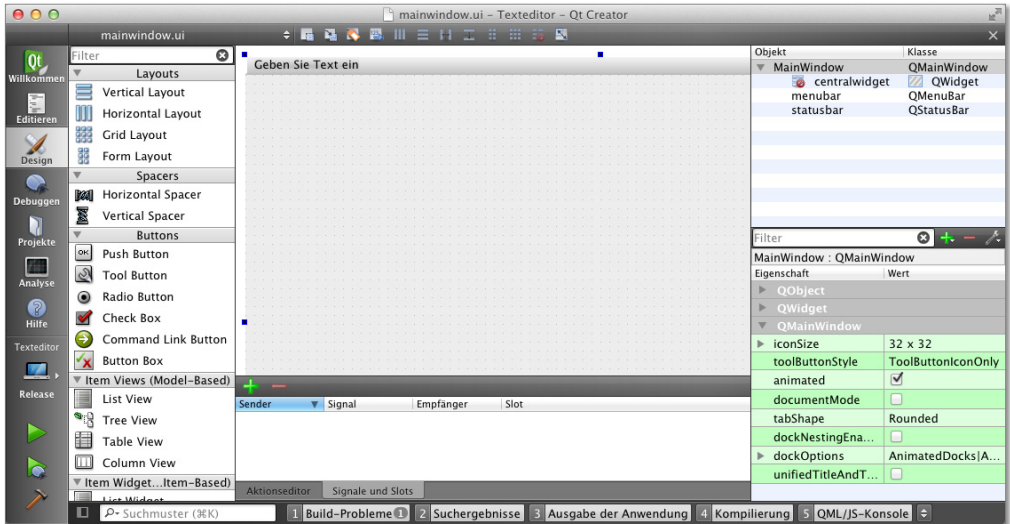


Abbildung 17.172 Ein leeres Hauptfenster im Design-Modus

Im Object Inspector können Sie sich gleich einen Überblick verschaffen, was der Designer alles einem Hauptfenster hinzugefügt hat (siehe Abbildung 17.173). Zunächst finden Sie hier das Zentral-Widget, eine Menüleiste und eine Statusbar.

Objekt	Klasse
▼ MainWindow	QMainWindow
centralwidget	QWidget
menubar	QMenuBar
statusbar	QStatusBar

Abbildung 17.173 Vorhandene Objekte beim Erstellen eines leeren Hauptfensters

Experimentieren erwünscht

Auch hier gilt wieder: Probieren geht über Studieren. Testen Sie die vielen Möglichkeiten, die Einstellungen eines Hauptfensters über den Eigenschaft-Editor zu verändern.

GUI-Elemente dem Hauptfenster hinzufügen

1. Als Nächstes wollen wir dem Hauptfenster eine Menüleiste hinzufügen. Wählen Sie dazu den Eintrag GEBEN SIE TEXT EIN bzw. TYPE HERE in der Menüleiste (Doppelklicken) (siehe Abbildung 17.174).

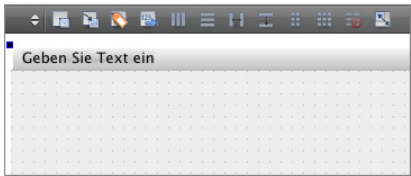


Abbildung 17.174 Die Menüleiste

2. Geben Sie jetzt den Titel ein, den das Menü haben soll. Im Beispiel erstellen wir zuerst das Menü »Datei« (siehe Abbildung 17.175). Mit einem Blick auf den Objekt Inspektor können Sie sich jederzeit einen Überblick über die Hierarchie verschaffen und (vor allem) behalten (siehe Abbildung 17.176).

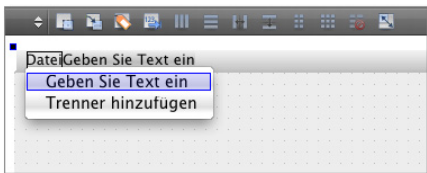


Abbildung 17.175 Menü »Datei«

Objekt	Klasse
MainWindow	QMainWindow
centralwidget	QWidget
menubar	QMenuBar
menuDatei	QMenu
statusbar	QStatusBar

Abbildung 17.176 Der Objekt-Inspektor mit dem neuen Menü »QMenu«

3. Um dem Element jetzt weitere Menüeinträge hinzuzufügen, brauchen Sie wiederum nur GEBEN SIE TEXT EIN bzw. TYPE HERE anzuklicken (siehe Abbildung 17.177). Im Beispiel wurden dem Menü DATEI die Aktionen ÖFFNEN, SPEICHERN und BEENDEN spendiert. Um einen Separator einzufügen, brauchen Sie nur auf TRENNER HINZUFÜGEN bzw. ADD SEPARATOR zu klicken.

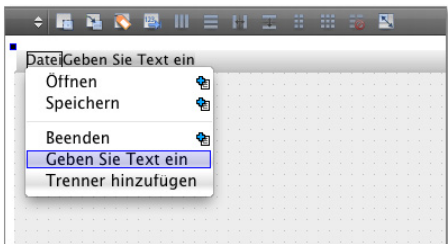


Abbildung 17.177 Neue Befehle (»QAction«) für das Menü

4. Natürlich können Sie den Aktionen auch Eigenschaften wie Tooltip, Statustipp, Tastaturkürzel usw. hinzufügen. Hierzu wählen Sie am besten über den Objekt-Inspektor die entsprechende Aktion aus (siehe Abbildung 17.178). Jetzt können Sie im Eigenschaft-Editor entsprechende Werte für die Eigenschaften vergeben, im Beispiel etwa das Tastenkürzel `[Strg]/[⌘]+[O]` für ÖFFNEN (siehe Abbildung 17.179).

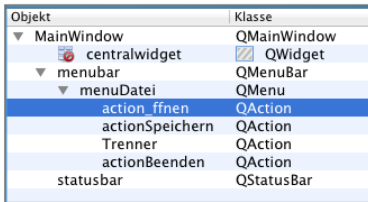


Abbildung 17.178 Befehle (»QAktion«) für das Menü auswählen ...

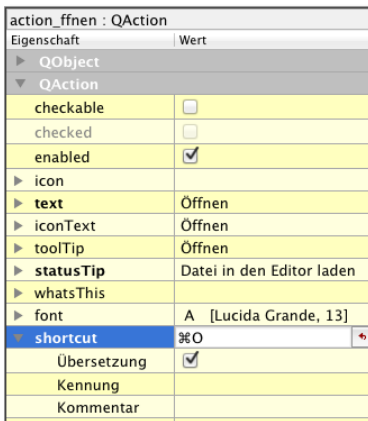


Abbildung 17.179 ... und einrichten, wie hier ein Tastenkürzel für »Öffnen«

5. Ebenso ist es auch möglich, bei den Menüeinträgen weitere Untermenüeinträge einzurichten. Hierzu müssten Sie nur auf das Plusymbol neben dem Menüeintrag klicken (siehe Abbildung 17.180). Im Beispiel wurde allerdings darauf verzichtet.

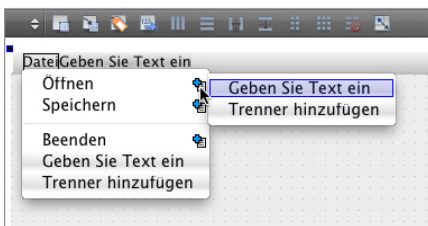


Abbildung 17.180 Weitere Untermenüeinträge lassen sich einrichten.

Beim Designer finden Sie auch einen Aktionseditor, der alle vorhandenen Aktionen des Hauptfensters anzeigt. Ist der Aktionseditor (Abbildung 17.181) nicht sichtbar, können Sie diesen über FENSTER • ANSICHTEN • AKTIONSEDITOR aktivieren. Hierbei können Sie gegebenenfalls auch gleich durch einen Doppelklick im Aktionseditor ein Icon vergeben, falls Sie das nicht bereits im Eigenschaft-Editor gemacht haben (Abbildung 17.182). Die Icons werden dann auch gleich bei den Menüeinträgen verwendet (siehe Abbildung 17.183).

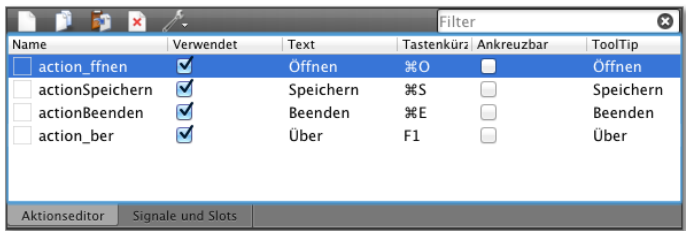


Abbildung 17.181 Übersicht über alle existierenden Befehle von »QAction«. Mit einem Doppelklick auf einen Eintrag ...

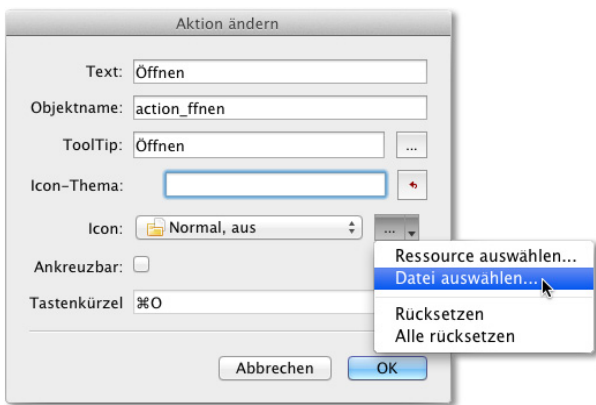


Abbildung 17.182 ... öffnet sich ein Dialog, mit dem Sie komfortabel den Befehl bearbeiten können. Hier wurde bspw. noch ein Icon hinzugefügt.

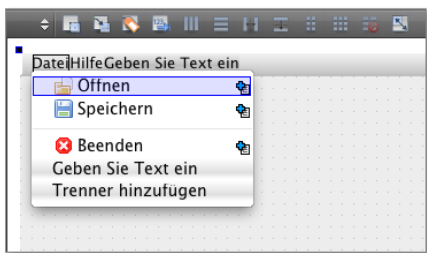


Abbildung 17.183 Icon als Menüeinträge

6. Das Hinzufügen einer Werkzeugleiste ist ebenfalls recht einfach (wenn man weiß wo). Hierzu müssen Sie zunächst in der Statusleiste unten mit der rechten Maustaste klicken und im Kontextmenü den Eintrag WERKZEUGLEISTE HINZUFÜGEN auswählen (siehe Abbildung 17.184). Jetzt finden Sie unter dem Menü eine schmale Leiste.

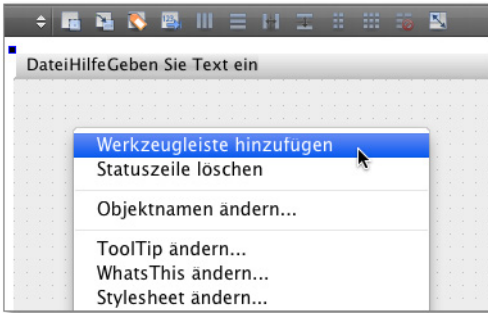


Abbildung 17.184 Eine leere Werkzeugleiste (QToolBar) hinzufügen

7. Da wir ja die Aktionen bereits mit QAction festgelegt haben, brauchen wir nur die gewünschten Aktionen der Werkzeugleiste vom Aktionseditor in die Werkzeugleiste zu ziehen (siehe Abbildung 17.185). Mit einem Rechtsklick in der Werkzeugleiste finden Sie auch ein Kontextmenü, über das Sie der Leiste einen Separator hinzufügen können.

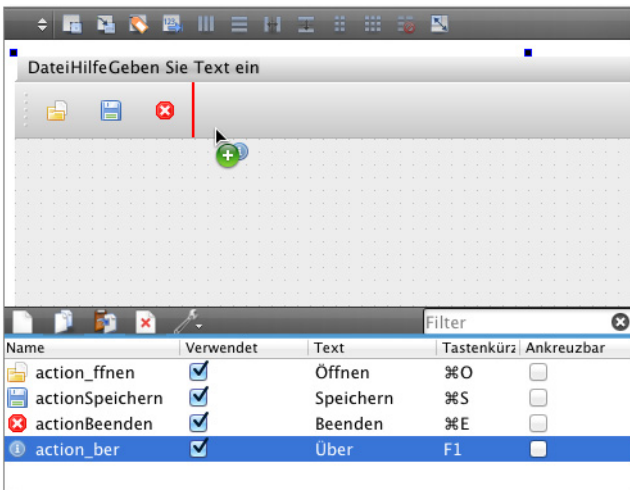


Abbildung 17.185 Fertige Befehle vom Typ »QAction« können über den Aktionseditor in die Werkzeugleiste gezogen werden.

Um den Überblick zu wahren, sollten Sie immer mal wieder einen Blick in den Objekt-Inspektor werfen (siehe Abbildung 17.186).

Objekt	Klasse
▼ MainWindow	QMainWindow
centralwidget	QWidget
▼ menubar	QMenuBar
▼ menuDatei	QMenu
action_ffnen	QAction
actionSpeichern	QAction
Trenner	QAction
actionBeenden	QAction
▼ menuHilfe	QMenu
action_ber	QAction
statusbar	QStatusBar
▼ toolBar	QToolBar
action_ffnen	QAction
actionSpeichern	QAction
Trenner	QAction
actionBeenden	QAction
Trenner	QAction
action_ber	QAction

Abbildung 17.186 Dank des Objekt-Inspektors bleibt die Übersicht auch dann erhalten, wenn das Projekt allmählich umfangreicher wird.

8. Zu guter Letzt wollen wir noch ein mehrzeiliges Textfeld als Zentral-Widget einfügen. Dieses Widget brauchen Sie nur von der Widget-Box in das Hauptfenster zu ziehen und dann die Größe noch entsprechend anzupassen (siehe Abbildung 17.187). Ein Blick auf den Objekt-Inspektor zeigt uns auch, dass QTextEdit unser zentrales Widget der Anwendung ist (siehe Abbildung 17.188).

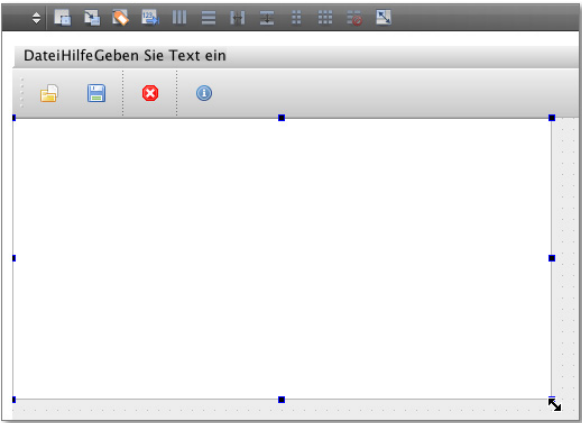


Abbildung 17.187 »QTextEdit« wird das Zentral-Widget unserer Anwendung, ...

Objekt	Klasse
▼ MainWindow	QMainWindow
centralwidget	QWidget
textEdit	QTextEdit
▼ menubar	QMenuBar
▼ menuDatei	QMenu
action_ffnen	QAction
actionSpeichern	QAction
Trenner	QAction
actionBeenden	QAction

Abbildung 17.188 ... was auch der Objekt-Inspektor bestätigt.

Mit dem Designer erstelltes Hauptfenster verwenden

Jetzt können Sie das erstellte Hauptfenster wieder im Projektverzeichnis abspeichern und daraus ein Programm erstellen, wie Sie dies schon im Abschnitt »Erstellten Dialog in einer Qt-Anwendung verwenden« in Abschnitt 17.9 gesehen haben.

Zunächst fügen Sie eine Headerdatei hinzu (hier *use_form.h*), welche Sie von *QMainWindow* und unserem mit dem Designer erstellten Hauptfenster (hier *Ui::MainWindow*) ableiten:

```
00 // kapitel17/mainForm/use_form.h
01 #ifndef MAINWINDOW_H
02 #define MAINWINDOW_H
03 #include <QFileDialog>
04 #include <QMessageBox>
05 #include "ui_mainwindow.h"

06 class MWindow : public QMainWindow, private Ui::MainWindow {
07     Q_OBJECT
08 public:
09     MWindow();
10     ~MWindow() {}
11 protected slots:
12     void openFile();
13     void saveFile();
14     void info();
15 };
16 #endif
```

Wenn Sie nicht mehr wissen, warum wir hier von *Ui::MainWindow* ableiten und woher die Headerdatei *ui_mainwindow.h* kommt, blättern Sie bitte zurück zu Abschnitt 17.9, »Eigenen Dialog mit dem Qt Designer erstellen«. Am Grundgerüst unserer Klasse *MWindow* können Sie sehen, dass wir hier drei Slots in den Zeilen 12 bis 14 einrichten, um einen Text in den Editor zu laden, den enthaltenen Text zu speichern und eine Information zur Anwendung einzublenden.

Jetzt zur Implementierung des Codes in *use_Form.cpp*, welche Sie jetzt neu dem Projekt hinzufügen, wobei auch hier wieder im Konstruktor zuerst *setupUi()* aufgerufen werden soll, damit Sie auf alle Membervariablen der Designerklasse einen Zugriff haben. Zusätzlich richten wir hierbei die Signal-Slot-Verbindungen ein und schreiben auch den Code für die entsprechenden Slots:

```

00 // kapitel17/mainForm/use_form.cpp
01 #include "use_form.h"

02 MWindow::MWindow() {
03     setupUi(this);
04     // Signal-Slot-Verbindungen
05     connect( actionBeenden, SIGNAL(triggered(bool)),
06             qApp, SLOT(quit()));
07     connect( action_ffnen, SIGNAL(triggered(bool)),
08             this, SLOT(openFile()));
09     connect( actionSpeichern, SIGNAL(triggered(bool)),
10             this, SLOT(saveFile()));
11     connect( action_ber, SIGNAL(triggered(bool)),
12             this, SLOT(info()));
13 }

14 void MWindow::openFile() {
15     QString fileName = QFileDialog::getOpenFileName(this);
16     QFile file(fileName);
17     if( file.open(QIODevice::ReadOnly|QIODevice::Text)) {
18         textEdit->setPlainText(
19             QString::fromUtf8(file.readAll() ) );
20         statusBar()->showMessage(
21             tr("Datei erfolgreich geladen"), 5000);
22     }
23 }

24 void MWindow::saveFile() {
25     QString fileName = QFileDialog::getSaveFileName(this);
26     if( fileName.isEmpty() ) return;
27     QFile file(fileName);
28     if( file.open(QIODevice::WriteOnly|QIODevice::Text) ){
29         file.write(textEdit->toPlainText().toUtf8());
30         statusBar()->showMessage(
31             tr("Datei erfolgreich gespeichert"), 5000);
32     }
33 }

34 void MWindow::info() {
35     QMessageBox::information( 0, tr("Über..."),
36         tr( "Ein kleiner Texteditor\n"
37            "der mit dem Designer erstellt wurde"));
38 }

```

Zum Schluss fehlt uns nur noch eine Hauptfunktion, um das Hauptfenster auszuführen. Der Code hierzu sieht wie folgt aus:

```
00 // kapitel17/mainForm/main.cpp
01 #include "use_form.h"
02 #include <QApplication>

03 int main(int argc, char*argv[] ) {
04     QApplication app(argc, argv);
05     MWindow window;
06     window.show();
07     return app.exec();
08 }
```

Fehlende Bibliotheken in der .pro-Datei

Und auch hier erneut zur Erinnerung: Beim Erstellen eines Qt-Projekts kann es passieren, dass qmake die nötigen Qt-Module nicht hinzulinkt. In unserem Beispiel würden hierzu die Module *Qt Core*, *Qt GUI* und *Qt Widgets* benötigt. Sollten Sie Probleme mit dem Übersetzen haben, brauchen Sie einfach nur in der *.pro*-Datei folgende Zeile einzufügen:

```
QT      += core gui widgets
```

Abbildung 17.189 zeigt das Programm bei der Ausführung.

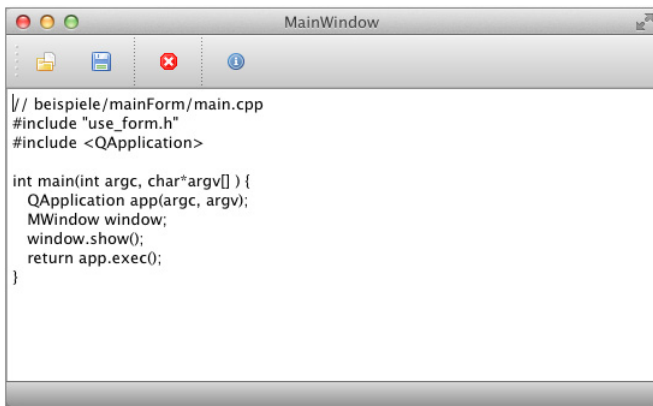


Abbildung 17.189 Der Texteditor bei der Ausführung

Anwendung lokalisieren

Natürlich lässt sich diese Anwendung genauso einfach lokalisieren, wie Sie dies in Abschnitt 17.12, »Anwendung lokalisieren mit Qt Linguistic«, erfahren haben.

17.14 Qt Designer vs. handgeschrieben

Sie haben in den vorangegangenen Kapiteln erfahren, wie Sie eigene Widgets, Dialoge und jetzt auch noch ein Hauptfenster von Hand schreiben können. Ebenso wurde Ihnen aber auch bei allen Beispielen der Weg mit dem Designer aufgezeigt. Wenn man ein Projekt mit einer grafischen Oberfläche erstellt, kommt zunächst immer erst die Frage auf, ob man es von Hand programmieren soll oder ein grafisches Werkzeug verwendet, um eine Anwendung zu erstellen.

Diese Frage kann nicht pauschal beantwortet werden, weil dies auch vom Projekt abhängt. Auch wenn der Designer ein tolles Werkzeug ist, wird man dennoch niemals ohne selbst programmierten Code auskommen. Es bietet sich auch an, eine Kombination aus beidem zu verwenden, das Grundgerüst von Hand zu erstellen und die nötigen Dialoge bzw. Widgets mit dem Designer zu erstellen und dem Projekt hinzuzufügen – oder eben andersherum.

Hier ein paar Vorteile, die Sie mit dem Qt Designer haben könnten:

- ▶ Sie sparen eine Menge an Zeit, um das Layout des Programms zu verwalten. Das Ausrichten und Anordnen der Widgets ist mit ein paar Mausklicks erledigt. Gerade wenn Sie ein Layout in ein anderes Layout verpacken, ist dies mit dem Designer ein Kinderspiel. Zudem als Einsteiger, wenn man noch nicht mit Qt vertraut ist, kann das Layout eine Menge an Zeit in Anspruch nehmen.
- ▶ Als Einsteiger in Qt können Sie außerdem sehr schön die gängigsten vorhandenen Widgets entdecken. Sie sehen, wie diese heißen, lernen deren Eigenschaften kennen, die Sie hierfür setzen können. Auch bekommen Sie durch den Designer einen guten Überblick, wie eine GUI-Anwendung mit Qt hierarchisch aufgebaut wird, wenn Sie einen Blick in den Objekt-Inspektor werfen.

Für Code, den Sie von Hand schreiben, könnten sich folgende Vorteile ergeben:

- ▶ Sie haben mehr Kontrolle, wenn Sie bspw. Dinge in einer bestimmten Reihenfolge oder etwas dynamisch zur Laufzeit erledigen wollen.
- ▶ Es ist einfach, Widgets zur Laufzeit hinzuzufügen und wieder zu entfernen.
- ▶ Wenn Sie benutzerdefinierte Widgets erstellen wollen, welche spezielle Parameter im Konstruktor enthalten, die jenseits eines QWidgetts gehen, dann lässt sich dies mit dem Designer nicht realisieren.

17.15 Dinge, die man wissen sollte ...

An dieser Stelle folgen noch ein paar Themen, über die Sie unbedingt Bescheid wissen sollten. Selbstverständlich gibt es darüber hinaus noch viel mehr, das man wissen sollte, aber dieses komplette Kapitel 17 stellt nur eine Einführung in Qt dar.

17.15.1 QApplication und QApplication

Die von `QCoreApplication` abgeleitete Klasse `QApplication` verwaltet den GUI-Kontrollfluss der Anwendung und Haupteinstellungen. Für eine Anwendung mit einem GUI ist hierbei genau ein `QApplication`-Objekt nötig, egal, wie viele Fenster die Anwendung hat. Die Klasse enthält die Haupt-Ereignisschleife (*main event loop*), in der alle Ereignisse des Fenstersystems und andere Quellen abgearbeitet und abgefertigt werden.

Um einen direkten Zugriff auf das `QApplication`-Objekt zu erhalten, können Sie die Methode `instance()` verwenden oder von überall den globalen Zeiger `qApp`.

Die Klasse `QApplication` ist für viele Dinge verantwortlich, wie Systemeinstellungen, Ereignisverarbeitung, GUI-Stile, Farben, Texte lokalisieren, fortgeschrittene Cursorverarbeitung, Synchronisation des Fenstersystems, Session-Verwaltung, Kommandozeilen-Parameter verarbeiten und noch eine Menge mehr. Ein Blick in die Dokumentation zu dieser Klasse fördert eine Menge an nützlichen Methoden zutage.

17.15.2 Konsolenanwendungen mit Qt

Für Qt-Anwendungen, die kein GUI benötigen, sollten Sie die Elternklasse `QCoreApplication` verwenden, weil diese Klasse nicht von der Bibliothek `QtGui` abhängig ist. `QCoreApplication` enthält, falls nötig, auch eine Haupt-Ereignisschleife `exec()` (bspw. bei typischen Client-/Server-Anwendungen), in der alle Ereignisse des Betriebssystems (wie bspw. Timer oder Netzwerk-Ereignisse) verarbeitet und abgefertigt werden. Auch hier finden Sie wieder Methoden zu system- und anwendungsweiten Einstellungen. Auch die Kommandozeile lässt sich mit `arguments()` relativ einfach auswerten.

Zeitpunkt der Erzeugung

In der Praxis wird immer empfohlen, ein `QCoreApplication`- oder ein `QApplication`-Objekt in der `main()`-Funktion sobald wie möglich zu erzeugen.

Wer bereits versucht hat, Konsolen-Anwendungen mit Qt zu erstellen, der dürfte sich besonders unter Windows das eine oder andere Mal die Zähne daran ausgebissen haben. Verwenden wir als Beispiel das folgende Listing:

```
00 // kapitel17/cmdline/main.cpp
01 #include <QCoreApplication>
02 #include <QStack>
03 #include <QStringList>
```



```

04 #include <iostream>
05 using namespace std;

06 int main(int argc, char *argv[]) {
07     QApplication app(argc, argv);
08     QStringList args = app.arguments();
09     QStringList::const_iterator constIt;
10     cout << "Argumente der Kommandozeile:\n";
11     for ( constIt = args.constBegin();
12           constIt != args.constEnd();
13           ++constIt )
14         cout << (*constIt).toLocal8Bit().constData()
15              << endl;
16     cout << "\nAusgabe des Stack:\n";
17     QStack <int> stack;
18     stack.push(5);
19     stack.push(4);
20     stack.push(3);
21     while( !stack.isEmpty())
22         cout << stack.pop() << "\n";
23     return 0;
24 }

```

Wenn das Beispiel ohne Vorkehrungen übersetzt und von der Kommandozeile aus gestartet wurde, gab es keinerlei Ausgabe auf dem Bildschirm. Hierzu müssen Sie in der Projektdatei (*pro*) noch ein paar Vorkehrungen treffen. Zunächst teilen wir dem Compiler mit, dass wir die Konsole verwenden wollen, und dann entfernen wir das GUI (QtGui) als Bibliothek unserer Anwendung. Hier müssen Sie gegebenenfalls folgende Einträge der Projektdatei hinzufügen:

```

QT += core
QT -= gui
CONFIG += console
SOURCES += main.cpp

```

Alternativ können Sie auch beim neuen Erstellen eines Projekts eine Qt-Konsolenanwendung auswählen.

Jetzt sollte es auch mit der Ausgabe des Programms klappen, wenn Sie die Anwendung in der Kommandozeile aufgerufen und gestartet haben (Abbildung 17.190).

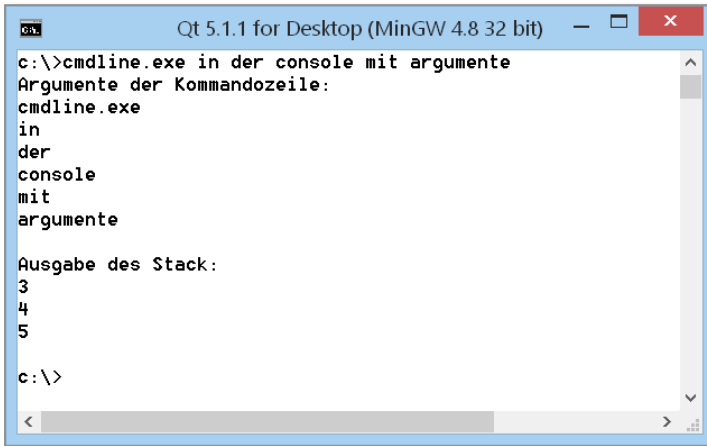


Abbildung 17.190 Qt-Konsolenanwendung bei der Ausführung

17.15.3 Wenn keine Icons angezeigt werden ...

Unter Linux/Unix macht es bspw. einen Unterschied, ob Sie das Programm von der Kommandozeile oder mit einem Mausklick auf das Icon starten, da es sich hierbei um zweierlei Umgebungen mit unterschiedlichen Werten handelt.

Der Unterschied macht sich erst bemerkbar, wenn Sie externe Dateien, wie bspw. Grafiken, beim Programm verwenden und dynamisch zur Laufzeit nachladen wollen. Was beim Starten aus der Kommandozeile noch angezeigt wurde, wird beim Mausklick auf der Anwendung nicht mehr angezeigt, weil hier der Wert für das Anwendungsverzeichnis ein anderer ist.

Fügen Sie bspw. einem Button ein Icon wie folgt hinzu:

```
button->setIcon("./images/icon.png");
```

In diesem Beispiel wird davon ausgegangen, dass sich das Icon im aktuellen Anwendungsverzeichnis (= dort, wo sich das ausführbare Programm befindet) in einem Verzeichnis `images` befindet. Dies trifft zu, wenn Sie, wie bereits erwähnt, das Programm aus der Kommandozeile starten. Beim Mausklick ist allerdings das aktuelle Anwendungsverzeichnis nicht mehr dasselbe und so kann logischerweise das Icon nicht mehr angezeigt werden.

Ein portabler und möglicher Weg, das Problem zu umgehen, ist es, das aktuelle Anwendungsverzeichnis direkt im Programm mit anzugeben und den Pfad zur Grafik hinzuzufügen. Sie ermitteln quasi zur Laufzeit des Programms den absoluten Pfad zur Anwendung. So etwas wird mit der statischen Methode `QCoreApplication::applicationDirPath()` ermittelt.

Zusammengebastelt würde ein solcher zur Laufzeit angegebener dynamischer Pfad wie folgt aussehen:

```
button01->setIcon( QIcon(QString("%1%2")
    .arg(QCoreApplication::applicationDirPath())
    .arg("/images/icon.png") ) );
```

Anwendungsverzeichnis bei Mac OS X

Bei Mac OS X müssen Sie etwas anders vorgehen, um die Icons ins Anwendungsverzeichnis zu legen. Nehmen wir an, die ausführbare Datei heißt hier *meinProgramm*. Hier müssen Sie jetzt *meinProgramm* mit der rechten Maustaste anklicken und im Kontextmenü den Befehl **PAKETINHALT ANZEIGEN** aufrufen. Hier hangeln Sie sich jetzt durch die Verzeichnisse über **CONTENTS/MACOS** und finden dort erst die eigentliche ausführbare Datei *meinProgramm*. Dies ist das eigentliche Anwendungsverzeichnis von *meinProgramm*, und dort legen Sie jetzt auch die Icons bzw. den Ordner mit den Icons ab.

Eine zweite Möglichkeit, wenn Sie Grafiken (oder auch andere Dateien) nicht zur Laufzeit laden wollen, ist der Ressourcenmechanismus von Qt. Dieser Mechanismus wird im folgenden Abschnitt beschrieben.

17.15.4 Das Ressourcen-System

Ressourcen-Verwaltung ist ja nichts Neues mehr und wird auch von vielen anderen GUI-Frameworks recht ähnlich verwendet. Mit einer Ressource haben Sie die Möglichkeit, binäre Dateien (bspw. Grafiken) in die ausführbare Datei einzubinden, um Sie hier direkt zu verwenden. Meistens werden diese Dateien in einem Unterverzeichnis gespeichert.

Um dieses Ressourcen-System überhaupt verwenden zu können, benötigen Sie eine Ressourcendatei, die in Qt mit der Dateiendung *.qrc* versehen wird. Die Ressourcendatei verwendet ein einfaches XML-Format und listet darin alle Dateien auf, die beim Programm zur Laufzeit hinzugefügt werden können. Hier sehen Sie einen Auszug einer solchen Ressourcendatei:

```
<!DOCTYPE RCC><RCC version="1.0">
  <qresource>
    <file>images/cancel.png</file>
    <file>images/folder_page_white.png</file>
    <file>images/page_white.png</file>
    <file>images/text_align_center.png</file>
    <file>images/text_align_justify.png</file>
    <file>images/text_align_left.png</file>
```

```

        <file>images/text_align_right.png</file>
        <file>images/text_underline.png</file>
    </qresource>
</RCC>

```

In diesem Beispiel befinden sich im Verzeichnis *images* mehrere PNG-Bilddateien. Um auf diese Ressourcendateien zugreifen zu können, müssen diese in der Projektdatei (*.pro*) wie folgt eingetragen sein:

```
RESOURCES = RessourcenDateiName.qrc
```

Der mitgelieferte Ressourcen-Compiler »rcc« erstellt aus dieser Datei eine Objektda-
tei und fügt diese am Ende auch endgültig der Anwendung hinzu. Der Vorteil ist ganz
klar, man braucht keine unzähligen Bilddateien mehr, sondern hat alles in einer
Datei, womit auch das versehentliche Löschen oder Vergessen einer Datei entfällt.
Ein weiterer Vorteil, Sie müssen sich nicht mehr viele Gedanken um das Anwen-
dungsverzeichnis des Programms machen und wo darin die Dateien liegen könnten.

Statt eines Dateinamens wird jetzt immer mit dem Pfadpräfix `:/` auf die Datei zuge-
griffen. Zum Beispiel:

```

end = new QAction(
    QIcon(":/images/cancel.png"), tr("B&eenden"), this);

```

Natürlich bezieht sich dieses Ressourcen-System nicht nur auf Grafiken und kann für
Dateien aller Art verwendet werden.

Ebenso besteht die Möglichkeit, eine andere Sprache bei den Ressourcendateien zu
berücksichtigen. Haben Sie bspw. Icons in anderen Sprachen bspw. den Buchstaben
»K« für Kursiv und die englische Alternative »I« für Invers, dann können Sie Folgen-
des in der Ressourcendatei schreiben:

```

<qresource>
    <file>kursiv.jpg</file>
</qresource>
<qresource lang="en">
    <file alias="kursiv.jpg">kursiv_en.jpg</file>
</qresource>

```

Durch die Verwendung von *alias* bei der englischen Version, wird bei der Überset-
zung auf einem System, auf dem die Lokale auf *en_En* gesetzt ist, automatisch das
Icon *kursiv_en.jpg* verwendet, auch wenn im Code *:/kursiv.jpg* gesetzt war.

Das Ressourcen-System in der Praxis verwenden

Natürlich müssen Sie hier jetzt nicht hergehen und eine solche Ressourcendatei von
Hand erstellen. Auch hier hilft uns Qt Creator wieder mit der Erstellung einer solchen

Ressourcendatei. Anhand des Texteditors *JWTexteditor* soll hier demonstriert werden, wie Sie das Ressourcen-System verwenden können.

1. Erstellen Sie hierzu ein neues Projekt aus den Quelldateien im Verzeichnis *kapitel17/JWTexteditor*. Die Icons mit dem Ordner *images* legen Sie jetzt in dasselbe Verzeichnis wie den Quellcode Ihres Projekts. Diesmal legen Sie den Ordner mit den Icons nicht ins Arbeitsverzeichnis des ausführbaren Programms.
2. Klicken Sie jetzt das Projekt in PROJEKTE mit der rechten Maustaste an, und wählen Sie HINZUFÜGEN aus (Abbildung 17.191).

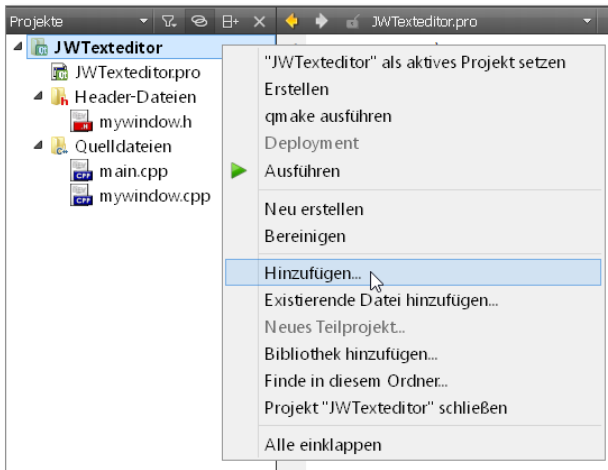


Abbildung 17.191 Dem Projekt eine neue Datei hinzufügen

3. Wählen Sie im sich öffnenden Dialog NEUE DATEI bei den Vorlagen unter DATEIEN UND KLASSEN QT und dann die Vorlage QT-RESSOURCENDATEI aus, und klicken Sie auf die Schaltfläche AUSWÄHLEN (Abbildung 17.192).

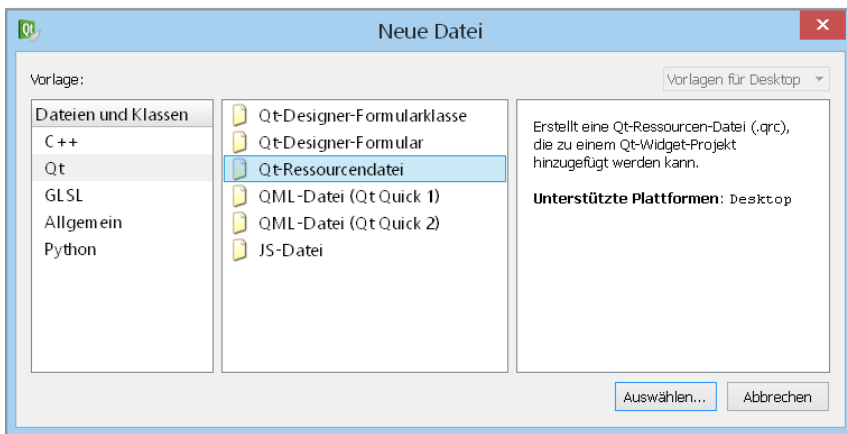


Abbildung 17.192 Diesmal soll eine Qt-Ressourcendatei erstellt werden.

4. Im nächsten Dialog geben Sie NAME und PFAD für die Ressourcendatei ein (Abbildung 17.193). Klicken Sie auf WEITER, und Sie erhalten am Ende eine Zusammenfassung.

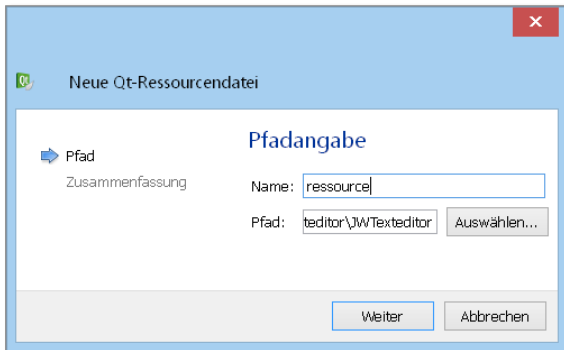


Abbildung 17.193 Name und Pfade der neuen Ressourcendatei

5. Sie finden die Ressourcendatei jetzt in der Übersicht PROJEKTE (Abbildung 17.194). Klicken Sie die Datei an, und es öffnet sich der Ressourcen-Browser in der Mitte des Editors. Wählen Sie hier unten zunächst bei HINZUFÜGEN den Befehl PRÄFIX HINZUFÜGEN aus.

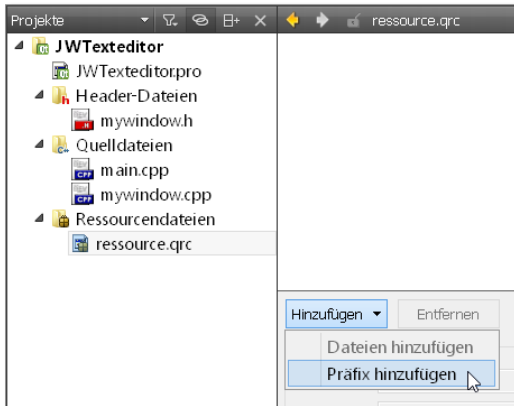


Abbildung 17.194 Die neue Ressourcendatei wird editiert.

6. In der Textzeile PRÄFIX finden Sie nun ein benutzerdefiniertes Präfix, welches Sie verwenden können, um unterschiedliche Kategorien von Ressourcen einzurichten. Im Beispiel wurde der Text aus PRÄFIX entfernt und leer gelassen. Jetzt wollen wir die Dateien über HINZUFÜGEN und dann DATEIEN HINZUFÜGEN der Ressource hinzufügen (Abbildung 17.195). Hierzu habe ich alle Dateien aus dem Ordner *images* ausgewählt und der Ressourcendatei hinzugefügt.

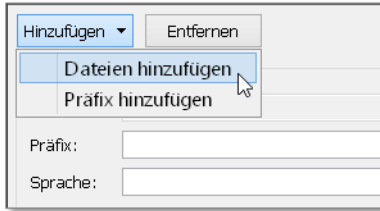


Abbildung 17.195 Dateien der Ressourcendatei hinzufügen

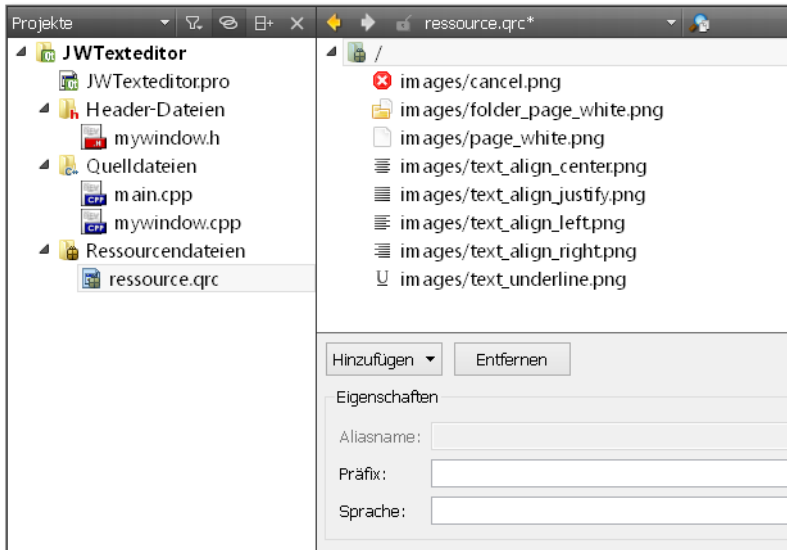


Abbildung 17.196 Übersicht über die hinzugefügten Dateien der Ressource

7. Jetzt können leicht Sie auf die Ressourcen zugreifen (Abbildung 17.196), statt einen umständlichen Dateinamen zu nutzen, den wir bisher wie folgt zusammengebastelt haben:

```
act1 = fileMenu->addAction(
    QIcon(QString("%1%2")
        .arg(QCoreApplication::applicationDirPath())
        .arg("/images/page_white.png")), tr("&Neu"),
    this, SLOT(newFile()),
    QKeySequence(tr("Ctrl+N", "Datei|Neu")) );
```

Jetzt brauchen Sie nur noch das Pfadpräfix `:/` davorzustellen. Ändern Sie daher alle Stellen im Quellcode, wo Sie bislang auf die Icons mit dem zusammengebastelten Pfad zugegriffen haben, auf das Pfadpräfix `:/` um. Bei der eben gezeigten Zeile wäre dies dann:

```
act1 = fileMenu->addAction(
    QIcon(":/images/page_white.png"), tr("&Neu"),
    this, SLOT(newFile()),
    QKeySequence(tr("Ctrl+N", "Datei|Neu")) );
```

Der Vorteil ist jetzt, dass unsere Icons fest an einem Ort in der Anwendung hinzugefügt wurden und Sie sich jetzt keine Gedanken mehr machen müssen, wo die Icons sind.

Icon für die Anwendung setzen

Auch für die Anwendung selbst können Sie ein Icon setzen. Fügen Sie einfach ein Icon (bspw. 32 × 32 oder 64 × 64) der Ressourcendatei hinzu, und verwenden Sie die Methode `QWidget::setWindowIcon()`. Allerdings betrifft dies nur das Icon in der Taskleiste und oben in der Titelleiste. Wenn Sie ein Icon für die ausführbare Anwendung erstellen wollen, dann ist dies systemabhängig. Hierfür empfehle ich Ihnen wieder, den Assistenten von Qt zu verwenden. Suchen Sie hierzu nach: »Setting the Application Icon«.

17.16 Klassen und Typen zum Speichern von Daten

Bisher haben wir immer so getan, als würden wir die Qt-Klassen zum Speichern von Daten kennen. In diesem Abschnitt soll kurz auf diese Klassen und Typen von Qt eingegangen werden, die wir im Buch verwendet haben, um Daten zu speichern. Für mehr Informationen sollten Sie auch hier wieder die Dokumentation von Qt zu Rate ziehen.

Qt-Container und Qt-Algorithmen

Qt bietet für fast jeden Standard-C++-Container eine eigene Version an. So finden Sie in Qt bspw. mit `QVector` eine gleichwertige Qt-Alternative zu `std::vector`. Selbiges gilt auch für die Algorithmen. Auch Namen der Containermethoden und Algorithmen sind mit den Standardversionen identisch. Hier soll allerdings auf die Qt-Varianten nicht mehr weiter eingegangen werden.

17.16.1 Qt-eigene Typdefinitionen

Sobald Sie vorhaben sollten, portable Programme zu schreiben, empfiehlt es sich, die Qt-eigenen Typdefinitionen zu verwenden. Mit diesen wird sichergestellt, dass ein Typ auf allen Plattformen dieselbe Eigenschaft besitzt. Qt verwendet hierfür `typedef` und keine Makros. In Tabelle 17.7 finden Sie einen Überblick zu den eigenen Typen in Qt und deren bekannten Gegenstücken.

Qt-Typ	typedef aus	Üblicher Wertebereich
qint8	signed char	–128 bis +127
qint16	signed short	–32768 bis +32767
qint32	signed int	–2147483648 bis +2147483647
qint64	long long int	Vorzeichenbehaftete 64-Bit-Werte. Unterstützt der Compiler solche Werte nicht, existiert das Makro <code>Q_INT64_C()</code> , bspw.: <code>qint64 val = Q_INT64_C(13412452412341343);</code>
qlonglong	long long int	dito
quint8	unsigned char	0 bis 255
quint16	unsigned short	0 bis 65535
quint32	unsigned int	0 bis 4294967296
quint64	unsigned long long int	Vorzeichenlose 64-Bit-Werte. Unterstützt der Compiler solche Werte nicht, existiert das Makro <code>Q_UINT64_C()</code> , bspw.: <code>quint64 val = Q_UINT64_C(13412452412341343);</code>
qulonglong	unsigned long long int	dito
qreal	double	Fließkommazahl mit doppelter Genauigkeit
uchar	unsigned char	0 bis 255
uint	unsigned int	0 bis 4294967296
ulong	unsigned long	0 bis 4294967296
ushort	unsigned short	0 bis 65535

Tabelle 17.7 Qt-eigene Typdefinitionen

17.16.2 QString

Die Klasse `QString` ist in Qt das Gegenstück zur Standardklasse `std::string` von C++. Auch die Methoden sind sowohl namentlich als auch in der Verwendung recht ähn-

lich bzw. gleichwertig wie im Standard. Die String-Klasse von Qt bringt noch mehr Features und Methoden mit als die Standardversion.

Wer einen Blick auf die Dokumentation bzw. Referenz von `QString` wirft, sollte keine Probleme mehr mit der Verwendung von `QString` haben. Mir geht es in diesem Abschnitt eher ein wenig um die Codierung von `QString`, ein Thema, auf das man schneller stößt, als einem lieb ist, und das sehr frustrierend sein kann, wenn die Zeichen nicht richtig dargestellt werden.

Qt speichert die Strings mit der Klasse `QString` im Unicode-Format. Das bedeutet, dass jedes Zeichen in einem `QString` ein 16-Bit-`QChar` ist und kein 8-Bit-`char`.

Qt und Unicode

Qt verwendet in seiner kompletten API immer Unicode. Somit ist es egal, welche Sprache Sie benutzen, Sie sind da unabhängig.

Einen String der Klasse `QString` zu initialisieren, ist wie bei der Klasse `std::string`. Beispielsweise übergeben Sie folgendermaßen ein `const char*` an den Konstruktor:

```
QString str = "Ein String";
```

Diese `const-char*`-Daten werden von `QString` im Unicode-Format konvertiert. Dies wird und kann auch von Hand mit der Methode `QString::fromAscii()` gemacht werden. Per Standard behandelt die Methode `fromAscii()` die Zeichen über 128 als Latin-1-Zeichen. Dies kann allerdings mit der Methode `QTextCodec::setCodecForCString()` geändert werden. Wollen Sie bspw. für Zeichen über 128 bei Konvertierung von einem `const char*` zu einem `QString` die unter Windows häufig verwendete Codierung Windows-1252 verwenden, können Sie dies wie folgt realisieren:

```
QTextCodec::setCodecForCString(
    QTextCodec::codecForName("Windows-1252") );
```

Durch die Methode `QTextCodec::codecForName()` erhalten Sie einen Zeiger auf die entsprechende Codierung zurück. Welche Codierung unterstützt wird, sollten Sie der Dokumentation von `QTextCodec` entnehmen.

Ebenso lassen sich auch einzelne Zeichen an `QString` übergeben:

```
str[0] = 'C';
str[1] = QChar(0x43); // ebenfalls das Zeichen: C
```

Ist die Quelldatei in Latin-1 codiert, können hierbei auch echte Latin-1-Zeichen verwendet werden:

```
str[2] = 'Æ';
```

Ansonsten können Sie bei einer anderen Codierung der Quelldatei jederzeit auf numerische Werte zurückgreifen, die auf jeden Fall funktionieren:

```
str[2] = QChar(0xC6); // wie Æ
```

17.16.3 QStringList

Die Klasse `QStringList` wurde aus `QList<QString>` abgeleitet. Somit bietet diese Klasse auch alle schnellen Listen-Operationen wie schnelles Einfügen oder Löschen an (mehr zu den entsprechenden Methoden finden Sie in der Dokumentation von `QList`). Natürlich bietet die Klasse `QStringList` auch noch zusätzliche Methoden mit an, um die Arbeit mit String-Listen weiter zu vereinfachen.

Um Strings der Liste hinzuzufügen, können Sie die Methoden `append()`, `operator+=(())` und `operator<<()` verwenden. Mehrere Strings lassen sich somit problemlos wie folgt der Liste hinzufügen:

```
QStringList slist;
slist << "red" << "green" << "blue" << "yellow";
```

Auf für das Durchlaufen der String-Liste stehen Ihnen mehrere Wege zur Verfügung, bspw. über einen Index mit `at()`:

```
for (int i = 0; i < slist.size(); ++i)
    cout << slist.at(i).toLocal8Bit().constData() << endl;
```

Eine weitere Möglichkeit, durch die String-Liste zu laufen, ist – C++-typisch – mit einem Iterator:

```
QStringList::const_iterator constIt;
for ( constIt = slist.constBegin();
      constIt != slist.constEnd(); ++constIt) {
    cout << (*constIt).toLocal8Bit().constData() << endl;
}
```

Auch zur Manipulation der Strings in der Liste bietet `QStringList` einige interessante Methoden. Zu erwähnen ist etwa die Methode `join()`, mit der Sie alle Strings in der Liste in einen einzelnen String schreiben. Zusätzlich kann noch ein Trennzeichen zwischen den einzelnen Strings in der Liste hinzugefügt werden, bspw.:

```
QString str = slist.join("-");
// str == "red-green-blue-yellow"
```

Andersherum geht dasselbe, wenn man einen String anhand eines bestimmten Trennzeichens in einzelnen Strings in die Liste einfügen will. Ein Beispiel:

```
QString str = "red-green-blue-yellow";
QStringList slist;
slist.split("-");
// sList = ("red", "green", "blue", "yellow")
```

17.16.4 QVariant

Die Klasse `QVariant` ist so etwas wie eine Union in C++. Mit dieser Klasse können Sie verschiedene Typen (bspw. `int`, `double` usw.), auch Qt-spezifische Typen (`QString`, `QTime` usw.) ineinander umwandeln.

Was für einen Typ von Daten die Klasse `QVariant` gerade beinhaltet, lässt sich mit der Methode `QVariant::type()` ermitteln. Diese Methode gibt einen Wert aus der enum-Variablen `QVariant::Type` zurück. Mögliche Werte (bzw. genauer Typen) und deren Bedeutung finden Sie in der Dokumentation zu `QVariant`.

Zum Konvertieren in den entsprechenden Typ stehen die `toT()`-Methoden zur Verfügung. Wobei hier `T` für den Typ steht. Ob überhaupt in den Typ konvertiert werden kann, lässt sich mit der Methode `QVariant::canConvert(T)` ermitteln.

17.16.5 Typen für Datum und Uhrzeit

Um mit Datum und Uhrzeit zu arbeiten, bietet Qt die Klassen `QDate`, `QTime` und `QDateTime` an. Alle Klassen bieten allerlei Methoden, um entsprechende Werte zu formatieren, zu vergleichen und zu berechnen. Anhand der Namen der Klassen lässt sich eigentlich schon herauslesen, wozu welche Klasse dient.

Sofern es Ihnen nur um das Kalenderdatum geht (also bspw. das Jahr, der Monat als Nummer und/oder Name, der Tag als Nummer und/oder Name usw.) stellt Ihnen die Klasse `QDate` alles hierfür zur Verfügung. Ebenso gibt es einige statische Funktionen, die bspw. das aktuelle Datum zurückgeben. Auch ist es möglich, Berechnungen mit Tagen, Monaten und Jahren durchzuführen. Und selbstverständlich geht es auch, alle möglichen Datumsarten als formatierten String zurückzugeben.

Gleiches wie für das Kalenderdatum mit `QDate` gilt für die Uhrzeit mit `QTime`. Die Klasse `QTime` bietet Ihnen alles, was mit Zeit zu tun hat, von der aktuellen Uhrzeit bis hin zu den einzelnen Zeitangaben (Stunden, Minuten, Sekunden, Millisekunden usw.). Natürlich sind bei dieser Klasse Methoden zur Manipulation und Berechnung von Zeiten vorhanden. `QTime` verwendet ein 24-Stunden-Zeitformat und hat kein AM-/PM-Konzept. Außerdem weiß `QTime` nichts von Zeitzonen oder DST (*daylight saving time*).

Wer beides, also Datum und Uhrzeit, gleichzeitig benötigt, der kann auf die Klasse `QDateTime` zurückgreifen. Diese Klasse ist eine Kombination aus `QDate` und `QTime` mit allen Features, die auch die einzelnen Klassen besitzen. `QDateTime` kann aber auch

seine Daten als lokale Zeit (*local time*) oder als UTC (Coordinated Universal Time, ersetzt Greenwich Mean Time/GMT) verarbeiten. Standardmäßig wird zwar die lokale Zeit verwendet (bspw. beim Rückgabewert von `QDateTime::currentDateTime()`), aber wenn Sie eine Zeit im UTC-Format benötigen, müssen Sie diese nur mit `QDateTime::toUTC()` konvertieren.

17.17 Fazit

Ich denke, in diesem Kapitel wurden Sie relativ gut mit dem Qt-Framework vertraut gemacht. Trotzdem muss hier immer wieder betont werden, dass Qt weitaus mehr zu bieten hat, als in diesem Kapitel gezeigt werden konnte. Das Kapitel befähigt Sie jetzt zwar, Anwendungen mit einer grafischen Oberfläche zu erstellen, aber für komplexere Oberflächen kommen Sie nicht um ein Studium der mitgelieferten Beispiele von Qt Assistant herum. Zusammen mit der Entwicklungsumgebung, dem Designer und den unzähligen Bibliotheken, die das Qt-Framework mitliefert, können Sie wirklich jede nur erdenkliche Anwendung damit programmieren.

Wenn man allerdings das Prinzip von Qt verstanden hat, fällt es einem nicht schwer, sich damit im Selbststudium vertraut zu machen. Ob der Code nun manuell geschrieben wird oder mit dem Designer erstellt wird, das Prinzip ist fast immer dasselbe: Sie legen fest, ob Sie ein Widget, einen Dialog oder ein Hauptanwendungsfenster erzeugen. Dann erstellen Sie die einzelnen Bedienelemente dafür. Mithilfe von Containern ordnen Sie diese Bedienelemente und layouten sie. Dann müssen Sie für die Bedienelemente noch Signal-Slot-Verbindungen einrichten und gegebenenfalls benutzerdefinierte Signale bzw. Slots erstellen.

Das Qt-Framework bietet auch unglaublich viele Funktionalitäten zur Ein- und Ausgabe von Daten, was ich Ihnen hier komplett unterschlagen habe. Die Verwendung von Ein- und Ausgabefunktionen für Dateien, Verzeichnisse, Puffer, Prozesse (Interprozesskommunikation), Threads, Datenbanken oder die Netzwirkommunikation mit Qt hat den Vorteil, dass Sie alle diese Techniken ohne Probleme auf andere Systeme portieren können. Gerade bei Dingen jenseits des C++-Standards (wie bspw. der Netzwirkommunikation) wird einem das Programmiererleben hiermit enorm vereinfacht, wenn man ein auf dem Windows-Rechner erstelltes Programm auf Mac OS X oder Linux/Unix portieren will.

Anhang

Kein Buch ohne einen Anhang. Neben den üblichen Tabellen zu den Operatoren, der Vorrangtabelle und den Schlüsselwörtern in C++ finden Sie hier auch noch zusätzliche Informationen zur Informationsspeicherung (Zahlensystem, Bits und Bytes) und den Zeichensätzen.

A.1 Operatoren in C++ und deren Bedeutung (Übersicht)

In Tabelle A.1 finden Sie eine Übersicht der Operatoren von C++ und deren Bezeichnung bzw. Verwendung.

Operator	Bezeichnung
Arithmetische Operatoren	
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo-Division
+	Vorzeichenoperator (unär)
-	Vorzeichenoperator (unär)
++	Inkrementoperator
--	Dekrementoperator
Vergleichsoperatoren	
==	gleich
!=	ungleich
<	kleiner
<=	kleiner gleich

Tabelle A.1 Übersicht der Operatoren in C++

Operator	Bezeichnung
Vergleichsoperatoren	
>	größer
>=	größer gleich
Logische Operatoren	
&&	UND
	ODER
!	NICHT
Zuweisungsoperatoren	
=	einfache Zuweisung
op=	zusammengesetzte Zuweisung
Bitoperatoren	
&	UND
~	NICHT
	ODER
^	exklusives ODER
>> bzw. <<	Shift-Operator
Cast-Operatoren	
(typ)	C-Cast
dynamic_cast<>	dynamischer Cast
static_cast<>	statischer Cast
const_cast<>	const-Cast
reinterpret_cast<>	reinterpret-Cast
Zugriffsoperatoren	
::	Bereichsoperator
[]	Indexoperator

Tabelle A.1 Übersicht der Operatoren in C++ (Forts.)

Operator	Bezeichnung
Zugriffsoperatoren	
*	Verweisoperator
.	Punktoperator
->	Pfeiloperator
.*	Dereferenzierung von Elementzeigern
->*	Dereferenzierung von Elementzeigern
Operatoren für die Speicherverwaltung	
new	ein Objekt dynamisch anlegen
delete	ein dynamisch angelegtes Objekt zerstören
new []	Klassen-Array dynamisch anlegen
delete []	ein dynamisch angelegtes Klassen-Array zerstören
Restliche Operatoren	
?:	Auswahloperator (ternär)
,	Kommaoperator
name()	Aufruf der Funktion name
typ()	ein temporäres Objekt vom Typ typ erzeugen
sizeof()	Größe eines Datentyps ermitteln
typeid()	Typinformationen

Tabelle A.1 Übersicht der Operatoren in C++ (Forts.)

A.2 Vorrangtabelle der Operatoren

In Tabelle A.2 werden die Operatoren von C++ und ihre Assoziativität (die Bindung der Operanden) in absteigender Reihenfolge aufgelistet. Operatoren derselben Prioritätsklasse haben dieselbe Rangstufe.

Priorität	Operator	Assoziativität
1.	::	von links nach rechts
2.	. -> [] ++ (Postfix) -- (Postfix) name(), typ(), typeid() dynamic_cast<>, static_cast<> const_cast<>, reinterpret_cast<>	von links nach rechts
3.	! ~ + (unär) - (unär) ++ (Präfix) -- (Präfix) & * (Verweis) new, new[] delete, delete[] (typ), sizeof()	von rechts nach links
4.	.* ->*	von links nach rechts
5.	* / %	von links nach rechts
6.	+ (binär) - (binär)	von links nach rechts
7.	<< >>	von links nach rechts
8.	< <= >= >	von links nach rechts
9.	== !=	von links nach rechts
10.	&	von links nach rechts
11.	^	von links nach rechts
12.		von links nach rechts
13.	&&	von links nach rechts
14.		von links nach rechts
15.	?:	von rechts nach links
16.	= += -= *= /= %= &= ^= = <<= >>=	von rechts nach links
17.	,	von links nach rechts

Tabelle A.2 Rangfolge und Assoziativität der Operatoren in C++

A.3 Schlüsselwörter von C++

Die Schlüsselwörter sind reservierte Bezeichner in C++ mit einer festen Bedeutung und daher sollten diese nicht für Variablen, Funktionen, Klassen usw. verwendet werden. Tabelle A.3 führt die Schlüsselwörter von C++ im Einzelnen auf (**fett** formatierte Schlüsselwörter wurden mit dem C++11-Standard eingeführt).

alignas	alignof	asm	auto	bool
break	case	catch	char	char16_t
char32_t	class	const	constexpr	const_cast
continue	decltype	default	delete	do
double	dynamic_cast	else	enum	explicit
export	extern	false	float	for
friend	goto	if	inline	int
long	mutable	namespace	new	noexcept
nullptr	operator	private	protected	public
register	reinterpreter_cast	return	short	signed
sizeof	static	static_assert	static_cast	struct
switch	template	this	thread_local	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

Tabelle A.3 Schlüsselwörter von C++

A.4 Informationsspeicherung

Ich denke mal, ich muss niemanden mehr erzählen, dass Computer ihre Informationen als Folge von Einsen (1) und Nullen (0) abspeichern. Man bezeichnet diese Form der Darstellung als binäre Codierung. Ein Beispiel dieser binären Darstellung ist das Dualsystem (oder auch Zweiersystem genannt).

Die einzelnen Binärstellen, die der Computer speichert, werden als Bit (Abkürzung für »BInary digiT«, dt. Binärziffer) bezeichnet. Ein Bit ist somit die kleinste Informationseinheit, mit der der Computer arbeiten kann. Im Grunde ist ein Bit die einzige vorstellbare Möglichkeit, um überhaupt in einem Computer Zeichen darstellen zu können.

Dezimalsystem

Das Dezimalsystem oder Zehnersystem (lat. *decimus* = der Zehnte) ist ein Stellenwertsystem zur Darstellung von Zahlen. Das Dezimalsystem ist heute das weltweit verbreitetste Zahlensystem und stammt ursprünglich aus Indien. Im Grunde hat das Dezimalsystem von allen Zahlensystemen mit dem Computer am wenigsten zu tun. Allerdings wird dieses Zahlensystem gewöhnlich im Alltag verwendet und ist auch das erste Zahlensystem, das einem beigebracht wird. Zur Ein- bzw. Ausgabe von Zahlen am Computer wird diese Form vorwiegend verwendet.

Das Dezimalsystem verwendet als Basis $B = 10$, so dass der Wert der ersten Stelle (rechts) 1 ist und sich mit jeder weiteren Stelle verzehnfacht. Die Dezimalzahl 2345 wird bspw. formal wie in Tabelle A.4 analysiert.

Ziffern	2	3	4	5
Stellenwert	1000	100	10	1
Schema	10^3	10^2	10^1	10^0
Gesamtwert	2000	300	40	5

Tabelle A.4 Darstellung der Dezimalzahl 2345

Dualsystem (Binärsystem)

Das Dualsystem (oder auch Binärsystem oder Zweiersystem) ist für den Computer selbst das Wichtigste, weil dieser intern damit arbeitet. Das Dualsystem (lat. *dualis* = zwei enthaltend) ist ein Zahlensystem, welches nur zwei Ziffern zur Darstellung von Zahlen verwendet. Das Dualsystem wurde nicht etwa im Computerzeitalter erfunden, sondern schon weiter vorher. Allerdings erlebte es erst mit der Entwicklung der Digitaltechnik eine starke Verbreitung.

Das Dualsystem ist ein Stellenwertsystem mit der Basis $B = 2$. Die beiden Ziffern mit den Werten Null und Eins werden gewöhnlich mit den Symbolen 0 und 1 dargestellt. In älterer Literatur findet man im Bezug auf die elektronische Datenverarbeitung auch manchmal die Symbole Low (L) und High (H) anstelle von 0 und 1. Somit ist das Dualsystem ein vollständiges Zahlensystem mit der geringstmöglichen Anzahl an verschiedenen Ziffern. Allerdings ist der Nachteil daran, dass die Zahlen beim Dualsystem sehr lang sind.

Dadurch, dass das Dualsystem die Basis $B = 2$ verwendet, hat eine Stelle immer den doppelten Wert der weiter rechts gelegenen Stelle. Dass hierbei die Ziffern 0 und 1 verwendet werden, ist einer der Hauptgründe, warum das Dualsystem für digitale Rechner geeignet ist, deren elektronische Bauteile ebenfalls binär arbeiten (fließt Strom oder nicht?). So ist bspw. der dezimale Wert von 1011 nicht etwa »eintausend-

undelf«, sondern 11. Diesen Wert erhalten Sie, wenn Sie die Werte in der Zeile »Gesamtwert« addieren (siehe Tabelle A.5).

Ziffern	1	0	1	1
Stellenwert	8	4	2	1
Schema	2^3	2^2	2^1	2^0
Gesamtwert	8	0	2	1

Tabelle A.5 Darstellung der Dualzahl 1011

Somit ist das Dualsystem im Grunde das einfachste aller Zahlensysteme, da der Stellenwert nie mit einem Ziffernwert multipliziert werden muss. Denn es gilt, die Stelle ist entweder gesetzt (1), so dass der Stellenwert selbst gilt, oder eben nicht gesetzt (0), dann ist eben auch der Wert 0.

Dualzahlen finden in der elektronischen Datenverarbeitung bei der Darstellung von Festkommazahlen oder ganzen Zahlen Verwendung. Negative Zahlen werden vor allem als 2-Komplement dargestellt, welches nur im positiven Bereich der Dualzahlendarstellung entspricht. Um näherungsweise rationale oder gar reelle Zahlen darzustellen, werden vorzugsweise Fließkommadarstellungen verwendet, bei denen die Zahl normalisiert und in Mantisse und Exponent aufgeteilt wird. Diese beiden Werte werden dann in Form von Dualzahlen gespeichert.

Oktalsystem

Da es recht unbequem ist, Dualzahlen ins Dezimalsystem umzurechnen und umgekehrt, wird gerne das Oktalsystem verwendet, womit sich Zahlen relativ einfach in Dualzahlen umwandeln lassen und umgekehrt. Das Oktalsystem hat die Basis $B = 8$ – also 8 verschiedene Ziffern (0 bis 7). Beim Zählen muss somit beachtet werden, dass nach der 7 nicht die 8, sondern eine Stelle weiter links erhöht werden muss. Somit folgt nach der oktalen 7 die 10. Jede dieser Ziffern einer Oktalzahl kann durch 3 Bit dargestellt werden und umgekehrt (siehe Tabelle A.6).

Oktal	0	1	2	3	4	5	6	7
Dual	000	001	010	011	100	101	110	111

Tabelle A.6 Darstellung einer Oktalzahl durch 3 Bit

In der Praxis finden Oktalzahlen in der Informatik eigentlich nur noch bei der Darstellung von Dateizugriffsrechten unter Linux/Unix Verwendung, wo je 3 Bit die Rechte einer Benutzerklasse darstellen (siehe Manualpage von `chmod`). Zur früheren

Zeit waren hierbei auch noch Datenworte von 24 Bit Länge gebräuchlich, und daher wurden Oktalzahlen zur Eingabe von Bitmustern verwendet, weil diese für den Menschen übersichtlicher als Dualzahlen sind und weil die Umwandlung ins Binärsystem relativ einfach ist (war). Allerdings beträgt heutzutage die übliche Datenwortlänge 16, 32 und 64 Bit, und dafür ist das Hexadezimalsystem für die Ein- und Ausgabe geeigneter. Hierzu eine systematische Analyse einer Oktalzahl: Der dezimale Wert für die Oktalzahl 1234 beträgt 668. Auch diesen Wert erhalten Sie, wenn Sie die Werte in der Zeile »Gesamtwert« addieren (siehe Tabelle A.7).

Ziffern	1	2	3	4
Stellenwert	512	64	8	1
Schema	8 ³	8 ²	8 ¹	8 ⁰
Gesamtwert	512	128	24	4

Tabelle A.7 Darstellung der Oktalzahl 1234

Hexadezimalsystem

Mit dem Hexadezimalsystem (griech. *hexa* = sechs, lat. *decem* = 10 oder auch seltener Sezimalsystem vom lat. *sedecim* = sechzehn) werden Zahlen mit einem Stellenwertsystem der Basis B = 16 dargestellt. In der Informatik ist dieses Zahlensystem das beliebteste zur Darstellung von Speicheradressen, Zeichencodes und sonstigen Byte-Inhalten.

Wie beim Oktalsystem lassen sich mit dem Hexadezimalsystem die von der Maschine verbreiteten Zahlen im Dualsystem in einer für den Menschen einfacher lesbaren Weise notieren und verwenden. Neben den gewohnten Symbolen des Dezimalsystems (0 bis 9) werden beim Hexadezimalsystem für die letzten sechs Ziffern die Buchstaben A bis F (oder auch kleingeschrieben a bis f) verwendet. In Tabelle A.8 finden Sie in der ersten Zeile die Hexadezimalen, gefolgt von den dualen, den dezimalen und den oktalen Zahlen.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17

Tabelle A.8 Alle wichtigen Zahlensysteme der Informatik

Auch hierzu ein Beispiel: Der hexadezimale Wert 2ABC hat im Dezimalsystem den Wert 10940. Diesen Wert erhalten Sie, wenn Sie die Werte in der Zeile »Gesamtwert« in Tabelle A.9 addieren.

Ziffern	2	A	B	C
Einzelwert	1	10	11	12
Stellenwert	4096	256	16	1
Schema	16^3	16^2	16^1	16^0
Gesamtwert	8192	2560	176	12

Tabelle A.9 Darstellung der Hexadezimalzahl 2ABC

A.4.1 Zahlensysteme umrechnen

Zwar gibt es zahlreiche Programme, die einem das Umrechnen der Zahlensysteme abnehmen, aber wenn man selbst solche Programme schreiben will, sind solche Kenntnisse nötig.

Dezimal nach Dual

Dezimalzahlen werden folgendermaßen in Dualzahlen umgerechnet: Zunächst sucht man die größte in der Zahl vorkommende Zweierpotenz (2^n). Dies kann man entweder durch das fortgesetzte Verdoppeln von 2 ausprobieren, bis die Zahl »passt«, oder aber man hat diese Zweierpotenzen im Kopf, bis zu einer gewissen Zahl funktioniert das schon (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 usw.). Bei der Zahl 1200 bspw. wäre dies 1024 (2^{10}).

Im nächsten Schritt müssen Sie die entsprechende Zweierpotenz von der Gesamtzahl abziehen und im Ergebnis an vorderster Stelle eine 1 notieren. Von 1200 blieben hierbei noch 176 übrig.

Jetzt überprüfen Sie, ob die nächstniedrigere Zweierpotenz in der Zahl vorkommt. Wenn dies der Fall sein sollte, notieren Sie wieder eine 1 und ziehen diese Zweierpotenz von der Zahl ab. Kommt sie nicht vor, notieren Sie im Ergebnis an vorderster Stelle eine 0. Von der Zahl 176, die zuvor vom Rest übrig geblieben ist, kommt 512 (2^9) nicht vor, ebenso wie 256 (2^8) nicht darin vorkommt. Also lautet das bisherige binäre Ergebnis 100.

Gehen Sie den vorherigen Schritt der Reihe der Zweierpotenzen nach bis zur 1 durch. Im Fall von 176 sieht dies folgendermaßen aus: 128 kommt in 176 vor (bisheriges binäres Ergebnis: 1001), der Rest: 48. 64 kommt in 48 nicht vor (binäres Ergebnis somit: 10010). 32 kommt allerdings in 48 vor (binäres Ergebnis: 100101), der Rest: 16.

16 kommt exakt in 16 vor (binäres Ergebnis: 1001011), Rest: 0. Zum Schluss müssen Sie nur noch für die nicht besetzten Stellen 2^3 , 2^2 , 2^1 und 2^0 eine 0 anhängen (Endergebnis: 10010110000).

Dual nach Dezimal

Beim Umrechnen von Dualzahlen in Dezimalzahlen müssen Sie nur die Stellenwerte derjenigen Stellen aufaddieren, die den Wert 1 haben. Am einfachsten fängt man hierbei von rechts bei 2^0 an. Beispiel: 10010110000 soll wieder ins Dezimalsystem umgerechnet werden. Die mit 1 besetzten Stellen sind hierbei 2^4 , 2^5 , 2^7 und 2^{10} . Daraus ergibt sich folgende Addition:

$$16 + 32 + 128 + 1024 = 1200$$

Dezimal nach Hexadezimal

Das Umrechnen einer Hexadezimalzahl in eine Dezimalzahl wird folgendermaßen realisiert: Zunächst benötigen Sie die kleinste Sechzehnerpotenz, die größer ist als die umzurechnende Zahl. Damit wissen Sie, dass die höchste besetzte Hexadezimalstelle Ihrer Zahl um einen Wert darunterliegt. Bei der Zahl 2345 bspw. wäre die größere Hexadezimalstelle 4096 (16^3). Angefangen wird somit beim Stellenwert 256 (16^2).

Jetzt können Sie die Dezimalzahl durch den soeben ermittelten Stellenwert dividieren. Das ganzzahlige Ergebnis dieser Division ist der gesuchte Ziffernwert, den Sie an der vordersten Stelle notieren können. Den Rest der Division benötigen wir für den nächsten Schritt. Somit wird aus $2345/256 = 9$, Rest 41. Der erste Ziffernwert der Stelle ist somit 9. Beachten Sie außerdem, dass bei einem Ergebnis ab 10 mit A bis F geschrieben wird.

Jetzt können Sie den vorherigen Schritt für die restlichen Stellen bis zu 1 (16^0) durchführen. In unserem Fall wäre dies: $41/16 = 2$, Rest 9 (bisherige Hexadezimalzahl: 92). Da es sich beim Rest dieser Berechnung um die Einerstelle handelt, können Sie die 9 als einfachen Ziffernwert hinschreiben. Somit lautet das Endergebnis der Dezimalzahl 2345 umgerechnet als Hexadezimalzahl 929.

Hexadezimal nach Dezimal

Die Umrechnung von Hexadezimalzahlen in Dezimalzahlen ist noch einfacher. Sie müssen hierbei nur den jeweiligen Ziffernwert mit dem Wert seiner Stelle multiplizieren und addieren die Ergebnisse zusammen, bspw. 12AB:

$$12AB = 1 \times 163 + 2 \times 162 + 10 \times 161 + 11 \times 160 =$$

$$1 \times 4096 + 2 \times 256 + 10 \times 16 + 11 \times 1 =$$

$$4096 + 512 + 160 + 11 = 4779$$

Oktalzahl nach Dezimal

Auf die Umrechnung von Oktalzahlen in Dezimalzahlen und umgekehrt wurde nicht eingegangen, weil die Berechnung ähnlich funktioniert wie bei der Umrechnung von Hexadezimalzahlen in Dezimalzahlen und umgekehrt.

Oktal nach Dual

Das Umrechnen von Oktalzahlen in Dualzahlen ist ebenfalls relativ einfach – was ja auch der Hauptgrund für die Verwendung von Oktalzahlen ist. Von rechts an gesehen, entsprechen je drei Dualstellen einer Oktalstelle. Die Umrechnung dieser Dreiergruppen erfolgt daher nach dem in Tabelle A.10 dargestellten Schema.

Dual	Oktal	Dual	Oktal
000	0	100	4
001	1	101	5
010	2	110	6
011	3	111	7

Tabelle A.10 Umrechnung von Dualzahlen in Oktalzahlen und umgekehrt

Hexadezimal nach Dual

Das Gleiche wie eben mit der Umrechnung von Oktalzahlen in Dualzahlen gilt für Hexadezimalzahlen und Dualzahlen. Hierbei entsprechen je vier Dualstellen von rechts einer Hexadezimalstelle und umgekehrt. Die Umrechnung erfolgt hierbei nach dem in Tabelle A.11 dargestellten Schema.

Dual	Hexadezimal	Dual	Hexadezimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Tabelle A.11 Umrechnung von Dualzahlen in Hexadezimalzahlen und umgekehrt

A.4.2 Bytes und Maschinenwörter

Da es relativ ineffizient und zudem schwer realisierbar wäre, jedem einzelnen Bit eine eigene Speicheradresse zuzuweisen, werden mehrere von ihnen zu einer Einheit zusammengefasst, welche eine gemeinsame Adresse erhält. Diese Speicherstellen, an denen die Daten im Computer gespeichert werden, können durch Nummern angesprochen werden. Will der Prozessor den Inhalt einer Speicheradresse lesen, erhält er jeweils den Wert aller dieser Bits; genauso muss er beim Schreiben Werte für alle Bits eines solchen Speicherbereichs liefern.

Die Größe der adressierbaren Speicherstelle war im Laufe der Computergeschichte nicht ganz klar. Es gab Varianten von 4 bis 36 Bit – auch solche, die man als »schräg« bezeichnet, worunter bspw. der berühmte PDP-7-Rechner von DEC mit seinen 18 Bit fällt. Auf dem PDP-7 wurde die ursprüngliche Version von Unix entwickelt.

Wie dem auch sei, die Größe der Speicherblöcke, mit denen ein Computer arbeitet, werden als Maschinenwörter bezeichnet. Die Anzahl der Bits eines solchen Maschinenwortes wird als Wortbreite des jeweiligen Prozessors genannt. In den 1970er Jahren hat man sich dann darauf geeinigt, dass bei jedem Computer die Adressierung in 8 Bit großen Blöcken erfolgt. Diese Blöcke wiederum bezeichnet man als Byte.

A.5 Zeichensätze

Irrtum

Um einen Irrtum gleich zu Beginn aufzuklären, der Standard von C bzw. C++ legt sich nicht auf einen bestimmten Zeichensatz fest.

Was ein Zeichensatz ist, lässt sich relativ einfach erklären. Betrachtet man bspw. eine Textdatei, handelt es sich schließlich um nichts anderes als um eine Abfolge von gleich großen Speicherblöcken, in denen verschiedene Bitmuster für die unterschiedlichen Zeichen stehen. Wie breit dieser Speicherblock ist, hängt von dem verwendeten Zeichensatz ab. So wird bspw. der weit verbreitete ASCII-Zeichensatz mit seinen Erweiterungen mit 8 Bit pro Zeichen gespeichert. Unicode hingegen stellt Varianten mit 16 bzw. 32 Bit oder gar mit variabler Bit-Breite zur Verfügung.

ASCII-Zeichensatz

Der in den 1960er Jahren in den USA entwickelte ASCII-Zeichensatz (American Standard Code for Information Interchange) war im Grunde nur eine Adaption der amerikanischen Schreibmaschine. Hierbei wurden sämtliche Zeichen und Funktionen

übernommen, die eine Schreibmaschine erzeugen konnte. Hierbei findet man auch die typischen Schreibmaschinen-Funktionen wie Zeilenvorschub, Wagenrücklauf oder die Glocke, welche das baldige Zeilenende andeutet. Diese sogenannten Steuerzeichen werden von den verschiedenen Terminals, Shells oder Texteditoren unterschiedlich genutzt.

Der ASCII-Code alleine ist nur 7 Bit breit, worin sich also 128 Zeichen (2^7) darstellen lassen. Der verfügbare Platz des achten Bits wurde als erweiterter Zeichensatz genutzt. Daraus (aus dem achten Bit) wurden verschiedene Zeichensätze entwickelt, welche alle die ersten 128 Zeichen gemeinsam haben und die restlichen 128 Zeichen für eigene Erweiterungen nutzen.

Dez		0	16	32	48	64	80	96	112
	Hex	0	10	20	30	40	50	60	70
0	0	NUL	DLE	(Blank)	0	@	P	'	p
1	1	SOH	DC1	!	1	A	Q	a	q
2	2	STX	DC2	"	2	B	R	b	r
3	3	ETX	DC3	#	3	C	S	c	s
4	4	EOT	DC4	\$	4	D	T	d	t
5	5	ENQ	NAK	%	5	E	U	e	u
6	6	ACK	SYN	&	6	F	V	f	v
7	7	BEL	ETB	'	7	G	W	g	w
8	8	BS	CAN	(	8	H	X	h	x
9	9	HT	EM	)	9	I	Y	i	y
10	A	NL	SUB	*	:	J	Z	j	z
11	B	VT	ESC	+	;	K	[	k	{
12	C	NP	FS	,	<	L	\	l	
13	D	CR	GS	-	=	M	]	m	}
14	E	SO	RS	.	>	N	^	n	~
15	F	SI	US	/	?	O	_	o	(DEL)

Tabelle A.12 ASCII-Code-Tabelle

In Tabelle A.12 sehen Sie den gesamten ASCII-Zeichensatz. Die Nummerierung ist hier dezimal und hexadezimal zu sehen. Das Zeichen »A« bspw. besitzt den hexadezimalen Code 41 oder dezimal 65, einfach den Wert der Zeile mit dem Wert der Spalte addieren.

Die Zeichen 0 bis 31 (hexadezimal 00 bis 1F) sind Steuerzeichen, die mit ihren traditionellen Kurznamen verzeichnet wurden. Beispielsweise ist OA (dezimal 10) das Line-Feed, OD (13) ist das Carriage Return und OO (das Zeichen '\O') wird vor allem als Abschlussmarkierung für C-Strings verwendet.

ASCII vs. Unicode

Auch wenn Unicode stark auf dem Vormarsch ist, bleibt der ASCII-Zeichensatz (ohne Erweiterung) der wichtigste und grundlegendste aller Computerzeichensätze. Viele (zum Teil ältere) Konsolen- oder Netzwerkanwendungen verarbeiten keine Zeichen über reines ASCII hinaus. Auch bei Bezeichnern der traditionellen Programmiersprachen wie eben C++ wird empfohlen, nur ASCII-Zeichen zu verwenden.

ASCII-Erweiterungen

Es wurde bereits erwähnt, dass der ASCII-Zeichensatz aus USA kommt und somit auch nur für englische Texte geeignet ist. Der ASCII-Zeichensatz unterstützt keine Umlaute und sonstige Sonderzeichen. Da man aber sowieso die einzelnen 7 Bit breiten ASCII-Zeichen in einem 8 Bit großen Block speichert, sind in diesem Byte also noch Platz für weitere 128 Zeichen. IBM führte dann einen solchen erweiterten ASCII-Zeichensatz ein, der die wichtigsten diakritischen Zeichen der west- und mitteleuropäischen Sprachen und eine Reihe von anderen Grafikzeichen (bspw. gerade und gewinkelte Linien) beinhaltet.

Andere Hersteller wiederum erfanden eigene Zeichensätze. Bspw. belegte Apple die 128 Zeichen über ASCII anders. Daher können Sie auch bis heute noch deutsche Textdateien von einem Mac nur mit Fehlern unter MS-Windows und umgekehrt lesen.

Windows-Eingabeaufforderung

MS-Windows verwendet übrigens nicht den von IBM erweiterten ASCII-Zeichensatz oder die MS-DOS-Variante davon, sondern den inzwischen standardisierten ANSI-Zeichensatz. Da MS-Windows eine grafische Oberfläche hatte, wurden einige Grafikzeichen nicht mehr benötigt. Dieser freigewordene Platz wurde für internationale diakritische Zeichen verwendet. Dies ist übrigens auch der Grund dafür, dass statt deutscher Umlaute irgendwelche seltsamen Sonderzeichen angezeigt werden, wenn Sie ein Konsolenprogramm in einem Windows-Editor schreiben.

Leider hatte man nun das Problem, dass die erweiterten Zeichensätze betriebssystemspezifisch waren, da für jede andere Sprache ein anderer Zeichensatz nötig war. Zeitweise störte dies niemanden. Die Betriebssystemhersteller lösten das Problem, indem sie für jedes Land, in dem das entsprechende System verkauft wurde, einen angepassten Zeichensatz oder eine sogenannte Codepage herstellten, um die Besonderheiten der jeweiligen Sprache darstellen zu können.

Zu einem echten Problem wurde dies allerdings erst, als sich das Internet immer schneller ausgebreitet hat. Daher wurde ein plattformübergreifender und international gültiger Standard für Zeichensätze nötig. Dieser wurde mit der ISO-8859 geschaffen. Diese Zeichensätze werden durchnummeriert von ISO-8859-1 bis mittlerweile ISO-8859-16. In Tabelle A.13 finden Sie die wichtigsten ISO-8859-Zeichensätze.

ISO-Zeichensatz	Alternativer Name	Beschreibung
ISO-8859-1	ISO-Latin-1	Entspricht weitgehend dem ANSI-Zeichensatz. Unterstützt die Sprachen von USA und Westeuropa.
ISO-8859-2	ISO-Latin-2	Unterstützt zusätzlich einige slawische Sprachen.
ISO-8859-3	ISO-Latin-3	Unterstützt Esperanto, Maltesisch und Türkisch.
ISO-8859-4	ISO-Latin-4	Dient der Unterstützung der baltischen Sprachen (Estnisch, Lettisch und Litauisch).
ISO-8859-5		Kyrillisch
ISO-8859-6		Arabisch
ISO-8859-7		Griechisch
ISO-8859-8		Hebräisch
ISO-8859-9	ISO-Latin-5	Wie Latin-1; ersetzt aber isländische durch türkische Zeichen. Wird für Türkisch häufiger verwendet als Latin-3.
ISO-8859-10	ISO-Latin-6	Unterstützt Grönländisch und Lappisch.

Tabelle A.13 Standardisierte ASCII-Erweiterungs-Zeichensätze

Erwähnt werden sollte auch noch ISO-8859-15 (eine Revision von ISO-8859-1) mit €-Zeichen. Einen besseren Überblick über die verschiedenen ISO-8859 finden Sie auf Wikipedia unter http://en.wikipedia.org/wiki/ISO/IEC_8859.

Unicode

Das Unicode-System wurde eingeführt, um das Problem mit den Schriften in den Griff zu bekommen. Dazu wurde von vornherein mehr Speicherplatz für die einzelnen Zeichen eingeplant, um mehrere unterschiedliche Zeichen darstellen zu können. Die erste Version von Unicode verwendete noch 16 Bit pro Zeichen, um bis zu 65.536 Zeichen darstellen zu können. Seit Unicode 3.1 gibt es auch andere Varianten mit 3 bzw. 4 Byte, um noch mehr Zeichen in Unicode unterzubringen.

Leider wird Unicode noch nicht von allen Betriebssystemen, Programmiersprachen, APIs und Anwendungen verwendet. Aus diesem Grund, um Probleme zu vermeiden, entsprechen die untersten 256 Zeichen exakt dem Zeichensatz von ISO-Latin-1. Mit der Unicode-Codierung UTF-8 können einzelne ASCII-Zeichen sogar in 1 Byte dargestellt werden, und für die Darstellung von höheren Zeichen werden dann spezielle Codes verwendet. Damit ist UTF-8 zu ASCII abwärts kompatibel.

Die grundlegende Teilmenge von Unicode (mit den 16 Bit breiten Zeichen), wird als Basic Multilingual Plane (kurz BMP) bezeichnet. Dabei handelt es sich um die wichtigsten Zeichen von Sprachen, mathematischen Sonderzeichen, Piktogrammen, Schmuckelementen und anderen Symbolen. In Tabelle A.14 finden Sie eine Übersicht über einige wichtige Codebereiche des BMP-Bereichs von Unicode. Die Codes von Unicode-Zeichen werden üblicherweise im Hexadezimalcode mit vorangestelltem U+ geschrieben; beim BMP-Bereich werden vier Hexadezimalstellen verwendet, ansonsten je nach Bedarf mehr.

Hexadezimal	Dezimal	Codebereich
\u0000 – \u007F	0 – 127	ASCII
\u0080 – \u00FF	128 – 255	ISO-Latin-1
\u0100 – \u017F	256 – 383	Latin Extended A (speziellere diakritische Zeichen)
\u0180 – \u024F	384 – 591	Latin Extended B (spezielle, eigentlich nichtlateinische Zeichen, die in einigen Sprachen verwendet werden)
\u0370 – \u03FF	880 – 1023	Griechisch
\u0400 – \u04FF	1024 – 1279	Kyrillisch
\u0590 – \u05FF	1424 – 1535	Hebräisch
\u0600 – \u06FF	1536 – 1791	Arabisch

Tabelle A.14 Codebereiche im Unicode-Block BMP

Hexadezimal	Dezimal	Codebereich
\u0900 – \u097F	2304 – 2431	Devanagari (Schrift der meisten indischen Sprachen)
\u2200 – \u22FF	8704 – 8959	mathematische Symbole
\u3040 – \u309F	12352 – 12447	Hiragana (japanische Lautschrift)
\u30A0 – \u30FF	12448 – 12543	Katakana (japanische Lautschrift)
\u4E00 – \u9FA5	19968 – 40869	CJK Unified Ideographs (chinesische Zeichen, die zum Teil auch für Japanisch und Koreanisch verwendet werden)

Tabelle A.14 Codebereiche im Unicode-Block BMP (Forts.)

Außerhalb des Basic Multilingual Plane (BMP)

Außerhalb der BMP und damit nur mit mehr als 2 Byte oder 16 Bit notierbar sind Dinge wie ägyptische Hieroglyphen, Spielkarten, Mahjongsymbole und Emoji (Smileys).

Index

[illegible]

A

Abgeleitete Klasse	408
<i>erweitern</i>	423
Ableitung, mehrfache	409
abort	221, 497
Abstrakte Klassen	446
Adapterklassen	553
Adressen ermitteln	156
advance	597

Algorithmen	546, 603	bad_array_new_length	493
<i>Bereich</i>	604	bad_cast	494
<i>für reguläre Ausdrücke</i>	705	bad_exception	495
<i>Prädikat</i>	606	bad_function_call	495
<i>Übersicht</i>	619	bad_typeid	451, 494
Alias-Template	185, 534	bad_weak_ptr	496
Anjuta	30	badbit	655
Anweisungen	39	basic_regex	699
Anweisungsblock	38	Basisdatentypen	51
Arithmetische Operatoren	78	Basis-Initialisierer	430
array	552	Basisklasse	407
array (C++11)	139	<i>Basis-Initialisierer</i>	430
Arrays	129	<i>direkte</i>	409, 415
<i>assoziative</i>	130, 142	<i>indirekte</i>	409, 415
<i>C-Style</i>	135	<i>initialisieren</i>	429
<i>rohe</i>	135	<i>virtuelle</i>	463
Assoziative Container	560, 627	Basisklassenzeiger	438, 448
Assoziatives Array	142	Bedingte Anweisung	99
async	755	Bedingte Kompilierung	256
Asynchrones Arbeiten	755	Bedingter Ausdruck	110
atoi	787	begin	575, 591
Aufzählungstyp	180	Benutzerdefinierte Literale	784
Ausgabe		Bereichsoperator	233
<i>unformatierte</i>	651	Bezeichner	51
Ausnahme		<i>reservierte</i>	52
<i>abfangen</i>	472	<i>Umlaute</i>	52
<i>weiterleiten</i>	478	Bibliotheken	26
<i>werfen</i>	471	bidirectional_iterator	595
Ausnahmebehandlung	467	Binäre Operatoren	
<i>catch</i>	472	<i>überladen</i>	361
<i>Fehlerklassen</i>	481	bind	639, 769
<i>Fehlermeldung</i>	496	Bit-Manipulationen	571
<i>noexcept</i>	501	bitset	571
<i>Stack-Unwinding</i>	477	bool	59
<i>terminate() behandeln</i>	497	boolalpha	660
<i>throw</i>	471	break	111, 125
<i>try</i>	470	Bucket	566
<i>what</i>	496		
Auswahloperator	110		
auto	243, 248, 327, 573		
auto (C++11)	68		
auto_ptr	790		

B

back_inserter	600, 712
bad	656
bad_alloc	493

C

C++11	
<i>vereinheitlichte Initialisierung</i>	55
c_str	676
Call by Reference	198
Call by Value	196
call_once	748
case	111
<i>break</i>	111

catch.....469, 472
 catch(...).....476
 cerr.....44, 648
 char.....62, 143
 Vorzeichen.....64
 char*
 const char*.....149
 char16_t.....66
 char32_t.....66
 chrono.....812
 cin.....44, 48, 648
 Clang.....32
 class.....167, 273
 clear.....656
 clog.....44, 648
 close.....677
 Code::Blocks.....28
 Compiler.....25, 31
 condition_variable.....753
 Condition-Variablen.....753
 const.....204, 244, 255, 314, 329
 extern.....245
 Iteratoren.....574
 Objekte.....315
 Referenzen.....775
 const_cast.....97, 244
 const_iterator.....594
 constexpr.....245, 248, 250, 255, 785
 Klassen.....343
 Containerklassen
 Adapterklassen.....553
 assoziative.....560, 627
 begin.....575
 Bit-Manipulation.....571
 emplace-Methoden.....576
 end.....575
 Methodenübersicht.....578
 Move-Semantik.....572
 Range-based for.....573
 sequenzielle.....550
 Standard-.....546, 550
 unordered-Versionen.....566
 continue.....126
 cout.....44, 648
 C-Style-Array.....135
 C-Style-Strings.....145
 ctor.....277
 cyclic reference.....801

D

Dangling Pointer.....792
 Datei-Streams.....672
 Fehlerüberprüfung.....674
 lesen.....678
 öffnen.....675
 schließen.....677
 schreiben.....678
 wahlfreier Zugriff.....686
 Datentypen.....51
 bool.....59
 char.....62
 double.....60
 float.....60
 Ganzzahl-.....55
 Gleitkommazahlen.....60
 Integer.....55
 konvertieren.....92
 long double.....60
 Deadlock.....740, 742
 vermeiden.....746
 dec.....662
 decltype.....223, 248, 327
 deep copy.....335
 default.....111, 248, 346
 defaultfloat.....663
 Default-Parameter.....204
 define.....250, 253
 Definition.....53
 Deklaration.....53
 Dekrement-Operator.....116
 delete.....161, 248, 348, 378
 Deprecated.....59
 deque.....552
 Designer Qt.....873
 Destruktor.....295
 Vererbung.....428
 virtueller.....442
 detach.....727
 Direkte Basisklasse.....409
 distance.....598
 do while.....119
 domain_error.....487
 double.....60
 duration.....815, 817
 duration_cast.....818
 dynamic linking.....439
 dynamic_cast.....96, 494

E

Eclipse.....	27
ECMAScript-Grammatik.....	696
Einfüge-Iteratoren.....	600
Eingabe	
<i>unformatierte</i>	652
Elementfunktionen.....	270
Element-Initialisierer.....	282
<i>Basis-Initialisierer</i>	431
else.....	108
else if.....	113
emplace_back.....	355
emplace-Methoden.....	576
empty.....	559
end.....	575, 591
endl.....	46, 660
Endlosschleife.....	124
ends.....	660
Entwicklungsumgebungen → IDE	
enum.....	255
<i>C-Style</i>	181
enum class	
<i>C++11</i>	181
eof.....	656
EOF (End of File).....	653
eofbit.....	655
error_code.....	491
exception.....	486
Exceptions	
<i>Ausnahmebehandlung</i>	467
exit.....	125, 221
EXIT_FAILURE.....	220
EXIT_SUCCESS.....	220
expception_ptr.....	499
explicit.....	247, 290
extern.....	54, 238
<i>const</i>	245

F

F.....	61
fabs.....	206
fail.....	656
failbit.....	655
failure.....	658
Fallunterscheidung.....	108, 111
false.....	59, 100

Fehlerbehandlung	
<i>Streams</i>	655, 675
Fehlerklassen.....	481
Felder → Arrays	
filebuf.....	694
final.....	247, 416, 455
fixed.....	663
Flache Kopie.....	334
flags.....	671
Fließkommazahlen.....	60
float.....	60
flush.....	652, 660
for.....	121, 573
<i>Range-based (C++11)</i>	123
forward.....	783
forward_iterator.....	595
forward_list.....	172, 552
friend.....	302, 324, 357
<i>Klassen</i>	357
<i>Operatoren überladen</i>	364
<i>unäre Operatoren überladen</i>	369, 386
front_inserter.....	600
fstream.....	673
Funktionen.....	187
<i>aufrufen</i>	189
<i>Call by Reference</i>	198
<i>Call by Value</i>	196
<i>definieren</i>	187
<i>deklarieren</i>	189
<i>friend-</i>	324
<i>Inline-</i>	217
<i>Lambda-</i>	218, 613
<i>Multithreading</i>	722
<i>neue Funktionssyntax</i>	221
<i>Parameter</i>	195
<i>Rückgabewert</i>	207
<i>Standardparameter</i>	204
<i>überladen</i>	215
<i>Zeiger</i>	200
Funktionsobjekte.....	395, 608
<i>Multithreading</i>	722
Funktionsoperator.....	392, 395
Funktionsparameter	
<i>const</i>	204
Funktions-Templates.....	508
<i>Spezialisierung</i>	513, 514, 515, 526
Funktor.....	608
Fußgesteuerte Schleife.....	119
Future.....	755, 757

G

g++	31
Ganzzahldatentypen	55
Ganzzahlen, Vorzeichen	58
GCC	31
gcc	31
Genauigkeit, Gleitkommazahlen	86
get	653, 679
get<>	634
get_id	729
get_terminate	498
getline	653, 681
Gleitkommazahlen	60
<i>Exponential-Schreibweise</i>	61
<i>Genauigkeit</i>	86
<i>Rundungsfehler</i>	85
<i>Vergleich auf 0</i>	206
good	656
goodbit	655
GUI	833
Gültigkeitsbereich	191
<i>globaler</i>	192
<i>lokaler</i>	193
<i>Namensbereich</i>	235
<i>static</i>	195

H

Hängender Zeiger	792
hardware_concurrency	733
has-a-relationship	409
Here doc	152
hex	662
hexfloat	663
high_resolution_clock	645, 813
Hilfsmittel	627

I

IDE	26
if	
<i>else</i>	108
<i>else if</i>	113
if (Anweisung)	100
ifstream	673
ignore	654
include	252

Indexoperator	397
Indirekte Basisklasse	409
Initialisieren	
<i>Klassen</i>	276
Initialisierung	54, 57
<i>vereinheitlichte</i>	55
initializer_list	828
Inkrement	
<i>-Operator</i>	116
inline	218, 247, 305
input_iterator	595
Insertor	
<i>Iterator</i>	600, 605
insertor	600
Instanz	274
int	56
Integer	55
Interface-Klassen	450
internal	663
invalid_argument	488
ios_base	648
ios_base::failure	492, 658
ios_state	655
iostate	655
iostream	44
is_open	678
is-a-relationship	409
is-relationship	434
Ist-Beziehung	409
istream_iterator	601
istreamstring	687
iterator	590
Iterator-Adapter	600
Iteratoren	546, 590
<i>begin</i>	591
<i>const</i>	574, 594
<i>end</i>	591
<i>für reguläre Ausdrücke</i>	712
<i>Hilfsfunktionen</i>	597
<i>Kategorien</i>	595
<i>Move-</i>	602
<i>Range-based for</i>	593
<i>Reverse-</i>	602
<i>Stream-</i>	601

J

join	727
joinable	728

K

KDevelop.....	30
Klassen.....	269
<i>ableiten</i>	408
<i>Ableiten verbieten</i>	416
<i>abstrakte</i>	446
<i>Ausnahme-</i>	481
<i>Basisklasse</i>	407
<i>class</i>	167
<i>constexpr</i>	343
<i>Eigenschaften</i>	273, 329
<i>entwerfen</i>	272
<i>final</i>	416
<i>friend</i>	357
<i>initialisieren</i>	276
<i>Instanziierung</i>	274
<i>Interface-</i>	450
<i>Move</i>	350
<i>polymorphe</i>	440
<i>-Prinzip</i>	271
<i>statische Eigenschaften</i>	338
<i>struct</i>	167
<i>Typumwandlung</i>	434
<i>verschieben</i>	290, 338, 350
<i>zerstören</i>	286, 295
<i>Zugriffskontrolle</i>	298
Klassen-Templates.....	523
<i>Default-Wert</i>	530, 532
<i>Non-Type-Parameter</i>	530
<i>Spezialisierung</i>	535
Kommandozeilenargumente.....	220
Kommentare.....	43
Komponenten-Hierarchie.....	869
Konstanten.....	58
Konstruktor.....	277
<i>delegieren</i>	283
<i>erben</i>	432
<i>Konvertierungs-</i>	290
<i>Kopier-</i>	285
<i>mit Parameter</i>	280
<i>Move-</i>	288, 350
<i>Standard-</i>	277
<i>Vererbung</i>	428
Kontrollstrukturen.....	99
Konvertieren, Datentypen.....	63, 77, 92
Konvertierungskonstruktor.....	290
Konvertierungsoperator.....	392
Kopfgesteuerte Schleife.....	117
Kopierkonstruktor.....	285

L

L.....	58, 61
Lambda-Funktionen.....	218, 613
<i>Multithreading</i>	722
<i>mutable</i>	616
late binding.....	439
Laufzeitfehler.....	489
Laufzeitklassen.....	553
left.....	663
length_error.....	488
Linker.....	26
list.....	172, 552
Literale.....	58
<i>benutzerdefinierte</i>	784
LL.....	58
locale.....	47
lock.....	725
lock_guard.....	741
Locks.....	741
logic_error.....	486, 487
Logische Algebra.....	105
Logische Operatoren.....	105
Logischer Ausdruck.....	100
long.....	56
long double.....	60
long long.....	56
long long int.....	56
LValues.....	774

M

main().....	38, 219
<i>Kommandozeilenargumente</i>	41
make_move_iterator.....	602
make_pair.....	214, 628
make_shared.....	795
make_tuple.....	633
Makros.....	253
Manipulatoren.....	660
map.....	142, 560
match_result.....	702
Mathematische Funktionen.....	89
Mehrfache Ableitung.....	409
Mehrfachvererbung.....	457
Memberfunktionen.....	271
Memory Leak.....	161
Mengenschleife.....	123
Meta Object Compiler Qt.....	855

Methoden	270, 274, 278, 293, 299, 303
<i>Ausgabe</i>	651
<i>const</i>	314
<i>Fehler bei Streams</i>	656
<i>für Threads</i>	727
<i>inline</i>	305
<i>Operatorüberladung</i>	371
<i>pure virtuelle</i>	445
<i>read-only</i>	314
<i>Redefinition</i>	426
<i>static</i>	341
<i>-Templates</i>	522
<i>this-Zeiger</i>	318
<i>überladen</i>	326
<i>virtuelle</i>	436
<i>Zugriffs-</i>	306
MinGW	29, 31
Modularisierung	227
Module	261
move	573, 642, 726, 780
Move-Iteratoren	602
Move-Konstruktor	288
Move-Semantik	355, 572, 773, 776
multimap	560
multiset	560
Multithreading	719
mutable	616
Mutex	724, 736
mutex	736

N

Namensraum	227
<i>erstellen</i>	229
<i>std</i>	229, 236
<i>verwenden</i>	232
Namespace	227
Narrowing	93
native_handle	739
NetBeans	27
new	160
next	599
NICHT, logisches	107
noboolalpha	660
noexcept	501
notify_all	754
notify_one	754
nounitbuf	661
nouppercase	661

nowshowbase	660
nowshowpoint	660
nowshowpos	660
nowskipws	660
nullptr	157
numeric_limits	73

O

Objektcode	25
Objektdatei	25
Objekte	
<i>erzeugen</i>	274, 312
<i>konstante</i>	315
Objektorientiertes Konzept	270
oct	662
ODER, logisches	106
ofstream	673
once_flag	748
OOP	269
open	678
openmode	675
operator	390
operator<<	648
operator=	379
<i>überladen</i>	373
<i>Vererbung</i>	419
operator>>	648
<i>überladen</i>	390
Operatoren	
<i>arithmetische</i>	78
<i>binäre</i>	380, 383
<i>bitweise</i>	383
<i>logische</i>	105, 383, 388
<i>ternäre</i>	110
<i>überladen</i>	359
<i>unäre</i>	80, 382
<i>Vergleich</i>	102
Operatorüberladung	
<i>friend-Funktion</i>	364
<i>Klassenmethode</i>	361
<i>Methoden</i>	371
<i>Regeln</i>	371
ostream_iterator	601
ostreamstringstream	687
out_of_range	132, 489
output_iterator	595
overflow_error	490
override	247, 453

P

packaged_task.....	755, 767
pair.....	627
pair<>.....	213
Perfect Forwarding.....	773, 781
Placement new	180
POD (Plain Old Data)	826
Pointer → Zeiger	
Polymorphie.....	436, 445
Postfix-Schreibweise.....	116
Prädikat	
<i>Algorithmen</i>	606
Präfix-Schreibweise.....	116
Präprozessor-Direktiven.....	251
Präprozessorlauf.....	251
prev	599
priority_queue.....	554
private.....	298, 417, 425
Promise.....	755, 760
protected.....	298, 417, 420
public.....	298, 417
Puffer, Streams	692
Pufferüberlauf.....	138
Pufferung.....	45
pure virtual	445
put.....	651, 679

Q

Qt.....	833
<i>Anwendungen lokalisieren</i>	998
<i>Bibliotheksaufbau</i>	835
<i>Container-Widgets</i>	940
<i>Dialog erstellen</i>	888
<i>Dialog mit Qt Designer</i>	909
<i>Direkthilfen</i>	968
<i>Eingabe-Widgets</i>	953
<i>fertige Dialoge</i>	896
<i>fertige Widgets verwenden</i>	930
<i>Hauptfenster</i>	969
<i>Hauptfenster mit Designer</i>	1006
<i>Icons</i>	1021
<i>Klassenhierarchie</i>	866
<i>Konsolenanwendung</i>	1019
<i>Layout</i>	880
<i>Menüleiste</i>	971
<i>Projekt erstellen</i>	838
<i>QAbstractButton</i>	930

<i>QAbstractSlider</i>	953
<i>QAbstractSpinBox</i>	962
<i>QAction</i>	975
<i>qApp</i>	861
<i>QApplication</i>	1019
<i>QCheckBox</i>	933
<i>QComboBox</i>	959
<i>QCoreApplication</i>	1019
<i>QDate</i>	1031
<i>QDateEdit</i>	962
<i>QDateTimeEdit</i>	962
<i>QDial</i>	953
<i>QDialog</i>	888
<i>QDockWidget</i>	987
<i>QDoubleSpinBox</i>	962
<i>QFileDialog</i>	902
<i>QFrame</i>	948
<i>QGridLayout</i>	881
<i>QGroupAction</i>	975
<i>QGroupBox</i>	941
<i>QHBoxLayout</i>	881
<i>QInputDialog</i>	907
<i>QLabel</i>	950
<i>QLCDNumber</i>	951
<i>QLineEdit</i>	956
<i>QMainWindow</i>	969
<i>QMenu</i>	971
<i>QMenuBar</i>	971
<i>QMessageBox</i>	896
<i>QObject</i>	866
<i>QProgressDialog</i>	952
<i>QPushButton</i>	868, 932
<i>QRadioButton</i>	937
<i>QSettings</i>	990
<i>QSlider</i>	953
<i>QSpinBox</i>	962
<i>QStackedLayout</i>	886
<i>QStatusBar</i>	980
<i>QString</i>	1028
<i>QStringList</i>	1030
<i>QTabWidget</i>	943
<i>QTextEdit</i>	958
<i>QTime</i>	1031
<i>QTimeEdit</i>	962
<i>QToolBar</i>	985
<i>QToolBox</i>	949
<i>QVariant</i>	1031
<i>QVBoxLayout</i>	881
<i>Ressourcen-System</i>	1022
<i>Schaltflächen</i>	930
<i>SDK</i>	836

<i>Signale</i>	846
<i>Slots</i>	846
<i>Speicherverwaltung</i>	869
<i>Spin-Boxen</i>	962
<i>Statuszeile</i>	980
<i>Statuszeilentipp</i>	966
<i>Textfeld</i>	956
<i>Tooltips</i>	967
<i>tr()</i>	972, 998
<i>Werkzeugleiste</i>	985
<i>Widgets anordnen</i>	880
<i>Widgets erstellen</i>	870
<i>Widgets mit Designer</i>	873
<i>Zustandsanzeige</i>	950
Qt Creator	836
Qt Designer	873
<i>Dialog erstellen</i>	880, 888, 909, 1006, 1015
<i>Hauptfenster-Anwendung</i>	1006
Qt Linguistic	998
Qt SDK	28
Quellcode	
<i>kommentieren</i>	43
Quelldatei	25
queue	554

R

Race Condition	735
RAD-Tool	
<i>Qt Designer</i>	873
RAII	333, 789
random_access_iterator	595
range_error	490
Range-based for	573
ratio	644, 815
Raw-String	152
rbegin	602
rdbuf	693
rdstate	656
read	653, 682
Rechnen	77
recursive_timed_mutex	738
Redefinition	413, 426, 440
Reference Collapsing	781, 783
Referenzen	164
<i>const</i>	775
<i>LValue-</i>	775
<i>RValue-</i>	776
regex	695
regex_error	486

regex_iterator	713
regex_match	706
regex_replace	711
regex_search	709
regex_token_iterator	715
register	244
Regular Expressions	695
Reguläre Ausdrücke	695
<i>Suchergebnis analysieren</i>	702
reinterpret_cast	97
rekursive_mutex	737
rel_ops	638
rend	602
resetiosflags	664
return	125, 207, 209, 221
reverse_iterator	602
Reverse-Iteratoren	602
right	663
Rohe Arrays	135
Rohstringlitterale	152
Rückgabewert	
<i>Funktionen</i>	207
Rundungsfehler	
<i>Gleitkommazahlen</i>	85
runtime_error	486, 489
RValue-Referenz	776
RValues	774

S

Schleifen	115
<i>do while</i>	119
<i>for</i>	121
<i>Range-based for</i>	135
<i>Range-based for (C++11)</i>	123
<i>while</i>	117
scientific	663
Scope-Operator	233
seekg	683, 686
seekp	686
Sequenzielle Container	550
set	560
set_terminate	498
setbase	664
setf	671
setfill	664
setiosflags	664
setprecision	664
setstate	656
setw	664

shallow copy	334	Streams	43, 647
Shared Pointer	791	<i>Datei-</i>	651, 672
shared_future	763	<i>Fehlerbehandlung</i>	655
shared_ptr	163, 789, 792	<i>manipulieren</i>	660
<i>make_shared</i>	795	<i>Standard-</i>	648
<i>weak_ptr</i>	801	<i>String</i>	687
short	56	streamsize	650
showbase	660	string	143
showpoint	660	<i>in C-Style-String umwandeln</i>	147
showpos	660	stringbuf	69
Signale		Strings	143
<i>Qt</i>	846, 870	<i>Raw</i>	152
signed	58	<i>rohe</i>	145
sizeof	72	<i>Streams</i>	687
skipws	660	<i>Unicode</i>	150
sleep_for	733	stringstream	687
sleep_until	733	struct	166, 167, 273
Slots		Strukturen	166
<i>Qt</i>	846	swap	320, 355, 733
Smart Pointer	642, 789	switch	111
smatch	702	<i>case</i>	111
sort	559	<i>default</i>	111
Speicherklassenattribute.....	238	Symbolische Konstanten	253
Speicherleck	161	Synonyme	183
Spezialisierung		system_clock.....	645, 812
<i>Funktions-Template</i>	515	system_error	491
Sprunganweisungen			
<i>break</i>	125		
<i>continue</i>	126		
stack	553		
Stack-Unwinding	478		
Standardausnahmen	485		
Standardcontainer-Klassen	546		
Standardkonstruktor	277		
Standardparameter	204		
Standard-Streams	648		
Standardzugriffsrechte	299		
static	195, 240, 338		
<i>Methoden</i>	341		
static_assert	72, 75		
static_cast	96, 521		
std:	39, 236, 237		
std::numeric_limits.....	73		
steady_clock	645, 813		
Steuerzeichen	64		
stoi	787		
str	688		
streambuf	693		
Stream-Iteratoren	601		
Stream-Objekte	44		
Stream-Puffer	692		

T

tellg.....	683, 686
tellp.....	686
Template	
<i>-Funktionen</i>	507
<i>Klassen</i>	507
<i>Methoden</i>	522
template	509, 516
Template-Programmierung	507
terminate	476, 497
terminate_handler.....	497
this	318
this_thread::get_id	729
thread	722
thread_local.....	750
Threads	
<i>Argumente übergeben</i>	723
<i>asynchrones Arbeiten</i>	755
<i>Locks</i>	741
<i>Methoden</i>	727
<i>move</i>	726
<i>synchronisieren</i>	734

throw 469, 471
 tie 634
 Tiefe Kopie 335
 time_point 813
 time_since_epoch 813
 timed_mutex 738
 true 59, 100
 try 469, 470
 try_lock_for 738
 try_lock_until 738
 tuple 215, 542, 632
 type_info 451
 typedef 183
 typeid 451, 494
 typename 509
 class 509
 Type-Traits 820
 Typinformationen 451
 Typqualifikatoren 244
 Typumwandlung
 einschränken 95
 explizite 96
 implizite 92
 Klassen 434

U

U 58
 u6string 150
 u32string 150
 Überladen
 binäre Operatoren 361
 Funktionen 215
 Methoden 326
 Operatoren 359
 unäre Operatoren 366
 Überlauf 81
 UL 58
 ULL 58
 Umlaute 46, 52
 Unäre Operatoren 80
 überladen 366
 uncaught_exception 499
 UND
 logisches 105
 undef 253
 underflow_error 490
 Unformatierte Ausgabe 651
 Unformatierte Eingabe 652
 Unicode 150
 Unicode-Unterstützung 65

union 177
 Unions 177
 Unique Pointer 804
 unique_lock 743
 unique_ptr 161, 162, 789, 804
 unitbuf 661
 unlock 725
 unordered_map 142, 561
 unordered_multimap 561
 unordered_multiset 561
 unordered_set 561
 unsetf 671
 unsigned 58
 uppercase 661
 using 185, 234
 UTF-16 67
 UTF-32 67
 UTF-8 67

V

Variable Argumentanzahl
 Templates 520, 530, 539
 Variadic Templates 539
 vector 130, 552
 Vereinheitlichte Initialisierung 55
 Vererbung 407
 Destruktor 428
 Konstruktor 428
 Mehrfach- 457
 operator= 419
 private 417
 protected 417
 public 417
 Zugriffsschutz 417
 Vergleichsoperatoren 102, 386, 638
 Verkettete Liste 172
 Verschiebekonstruktor 288, 313, 350
 Verschieben
 move() 642
 Verzweigung
 if else 108
 mehrfache 111, 113
 virtual 247, 439
 Virtuelle Methoden 436
 Virtueller Destruktor 442
 Visual Studio 28
 void 187
 volatile 246
 Vorausdeklaration
 Funktionen 189

W

Wahlfreier Zugriff, Datei-Streams..... 686

wait..... 754

wchar_t..... 65

Weak Pointer..... 801

weak_ptr..... 789, 803

Wertebereich

 Überlauf..... 81

what..... 486, 496

while..... 117

width..... 672

write..... 651, 682

ws..... 660

wstring..... 150

X

Xcode..... 31

Y

yield..... 733

Yoda Conditions..... 104

Z

Zähloperatoren..... 116

Zählschleife..... 121

Zeichen..... 62

Zeichenketten..... 143

Zeichenliteral..... 63

Zeichensatz..... 62

Zeiger..... 153

 Funktionsparameter..... 200

 this..... 318

Zeitbibliothek..... 812

Zeitdauer..... 815

Zeitgeber..... 812

Zufallszahlen..... 90

Zugriffskontrolle

 Klassen..... 298

Zugriffsmethoden..... 306

Zugriffsrechte

 Standard-..... 299

Zugriffsschutz..... 411, 417

Zuweisung

 verbieten..... 378

Zuweisungsoperator..... 373, 379

 mehrfach überladen..... 376

Zyklische Referenzen..... 801