jetzt lerne ich

Start ohne Vorwissen

C++

Das komplette Starterkit für den einfachen Einstieg in die Programmierung

JESSE LIBERTY DAVID HORVATH

Markt+Technik

Borland C++BuilderX Personal,
alle Quellcodes und Lösungen zum Buch

jetzt lerne ich

C++

Unser Online-Tipp
für noch mehr Wissen ...

... aktuelles Fachwissen rund
um die Uhr – zum Probelesen,
Downloaden oder auch auf Papier.

www.InformIT.de

jetzt lerne ich

C++

Das komplette Starterkit für den einfachen Einstieg in die Programmierung

JESSE LIBERTY DAVID HORVATH

Markt+Technik

eBook

Die nicht autorisierte Weitergabe dieses eBooks an Dritte ist eine Verletzung des Urheberrechts!

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

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Hardware- und Software-Bezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig auch eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:
Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.

10 9 8 7 6 5 4 3 2 1

07 06 05

ISBN 3-8272-6830-3

© 2005 by Markt+Technik Verlag,
ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten
Lektorat: Frank Eller, feller@pearson.de
Herstellung: Claudia Bäurle, cbaeurle@pearson.de
Coverkonzept: independent Medien-Design
Coverlayout: adesso 21, Thomas Arlt
Titelillustration: Karin Drexler
Satz: text&form GbR, Fürstenfeldbruck
Druck und Verarbeitung: Bosch Druck, Ergolding
Printed in Germany

Übersicht

Einführung		19
1	Erste Schritte	23
2	Die Teile eines C++-Programms	41
3	Variablen und Konstanten	57
4	Ausdrücke und Anweisungen	75
5	Funktionen	101
6	Programmsteuerung	127
7	Klassen – die Grundlagen	151
8	Mehr über Klassen	169
9	Zeiger	183
10	Zeiger – weiterführende Themen	203
11	Referenzen	217
12	Referenzen und Zeiger – weiterführende Themen	235
13	Funktionen – weiterführende Themen	253
14	Überladen von Operatoren	267
15	Arrays	287
16	Vererbung	311
17	Polymorphie und abgeleitete Klassen	335
18	Polymorphie – weiterführende Themen	351
19	Verkettete Listen	375
20	Spezielle Klassen, Funktionen und Zeiger	391
21	Der Präprozessor	427
22	Objektorientierte Analyse und objektorientierter Entwurf	457
23	Templates	489
24	Exceptions und Fehlerbehandlung	509
Anhang A: Zahlensysteme		533
Anhang B: Glossar		543
Stichwortverzeichnis		553

»A programmer is just a tool which converts caffeine into code«

(anonym)

Im Sinne dieses bekannten und vielsagenden Zitats widmen Ihnen die Autoren und Lektoren der Buchreihe »Jetzt lerne ich« in jeder Ausgabe ein Rezept mit oder rund um das belebende und beliebte Getränk. Sollten Sie gerade ohne Bohnen oder Pulver sein: Über die Adresse *http://www.mut.de/jetzt-lerne-ich* können Sie einen eigens entwickelten Markt+Technik Programmiererkaffee bestellen.

Viel Spaß und Genuß!

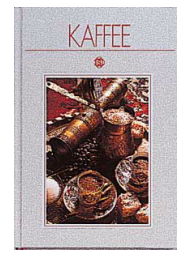
Pharisäer

(1 Tasse)
1 Teelöffel Kakaopulver
1 Teelöffel Zucker
1 Likörglas Rum
3/4 Tasse heißer starker Kaffee
Schlagsahne

In angewärmter Tasse Zucker, Kakao und Rum verrühren. Dann den heißen Kaffee dazugießen und mit reichlich Schlagsahne zudecken.

Zu diesem Kaffeerezept gehört eine lustige Geschichte: Auf einer Insel im äußersten Norden Deutschlands hielt ein neuer Pfarrer Einzug. Er hatte keineswegs etwas gegen Kaffee einzuwenden, umso mehr aber wetterte er gegen Alkohol. Bei einem Fest kredenzte der Hausherr Kaffee, der aber keine rechte Stimmung aufkommen lassen wollte. Deshalb setzte er heimlich Rum zu, den Kaffee des Herrn Pfarrers natürlich sorgsam auslassend. Um verräterische Düfte des Alkohols zu verbergen, wurde auf die Kaffeetasse eine dicke Haube Schlagsahne gesetzt. Die Stimmung stieg zusehends. Als man die Tasse des Herrn Pfarrers verwechselte, kam der Schwindel auf und der Gottesmann beschimpfte den Hausherrn als Heuchler und Pharisäer. Wenn sie nicht ganz wahr sein sollte, diese Geschichte, so ist sie auf jeden Fall gut erfunden.

Das Kaffeerezept wurde entnommen aus:



»Kaffee«
Dr. Eugen C. Bürgin
Sigloch Edition, Blaufelden
ISBN: 3-89393-135-X

Mit freundlicher Genehmigung des Verlags.

Inhaltsverzeichnis

Einführung		19
1	**Erste Schritte**	23
1.1	Vorbereitungen	23
1.2	Klarheit schaffen: C++, ANSI C++, ISO C++ und Windows	24
1.2.1	Den Compiler installieren und einrichten	25
1.2.2	Den Borland C++BuilderX-Compiler konfigurieren	27
1.2.3	Kompilieren mit dem Borland C++BuilderX-Compiler	29
1.2.4	Compiler und Editor	30
1.3	Quellcode kompilieren und linken	31
1.3.1	Kompilieren in einer integrierten Entwicklungsumgebung	32
1.3.2	Ein Programm linken	32
1.4	Der Entwicklungszyklus	33
1.5	HELLO.CPP – Ihr erstes C++-Programm	33
1.6	Fehler zur Kompilierzeit	36
1.7	Fragen und Antworten	37
1.8	Workshop	38
1.8.1	Kontrollfragen	38
1.8.2	Übungen	38
1.8.3	Antworten zu den Kontrollfragen	38
2	**Die Teile eines C++-Programms**	41
2.1	Warum C++ die richtige Wahl ist	41
2.1.1	Prozedurale, strukturierte und objektorientierte Programmierung	43

2.1.2	C++ und objektorientierte Programmierung	44
2.2	Die Teile eines einfachen Programms	45
2.2.1	Die Anweisung `#include` im Detail	46
2.2.2	Zeilenweise Analyse	47
2.3	Kommentare	48
2.3.1	Arten von Kommentaren	49
2.3.2	Kommentare in einem einfachen Beispielprogramm	49
2.4	Funktionen	50
2.4.1	Funktionen aufrufen	51
2.4.2	Funktionen verwenden	52
2.4.3	Parameter in Funktionen	53
2.5	Fragen und Antworten	54
2.6	Workshop	54
2.6.1	Kontrollfragen	55
2.6.2	Übungen	55
2.6.3	Antworten zu den Kontrollfragen	55
3	**Variablen und Konstanten**	57
3.1	Was ist eine Variable?	57
3.1.1	Speicher reservieren	58
3.1.2	Größe von Integer-Werten	58
3.1.3	Mit und ohne Vorzeichen – signed und unsigned	60
3.1.4	Grundlegende Variablentypen	60
3.2	Variablen definieren	62
3.2.1	Groß-/Kleinschreibung	62
3.2.2	Schlüsselwörter	63
3.3	Mehrere Variablen gleichzeitig erzeugen	64
3.4	Werte an Variablen zuweisen	64
3.5	Das Schlüsselwort typedef	66
3.6	Wann verwendet man short und wann long?	67
3.6.1	Bereichsüberschreitung bei Integer-Werten vom Typ unsigned	68
3.6.2	Bereichsüberschreitung bei Integer-Werten vom Typ signed	69
3.7	Konstanten	70
3.7.1	Literale Konstanten	70
3.7.2	Symbolische Konstanten	70
3.8	Aufzählungskonstanten	71
3.9	Fragen und Antworten	72
3.10	Workshop	73
3.10.1	Kontrollfragen	73
3.10.2	Übungen	73
3.10.3	Antworten zu den Kontrollfragen	74

4	**Ausdrücke und Anweisungen**	75
4.1	Anweisungen	75
4.1.1	Whitespaces	76
4.1.2	Verbundanweisungen	76
4.2	Ausdrücke	77
4.3	Operatoren	79
4.3.1	Zuweisungsoperator	79
4.3.2	Mathematische Operatoren	79
4.4	Zusammengesetzte Operatoren	80
4.5	Inkrementieren und Dekrementieren	81
4.5.1	Präfix und Postfix	82
4.6	Rangfolge: die Reihenfolge der Operationen	84
4.7	Verschachtelte Klammern	85
4.8	Vergleichsoperatoren	86
4.9	Die if-Anweisung	88
4.9.1	Die `else`-Klausel	88
4.9.2	Erweiterte `if`-Anweisungen	90
4.10	Geschweifte Klammern in verschachtelten if-Anweisungen	92
4.11	Logische Operatoren	95
4.11.1	Logisches AND	95
4.11.2	Logisches OR	95
4.11.3	Logisches NOT	96
4.12	Rangfolge der Vergleichsoperatoren	96
4.13	Mehr über Wahr und Unwahr	97
4.14	Fragen und Antworten	97
4.15	Workshop	98
4.15.1	Kontrollfragen	98
4.15.2	Übungen	99
4.15.3	Antworten zu den Kontrollfragen	99
5	**Funktionen**	101
5.1	Was ist eine Funktion?	101
5.2	Funktionen deklarieren und definieren	102
5.2.1	Funktionen deklarieren	103
5.2.2	Funktionen definieren	105
5.2.3	Funktionen	106
5.3	Variablen in Funktionen	107
5.3.1	Lokale Variablen	107
5.3.2	Globale Variablen	110
5.4	Funktionsargumente	110
5.4.1	Funktionen als Parameter in Funktionen	111
5.5	Parameter sind auch lokale Variablen	112

5.6	Rückgabewerte	113
5.7	Standardparameter	116
5.8	Funktionen überladen	118
5.8.1	Inline-Funktionen	119
5.8.2	Stack und Funktionen	122
5.9	Fragen und Antworten	124
5.10	Workshop	125
5.10.1	Kontrollfragen	125
5.10.2	Übungen	125
5.10.3	Antworten zu den Kontrollfragen	125
6	**Programmsteuerung**	127
6.1	Schleifenkonstruktionen	127
6.1.1	Ursprung der Schleifen – Konstruktionen mit `goto`	127
6.1.2	Warum man `goto` nicht verwendet	128
6.2	while-Schleifen	128
6.2.1	Komplexere `while`-Anweisungen	129
6.2.2	Die Anweisungen continue und break	131
6.2.3	while(1)-Schleifen	133
6.3	do…while-Schleifen	135
6.4	for-Schleifen	137
6.4.1	Initialisierung, Test und Inkrementierung	138
6.4.2	Erweiterte `for`-Schleifen	139
6.4.3	Leere `for`-Schleifen	142
6.4.4	Verschachtelte Schleifen	143
6.5	switch-Anweisungen	145
6.6	Fragen und Antworten	147
6.7	Workshop	148
6.7.1	Kontrollfragen	148
6.7.2	Übungen	148
6.7.3	Antworten zu den Kontrollfragen	149
7	**Klassen – die Grundlagen**	151
7.1	Was ist ein Typ?	151
7.2	Neue Typen erzeugen	152
7.3	Klassen und Elemente	152
7.3.1	Klassen deklarieren	153
7.3.2	Ein Wort zu Namenskonventionen	154
7.3.3	Objekte definieren	155
7.3.4	Klassen und Objekte	155
7.4	Auf Klassenelemente zugreifen	155
7.4.1	An Objekte, nicht an Klassen zuweisen	155

7.5	Die Schlüsselwörter private und public	156
7.6	Klassenmethoden implementieren	158
7.7	Konstruktoren und Destruktoren – Objekte erstellen und löschen	161
7.7.1	Standardkonstruktoren	162
7.7.2	Vom Compiler bereitgestellte Konstruktoren	162
7.8	Fragen und Antworten	165
7.9	Workshop	166
7.9.1	Kontrollfragen	166
7.9.2	Übungen	166
7.9.3	Antworten zu den Kontrollfragen	166
8	**Mehr über Klassen**	169
8.1	Konstante Elementfunktionen	169
8.2	Schnittstelle und Implementierung	170
8.3	Klassendeklarationen und Methodendefinitionen platzieren	171
8.4	Inline-Implementierung	173
8.5	Klassen als Datenelemente einer Klasse	175
8.6	Fragen und Antworten	181
8.7	Workshop	181
8.7.1	Kontrollfragen	182
8.7.2	Übungen	182
8.7.3	Antworten zu den Kontrollfragen	182
9	**Zeiger**	183
9.1	Zeiger und ihre Verwendung	183
9.1.1	Die Adresse in einem Zeiger speichern	187
9.1.2	Zeigernamen	189
9.1.3	Der Indirektionsoperator	189
9.1.4	Zeiger, Adressen und Variablen	190
9.1.5	Daten mithilfe von Zeigern manipulieren	191
9.1.6	Adressen untersuchen	192
9.2	Warum man Zeiger verwendet	194
9.3	Stack und Heap	195
9.3.1	Das Schlüsselwort new	196
9.3.2	Das Schlüsselwort delete	197
9.3.3	Speicherlücken	199
9.4	Fragen und Antworten	201
9.5	Workshop	201
9.5.1	Kontrollfragen	201
9.5.2	Übungen	201
9.5.3	Antworten zu den Kontrollfragen	202

10	**Zeiger – weiterführende Themen**	203
10.1	Objekte auf dem Heap erzeugen	203
10.2	Objekte löschen	204
10.3	Mit Zeigern auf Datenelemente zugreifen	205
10.4	Datenelemente auf dem Heap	206
10.5	Der Zeiger this	208
10.6	Wilde Zeiger	210
10.7	Konstante Zeiger	211
10.7.1	Konstante Zeiger und Elementfunktionen	212
10.7.2	Konstante this-Zeiger	214
10.8	Fragen und Antworten	214
10.9	Workshop	214
10.9.1	Kontrollfragen	214
10.9.2	Übungen	215
10.9.3	Antworten zu den Kontrollfragen	215
11	**Referenzen**	217
11.1	Was ist eine Referenz?	217
11.2	Referenzen erzeugen	218
11.3	Der Adressoperator bei Referenzen	219
11.4	Was kann man referenzieren?	223
11.5	Null-Zeiger und Null-Referenzen	223
11.6	Funktionsargumente als Referenz übergeben	224
11.6.1	Parameter mit Zeigern übergeben	225
11.6.2	Parameter mit Referenzen übergeben	226
11.7	Header und Prototypen von Funktionen	228
11.8	Mehrere Werte zurückgeben	229
11.8.1	Werte als Referenz zurückgeben	231
11.9	Fragen und Antworten	232
11.10	Workshop	233
11.10.1	Kontrollfragen	233
11.10.2	Übungen	233
11.10.3	Antworten zu den Kontrollfragen	233
12	**Referenzen und Zeiger – weiterführende Themen**	235
12.1	Übergabe als Referenz aus Gründen der Effizienz	235
12.1.1	Einen konstanten Zeiger übergeben	239
12.2	Referenzen als Alternative	241
12.3	Wann verwendet man Referenzen und wann Zeiger?	243
12.4	Referenzen auf nicht mehr vorhandene Objekte	244
12.5	Referenzen auf Objekte im Heap zurückgeben	246
12.6	Wem gehört der Zeiger?	249
12.7	Fragen und Antworten	250

12.8	Workshop	250
12.8.1	Kontrollfragen	250
12.8.2	Übungen	251
12.8.3	Antworten zu den Kontrollfragen	251
13	**Funktionen – weiterführende Themen**	253
13.1	Überladene Elementfunktionen	253
13.2	Standardwerte	256
13.3	Standardwerte oder überladene Funktionen?	258
13.4	Konstruktoren überladen	258
13.5	Objekte initialisieren	259
13.6	Der Kopierkonstruktor	260
13.7	Fragen und Antworten	264
13.8	Workshop	265
13.8.1	Kontrollfragen	265
13.8.2	Übungen	266
13.8.3	Antworten zu den Kontrollfragen	266
14	**Überladen von Operatoren**	267
14.1	Operatoren überladen	267
14.1.1	Eine Inkrement-Funktion schreiben	268
14.1.2	Den Postfix-Operator überladen	270
14.1.3	Unterschied zwischen Präfix und Postfix	270
14.1.4	Der Operator operator+	272
14.1.5	Den operator+ überladen	274
14.1.6	Einschränkungen beim Überladen von Operatoren	276
14.1.7	Was überlädt man?	276
14.1.8	Der operator =	276
14.2	Umwandlungsoperatoren	279
14.2.1	Der Operator int()	282
14.3	Fragen und Antworten	283
14.4	Workshop	284
14.4.1	Kontrollfragen	284
14.4.2	Übungen	284
14.4.3	Antworten zu den Kontrollfragen	284
15	**Arrays**	287
15.1	Was ist ein Array?	287
15.2	Array-Elemente	288
15.3	Array-Grenzen überschreiten	289
15.4	Fence Post Errors	290
15.5	Arrays initialisieren	291
15.6	Arrays von Objekten	292

15.7	Mehrdimensionale Arrays	293
15.7.1	Mehrdimensionale Arrays initialisieren	295
15.8	Ein Wort zum Speicher	296
15.9	Arrays von Zeigern	297
15.10	Arrays auf dem Heap	299
15.11	Zeiger auf ein Array und ein Array von Zeigern	300
15.12	Zeiger und Array-Namen	300
15.13	Arrays auf dem Heap löschen	302
15.14	char-Arrays	303
15.15	Die Funktionen strcpy() und strncpy()	306
15.16	String-Klassen	307
15.17	Fragen und Antworten	308
15.18	Workshop	308
15.18.1	Kontrollfragen	308
15.18.2	Übungen	309
15.18.3	Antworten zu den Kontrollfragen	309
16	**Vererbung**	311
16.1	Was ist Vererbung?	311
16.1.1	Vererbung und Ableitung	312
16.1.2	Das Tierreich	313
16.1.3	Die Syntax der Ableitung	314
16.2	Private und Protected	316
16.3	Konstruktoren und Destruktoren	318
16.3.1	Argumente an Konstruktoren der Basisklasse übergeben	321
16.4	Funktionen redefinieren	326
16.4.1	Überladen und Überschreiben	328
16.4.2	Die Methode der Basisklasse verbergen	328
16.4.3	Die Basismethode aufrufen	330
16.5	Fragen und Antworten	332
16.6	Workshop	332
16.6.1	Kontrollfragen	333
16.6.2	Übungen	333
16.6.3	Antworten zu den Kontrollfragen	333
17	**Polymorphie und abgeleitete Klassen**	335
17.1	Polymorphie mit virtuellen Methoden	335
17.1.1	Arbeitsweise virtueller Elementfunktionen	340
17.1.2	Verbotene Pfade	341
17.1.3	Slicing (Datenteilung)	342
17.1.4	Virtuelle Destruktoren	344
17.1.5	Virtuelle Kopierkonstruktoren	344
17.1.6	Der Preis der virtuellen Methoden	348

17.2	Fragen und Antworten	349
17.3	Workshop	349
17.3.1	Kontrollfragen	349
17.3.2	Übungen	350
17.3.3	Antworten zu den Kontrollfragen	350
18	**Polymorphie – weiterführende Themen**	351
18.1	Probleme mit einfacher Vererbung	351
18.2	Abstrakte Datentypen	356
18.2.1	Abstrakte Funktionen	360
18.2.2	Komplexe Abstraktionshierarchien	367
18.2.3	Welche Typen sind abstrakt?	372
18.3	Fragen und Antworten	372
18.4	Workshop	373
18.4.1	Kontrollfragen	373
18.4.2	Übungen	373
18.4.3	Antworten zu den Kontrollfragen	374
19	**Verkettete Listen**	375
19.1	Verkettete Listen und andere Strukturen	375
19.2	Eine Fallstudie	377
19.2.1	Delegierung von Verantwortlichkeit	377
19.3	Die Einzelkomponenten	377
19.4	Was haben wir gelernt?	387
19.5	Fragen und Antworten	388
19.6	Workshop	388
19.6.1	Kontrollfragen	389
19.6.2	Übungen	389
19.6.3	Antworten zu den Kontrollfragen	389
20	**Spezielle Klassen, Funktionen und Zeiger**	391
20.1	Statische Datenelemente	391
20.2	Statische Elementfunktionen	394
20.3	Containment von Klassen	396
20.3.1	Auf Elemente der enthaltenen Klasse zugreifen	402
20.3.2	Zugriff auf enthaltene Elemente filtern	402
20.3.3	Nachteile des Containments	403
20.3.4	Als Wert und als Referenz kopieren	403
20.4	Friend-Klassen	404
20.5	Friend-Funktionen	405
20.6	Zeiger auf Funktionen	405
20.6.1	Kurzaufruf	408
20.6.2	Arrays mit Zeigern auf Funktionen	409

20.6.3	Zeiger auf Funktionen an andere Funktionen übergeben	412
20.6.4	typedef bei Zeigern auf Funktionen	415
20.7	Zeiger auf Elementfunktionen	418
20.7.1	Arrays mit Zeigern auf Elementfunktionen	421
20.8	Fragen und Antworten	423
20.9	Workshop	424
20.9.1	Kontrollfragen	424
20.9.2	Übungen	424
20.9.3	Antworten zu den Kontrollfragen	424
21	**Der Präprozessor**	427
21.1	Präprozessor und Compiler	427
21.2	Das Zwischenformat ansehen	428
21.3	Die Anweisung #define	428
21.3.1	Konstanten mit #define erzeugen	428
21.3.2	Testen mittels #define und #ifdef	429
21.3.3	Der Präprozessor-Befehl #else	429
21.4	Schutz vor Mehrfachdeklarationen	431
21.4.1	Definitionen auf der Befehlszeile	432
21.4.2	Definitionen aufheben	432
21.4.3	Bedingte Kompilierung	432
21.5	Makros	433
21.5.1	Warum so viele Klammern?	434
21.5.2	Makros, Funktionen und Templates	435
21.6	String-Manipulation	436
21.6.1	Zeichenkettenbildung	436
21.6.2	Verkettung	436
21.7	Vordefinierte Makros	437
21.7.1	Das Makro assert()	438
21.8	Fragen und Antworten	455
21.9	Workshop	455
21.9.1	Kontrollfragen	456
21.9.2	Übungen	456
21.9.3	Antworten zu den Kontrollfragen	456
22	**Objektorientierte Analyse und objektorientierter Entwurf**	457
22.1	Der Entwicklungszyklus	457
22.2	Ein Alarmsystem simulieren	458
22.2.1	Konzeptionelle Phase	459
22.2.2	Analyse und Anforderungen	459
22.2.3	High-Level- und Low-Level-Entwurf	460
22.2.4	Andere Objekte	461

22.2.5	Wie sehen die Klassen aus?	461
22.2.6	Wie wird ein Alarm gemeldet?	462
22.2.7	Ereignisschleifen	463
22.3	PostMaster: Eine Fallstudie	466
22.3.1	Wegweisende Entscheidungen	466
22.3.2	Teile und herrsche	467
22.3.3	Nachrichtenformat	468
22.3.4	Anfänglicher Klassenentwurf	468
22.3.5	Hierarchien mit und ohne Wurzel	470
22.3.6	Die Schnittstellen entwerfen	472
22.3.7	Einen Prototyp erstellen	473
22.3.8	Die 80/80-Regel	474
22.3.9	Entwurf der Klasse PostMasterMessage	475
22.3.10	Die API	475
22.3.11	Team-Arbeit	477
22.3.12	Weiterführende Betrachtungen zum Entwurf	477
22.3.13	Rahmenprogramme	479
22.4	Fragen und Antworten	487
22.5	Workshop	487
22.5.1	Kontrollfragen	487
22.5.2	Übungen	488
22.5.3	Antworten zu den Kontrollfragen	488
23	**Templates**	489
23.1	Was sind Templates?	489
23.2	Instanzen der Templates	490
23.3	Definition von Templates	490
23.4	Template-Elemente	498
23.5	Die Standard Template Library	506
23.6	Fragen und Antworten	507
23.7	Workshop	507
23.7.1	Kontrollfragen	507
23.7.2	Übungen	508
23.7.3	Antworten zu den Kontrollfragen	508
24	**Exceptions und Fehlerbehandlung**	509
24.1	Fehler	509
24.1.1	Das Unerwartete behandeln	511
24.2	Exceptions	511
24.2.1	Komponenten einer Exception-Behandlung	512
24.3	try-Blöcke und catch-Blöcke einsetzen	517
24.3.1	Exceptions abfangen	517
24.3.2	Mehrere catch-Spezifikationen	518

24.3.3 Abfangen als Referenz und Polymorphie 518
24.4 Wie geht es weiter? 523
24.4.1 Geschweifte Klammern 523
24.4.2 Lange Zeilen 524
24.4.3 switch-Anweisungen 524
24.4.4 Programmtext 525
24.4.5 Namen von Bezeichnern 526
24.4.6 Schreibweise von Namen 526
24.4.7 Kommentare 527
24.4.8 Zugriff 527
24.4.9 Klassendefinitionen 528
24.4.10 include-Dateien 528
24.4.11 assert()-Anweisungen 528
24.4.12 const-Deklarationen 529
24.5 Wie geht es weiter? 529
24.5.1 Hilfe und Rat 529
24.5.2 Literatur 529
24.6 Fragen und Antworten 530
24.7 Workshop 531
24.7.1 Kontrollfragen 531
24.7.2 Übung 531
24.7.3 Antworten zu den Kontrollfragen 531

Anhang A: Zahlensysteme 533

Anhang B: Glossar 543

Stichwortverzeichnis 553

Einführung

Dieses Buch soll Ihnen helfen, mit C++ zu programmieren. In nur 24 Lektionen zu je einer Stunde lernen Sie die Grundlagen zu Themen wie Ein-/Ausgabe, Schleifen und Arrays, objektorientierte Programmierung, Templates und Erstellen von C++-Anwendungen – alles in einem didaktisch gut aufbereiteten Rahmen. Die Listings demonstrieren zusammen mit Beispielausgaben und einer Analyse des Codes die Themen der jeweiligen Lektion. Als praktische Referenz dienen die gekennzeichneten Syntaxbeispiele.

Damit Ihnen der vermittelte Stoff geläufiger wird, schließt jede Stunde mit einer Reihe von Fragen und Antworten.

Neu in dieser Ausgabe des Buches ist der Workshop, der sowohl Kontrollfragen (einschließlich der Antworten) als auch Übungen umfasst.

Für wen ist dieses Buch gedacht?

Das Buch setzt keinerlei Programmierkenntnisse voraus. Es beginnt bei den Grundlagen und vermittelt Ihnen sowohl die Sprache als auch die Konzepte, die mit der Programmierung in C++ verbunden sind. Zahlreiche Syntaxbeispiele und detaillierte Codeanalysen begleiten Sie, wenn Sie in die faszinierende Welt der Programmierung eintauchen. Ob Sie nun ganz am Anfang stehen oder bereits etwas Programmiererfahrung haben – der klare Aufbau des Buches erleichtert Ihnen das Erlernen von C++.

Sollte ich zuerst C lernen?

Es drängt sich die Frage auf: »Soll ich zuerst C lernen, weil C++ eine Obermenge von C ist?« Stroustrup und die meisten C++-Programmierer stimmen überein: Es ist nicht nur nicht erforderlich, zuerst C zu lernen, sondern auch eine schlechte Empfehlung, da Sie sich dabei Programmiergewohnheiten aneignen könnten, die C++ vermeiden soll. Dieses Buch geht davon aus, dass Sie kein C-Programmierer sind. Andernfalls wäre das aber auch kein großes Problem. Lesen Sie die ersten Stunden als Wiederholung und legen Sie sich dann richtig ins Zeug. Dabei werden Sie feststellen, dass die Sprache C in einer Weise erweitert wurde, die Sie nicht für möglich gehalten hätten.

Warum sollte ich C++ lernen?

Das ist eine andere häufig gestellte Frage, die gewöhnlich aus Feststellungen wie »Letztlich ist Sprache X das Neueste und Beste und wird C++ ersetzen« herrührt. Dem kann man nur entgegnen: »Das stimmt nicht!« Viele Sprachen hatten das Ziel, COBOL zu ersetzen, aber es gibt derzeit immer noch Millionen Codezeilen von COBOL-Anwendungen (und es kommen ständig neue hinzu). Überall treffen Sie auf C++-Code und der riesige Fundus an C++-Programmen nimmt täglich zu. Doch selbst wenn Sie glauben, dass die Zukunftsaussichten von C++ nicht rosig sind, ist es immer noch eine Sprache, die es sich zu lernen lohnt. Denn viele der »neuen und verbesserten« Sprachen lehnen sich zu einem nicht geringen Prozentsatz an C++ an. Sowohl Java als auch C# haben syntaktische Gemeinsamkeiten mit C++, sind aber in mancher Hinsicht schwerer zu erlernen.

Wenn Sie C++ lernen, hilft Ihnen das auch beim Erlernen ähnlicher Sprachen. Das gilt nicht nur für Java und C#, sondern auch für awk, Perl und natürlich C. Vor allem machen Sie sich mit portabler Programmierung vertraut, die Sie bei nahezu jeder Plattform anwenden können, die heute auf dem Markt ist: angefangen bei PCs über große Unix-Systeme bis hin zu Mainframes, die ganze Rechenzentren füllen. Auch wenn Ihr Code nicht hundertprozentig portabel ist (d.h. sich nicht gänzlich ohne Änderungen auf andere Plattformen übertragen lässt), wissen Sie doch, wie das Ganze funktioniert!

Konventionen in diesem Buch

Besondere Textpassagen sind im Buch mit folgenden Elementen gekennzeichnet:

Hier erhalten Sie Informationen, wie Sie effizienter in C++ programmieren.

Weist Sie auf Probleme oder Seiteneffekte hin, die in speziellen Situationen auftreten können oder weist Sie auf Fehlerquellen und Stolpersteine bei der Programmierung hin.

Gibt Ihnen Tipps und zusätzliche Informationen zum gerade behandelten Thema.

Was Sie tun sollten

Hier finden Sie eine knappe Zusammenfassung der grundlegenden Prinzipien, die diese Stunde behandelt hat.

Unter diesem Symbol finden Sie Beispielausgaben des im Listing dargestellten Programms.

Dieser Textabschnitt erläutert das im Listing angegebene Codebeispiel.

Außerdem sind folgende Textelemente durch eine andere Schrift gekennzeichnet:

- C++-Code oder einzelne Schlüsselwörter sind vom laufenden Text durch `Schreibmaschinenschrift` unterschieden. Platzhalter – Wörter oder Zeichen, die symbolisch für tatsächlich einzugebende Wörter oder Zeichen stehen – werden in *`kursiver Schreibmaschinenschrift`* dargestellt.
- Neue oder wichtige Begriffe sind in *Kursivschrift* hervorgehoben.
- In den Listings sind alle Codezeilen nummeriert. Falls eine nicht nummerierte Codezeile erscheint, gehört diese zur unmittelbar vorhergehenden Zeile (manche Codezeilen sind so lang, dass sie die Breite einer Buchseite überschreiten). In diesem Fall sollten Sie die beiden Zeilen als eine Zeile eintippen, d.h. die Zeilen nicht trennen.

Die Begleit-CD

Die Begleit-CD zum Buch enthält:

- Sämtliche im Buch wiedergegebenen Listings.
- Einen kostenlosen Compiler mit der integrierten Entwicklungsumgebung von Borland: C++BuilderX. Dieser benötigt einen Key, den Sie auf der Borland-Webseite erhalten:

  http://www.borland.com/products/downloads/download_cbuilderx.html

  Bitte beachten Sie, dass die Software von Borland und nicht vom Verlag zur Verfügung gestellt wird. Anfragen bezüglich der Software können von Verlagsseite nicht bearbeitet werden.

  Auch wenn alle Codebeispiele mit diesem Compiler getestet wurden, ist es kein Buch, das für diesen Compiler geschrieben wurde. Die Beispiele lassen sich mit fast jedem modernen C++-Compiler verwenden. Beachten Sie aber: Nicht alle Compiler sind robust oder mit dem Standard voll kompatibel.
- Lösungen für viele Übungen, die am Ende jeder Stunde angegeben sind.

KAPITEL 1

jetzt lerne ich

Erste Schritte

In der ersten Stunde lernen Sie,

- wie Sie Ihren Compiler einrichten und verwenden,
- in welchen Schritten ein C++-Programm entwickelt wird,
- wie Sie Ihr erstes funktionsfähiges C++-Programm eingeben, kompilieren und linken.

1.1 Vorbereitungen

In C++ ist es vielleicht wichtiger als in anderen objektorientierten Sprachen wie zum Beispiel Java, dass der Programmierer das Programm entwirft, bevor er es niederschreibt. Triviale Probleme, wie etwa die in den ersten Kapiteln dieses Buches behandelten, erfordern kaum Entwurfsarbeit. Allerdings ist ein Entwurf unumgänglich bei komplexen Problemen, mit denen der professionelle Programmierer tagtäglich zu tun hat. Je gründlicher der Entwurf, desto wahrscheinlicher stellt das Programm für die vorgesehenen Aufgaben eine Lösung dar – sowohl im zeitlichen als auch im finanziellen Rahmen. Ein guter Entwurf führt auch zu einem relativ fehlerfreien und leicht zu wartenden Programm. Man schätzt, dass sich 90 Prozent der Software-Kosten aus Fehlersuche und Wartung zusammensetzen. Ein guter Entwurf hat damit bedeutenden Einfluss auf die Senkung der Gesamtkosten des Projekts.

Bei der Vorbereitung auf den Programmentwurf ist zuerst die Frage zu beantworten: »Welches Problem versuche ich zu lösen?« Jedes Programm sollte ein klares, gut formuliertes Ziel besitzen. Sogar bei den einfachsten Programmen in diesem Buch begegnen Sie diesem Ansatz.

Die zweite Frage für jeden guten Programmierer ist: »Kann man das Ziel erreichen, ohne benutzerspezifische Software schreiben zu müssen?« Ein vorhandenes Programm wiederzuverwenden und anzupassen oder konfektionierte Software zu kaufen ist oft besser als etwas Neues zu schreiben. Der Programmierer, der diese Alternativen zu bieten hat, wird niemals arbeitslos; auf der Suche nach kostengünstigen Lösungen für die aktuellen Probleme bieten sich immer neue Möglichkeiten für die Zukunft.

Selbst wenn man neue Software schreibt, lassen sich die Anstrengungen anderer nutzen, indem man auf bereits vorhandene Bibliotheken oder Klassen (Bibliotheken von Objektdefinitionen) zurückgreift. Derartige Instrumente können die Entwicklung beschleunigen, woraus sich unter anderem auch die Popularität von C++ erklärt.

developerWorks von IBM

Unter *http://www.ibm.com/developerworks/* bieten die *developerWorks* von IBM viele ausgezeichnete Beispiele, wie man Bibliotheken gemeinsam nutzen kann, auch wenn sich die *developerWorks* vorrangig auf Java konzentrieren.

Hat man das Problem schließlich erfasst und ist das Schreiben eines neuen Programms erforderlich, kann man mit dem Entwurf beginnen.

1.2 Klarheit schaffen: C++, ANSI C++, ISO C++ und Windows

C++ ist eine Programmiersprache. Windows, UNIX (wie AIX, Solaris, HP-UX und andere Dialekte), Linux und MacOS X sind Betriebssysteme. C++ werden Sie als portable Sprache kennen lernen, die unabhängig von der konkreten Maschine und dem benutzten Betriebssystem ist.

Jetzt lerne ich C++ trifft keine Annahmen über Ihr Betriebssystem. Dieses Buch behandelt *ANSI/ISO C++*. Man könnte genauso gut auch »Standard-C++« sagen – die international vereinbarte Version, die auf *jede* Plattform und *jede* Entwicklungsumgebung portierbar ist.

Was ist ANSI? Was ist ISO?

ANSI ist die Abkürzung für American National Standards Institute und ISO für International Organization for Standardization. Wie diese Namen verraten, verkünden diese Organisationen verschiedenste Standards für die Vereinigten Staaten bzw. die Welt. Die Verwendung einer Sprache, die auf veröffentlichten Standards basiert, ist in dem Sinne nützlich, dass sich jeder an die gleiche Beschreibung und die gleichen Regeln hält.

Der im Buch vorgestellte Code ist Standard-ANSI/ISO C++ und sollte auf nahezu jedem Computer laufen. Deshalb finden Sie kaum Verweise auf Fenster, Listenfelder, Grafiken usw. Diese Elemente sind vom jeweiligen Betriebssystem und der konkreten Entwicklungsumgebung abhängig.

Alle Ausgaben laufen über die Standard-Ausgabe. Damit das funktioniert, muss man gegebenenfalls den Compiler anweisen, eine Konsolenanwendung zu erzeugen. Das ist der Fall beim C++BuilderX-Compiler von Borland. Bestimmte Compiler, die für Windows, den Mac oder eine andere Fenster-Umgebung vorgesehen sind, bezeichnen das als Quick-Fenster, einfaches Fenster oder auch Konsolenfenster.

Ein *Compiler* ist die Software, mit der Sie das ganze Buch hindurch arbeiten. Er übersetzt ein Programm aus einer für den Menschen lesbaren Form in Maschinencode. Dabei entsteht eine Objektdatei, die später mit anderen Objektdateien zu einem ausführbaren Programm gebunden – gelinkt – wird. Ein *Linker* ist ein Programm, das eine ausführbare Datei aus den vom Compiler erzeugten Objektcodedateien erstellt.

Einen Compiler finden Sie auf der Begleit-CD zum Buch. Der nächste Abschnitt führt Sie durch den Einrichtungs- und Installationsvorgang für den Borland C++BuilderX-Compiler. Dieser Compiler ist für Windows-basierte Betriebssysteme konzipiert und bietet eine integrierte Entwicklungsumgebung (Integrated Development Environment, IDE), mit der Sie Ihre Programme in einer grafischen Benutzeroberfläche bearbeiten, kompilieren und debuggen sowie weitere Funktionen ausführen können. Vielleicht möchten Sie auch einen anderen Compiler einsetzen, der nicht von einer Windows-Umgebung abhängig ist, das ist aber allein eine Sache Ihrer persönlichen Entscheidung.

1.2.1 Den Compiler installieren und einrichten

Die integrierte Entwicklungsumgebung C++BuilderX ist für die gängigen Betriebssysteme auf der Begleit-CD enthalten.

Installieren Sie nun den Borland C++-Compiler und die integrierte Entwicklungsumgebung, d.h. die grafische Benutzeroberfläche, mit der Sie im weiteren Verlauf des Buches arbeiten. Für diese grafische Benutzeroberfläche finden Sie die Namen Borland Compiler, Borland C++BuilderX, C++BuilderX oder C++BuilderX IDE. Diese Namen bezeichnen alle dasselbe.

Von der CD installieren

Zur Installation des C++Builders wechseln Sie über die CD-Oberfläche in das Verzeichnis der Software und starten die für Ihr System relevante *Launcher_xxx*-Datei. Für Windows beispielsweise *launcher_windows.exe*. Beachten Sie auch die Readme-Dateien in diesem Verzeichnis.Starten Sie dann die »vollständige Installation« und führen Sie folgende Schritte aus:

1. Klicken Sie auf WEITER.
2. Wählen Sie die Option *Ich bin mit den Bedingungen des Lizenzvertrags einverstanden*.
3. Klicken Sie auf WEITER (es empfiehlt sich, die Standardeinstellungen zu übernehmen).
4. Wählen Sie den Funktionssatz *Vollständig* und klicken Sie auf WEITER.
5. Legen Sie den Speicherort für die Installation fest (als Standardeinstellung ist das Verzeichnis `C:\CBuilderX` angegeben) und klicken Sie auf WEITER.
6. Wählen Sie den Speicherort für die Symbole (auch hier können Sie die Standardeinstellungen übernehmen) und klicken Sie auf WEITER.
7. Klicken Sie abschließend auf INSTALLIEREN.
8. Nach kurzer Zeit erscheint das Fenster mit der Meldung *Installation abgeschlossen*. Klicken Sie auf FERTIG.

Einen Lizenzschlüssel erhalten Sie unter *http://www.borland.com/products/downloads/download_cbuilderx.html*, in der Tabelle »Keys Only«. Borland sendet Ihnen per E-Mail eine Datei mit einem Schlüssel. Speichern Sie diese Datei im Installationsverzeichnis des Programms (folgen Sie dabei den Anweisungen, die Sie zusammen mit der gesendeten Schlüsseldatei erhalten). Damit haben Sie die integrierte Entwicklungsumgebung von C++BuilderX installiert und brauchen sie nur noch zu konfigurieren, damit sie einfacher zu handhaben ist.

1.2.2 Den Borland C++BuilderX-Compiler konfigurieren

Öffnen Sie jetzt die C++BuilderX-Entwicklungsumgebung. Die zu wählenden Menübefehle sind in der Reihenfolge aufgeführt, in der sie nacheinander erscheinen. Klicken Sie vom Desktop aus auf START | PROGRAMME | BORLAND C++BUILDERX | C++BUILDERX. Daraufhin startet die integrierte Entwicklungsumgebung von C++BuilderX. Wir verwenden hier die Standardkonfiguration, die Sie aber jederzeit über das Menü TOOLS in der Hauptmenüleiste der Entwicklungsumgebung ändern können.

Wie Abbildung 1.1 zeigt, sind unterhalb der Menüleiste zwei Zeilen mit Symbolen zu sehen. Das ist die Symbolleiste. Wenn Sie den Mauszeiger auf ein Symbol setzen, erscheint eine kurze Beschreibung mit der Funktion der jeweiligen Schaltfläche. In der ersten Reihe heißen die ersten sieben Symbole NEU (Erstellen eines Projekts), DATEI ÖFFNEN (Öffnen eines vorhandenen Projekts), NEU ÖFFNEN (zur vorherigen Version auf der Festplatte zurückkehren), QUELLDATEI SCHLIEßEN, QUELLDATEI SPEICHERN, ALLES SPEICHERN und DRUCKEN. Das erste Symbol in der zweiten Reihe mit dem kleinen Schriftzug 10101 ist die Schaltfläche, über die Sie Ihren Code kompilieren und eine ausführbare Datei erstellen. Die Schaltfläche mit dem grünen Pfeil führt ein kompiliertes Programm aus. Vergleichen Sie diese Symbole mit den Menübefehlen in der Menüleiste.

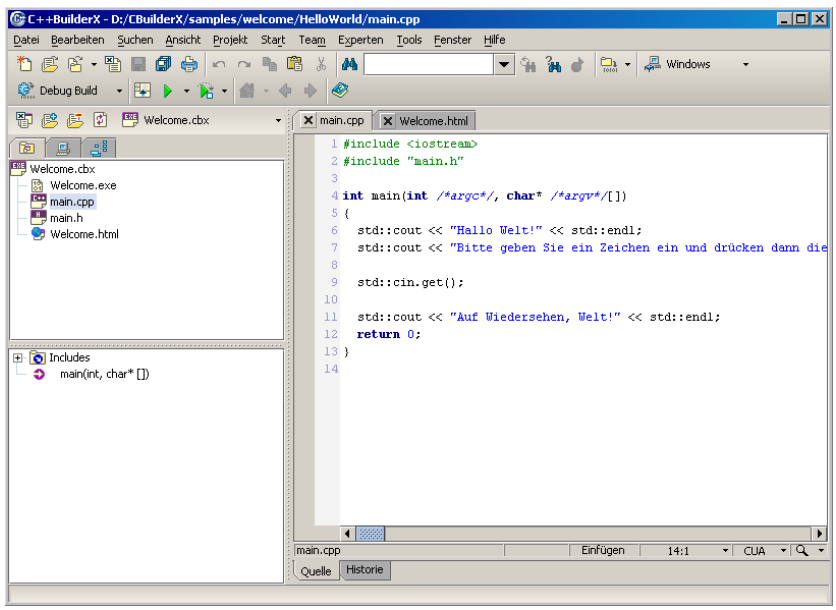


Abbildung 1.1: Die integrierte Entwicklungsumgebung von C++BuilderX

Bei mehreren Symbolen taucht der Begriff *Projekt* auf. In einem Projekt fassen Sie mehrere Dateien zusammen. Die C++BuilderX-Projektdateien haben die Erweiterung `.cbx`. Die meisten Projekte in diesem Buch bestehen nur aus einer einzigen Datei. Der praktische Wert, ein Projekt zu erstellen, zeigt sich erst, wenn Sie Dutzende oder sogar Hunderte von Quelldateien haben, aus denen ein sehr komplexes Programm besteht.

Standardmäßig ist die integrierte Entwicklungsumgebung von C++BuilderX so konfiguriert, dass Sie Ihre Programme debuggen (Fehler suchen und beseitigen) können. Auf diese Funktion geht dieses Buch später näher ein.

Die Borland-IDE hat einen integrierten Editor. Klicken Sie auf das Symbol NEU und wählen Sie die Registerkarte QUELLDATEI. Hier können Sie auf einzelne Dateien zugreifen. Um mit mehreren Dateien und Projekten zu arbeiten, klicken Sie erneut auf das Symbol NEU und wählen *Neue Konsole* auf der Registerkarte PROJEKT. Klicken Sie auf OK um fortzufahren. Haben Sie einen Assistenten gewählt, erstellt er das Gerüst für Ihr Hauptprogramm. Wenn Sie diese Option deaktiviert lassen, erhalten Sie ein leeres Quelltextfenster, in das Sie Ihren Code eintippen können. Abbildung 1.2 zeigt ein Programm, das unter dem Debugger ausgeführt wird. Hier wurde ein Haltepunkt auf Zeile 7 gesetzt und der Code bis zu diesem Punkt ausgeführt.

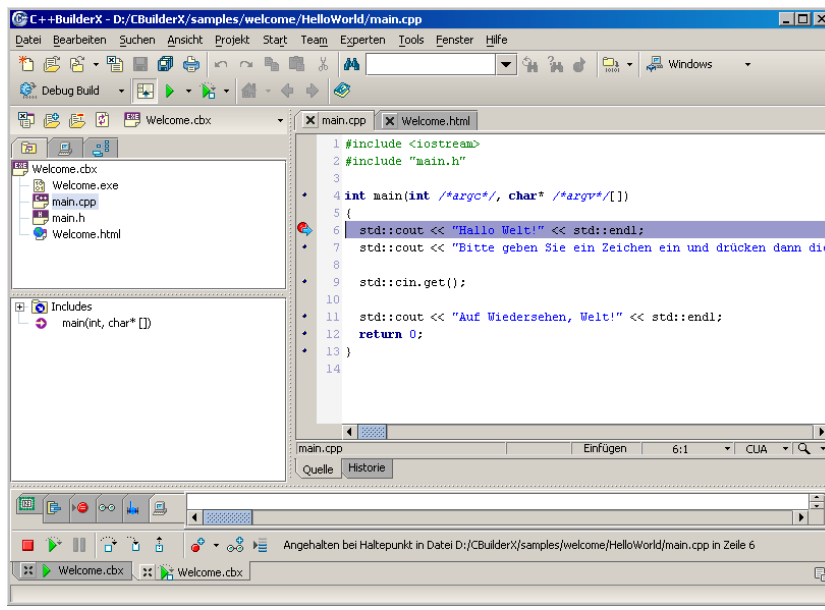


Abbildung 1.2: Den Debugger in Borland C++BuilderX verwenden

Speichern Sie das Projekt und alle dazugehörenden Dateien in einem separaten Verzeichnis. Die Projektdatei hat die Erweiterung `.cbx`, die C++-Quellda-

teien tragen die Erweiterung `.cpp`. Der Editor fügt die Erweiterung automatisch an den Quelldateinamen an, wenn Sie die Datei speichern. Um die Datei zu speichern, an der Sie gerade arbeiten, klicken Sie auf das Symbol *Dateiname* SPEICHERN.

1.2.3 Kompilieren mit dem Borland C++BuilderX-Compiler

Um ein Projekt oder eine einzelne Quelldatei zu kompilieren und zu linken, klicken Sie auf das BUILD-Symbol. Bei manchen Compilern sind zwei getrennte Schritte erforderlich: zuerst kompilieren und dann linken. Die meisten integrierten Entwicklungsumgebungen – einschließlich dieser – führen allerdings beide Schritte auf einmal aus, wenn Sie die Kompilieren-Funktion aufrufen. Um ein Programm aus der integrierten Entwicklungsumgebung auszuführen, klicken Sie auf das Symbol PROJEKT AUSFÜHREN. Ein Programm lässt sich über das Symbol FEHLERSUCHE IN PROJEKT debuggen.

Eine braune Markierung zeigt an, welche Zeile als Nächstes ausgeführt wird. Der rote Kreis stellt einen Haltepunkt dar. An einem Haltepunkt stoppt Ihr Programm vorübergehend. Es empfiehlt sich, einen Haltepunkt auf die erste ausführbare Zeile zu setzen, wenn Sie mit der Fehlersuche in einem Programm beginnen. Um einen Haltepunkt zu setzen, klicken Sie mit der linken Maustaste auf die betreffende Zeile. Die Schaltfläche EINZELANWEISUNG führt eine Zeile aus und hält das Programm auf der nächsten Zeile an. Die Schaltfläche PROGRAMM FORTSETZEN führt das Programm weiter aus, bis es beendet ist oder einen weiteren Haltepunkt erreicht. Die anderen Schaltflächen erlauben es, verschiedene Aspekte des Programms zu untersuchen. Die entsprechenden Symbole finden Sie im unteren Teil des Fensters (am unteren Rand von Abbildung 1.2). Dieser Compiler kompiliert oder erstellt die ausführbare Datei automatisch, falls Sie diesen Schritt nicht explizit aufgerufen haben, bevor Sie versuchen, den Code auszuführen oder zu debuggen.

Wenn Sie im Menü HILFE auf INFO ÜBER C++BUILDERX klicken, erscheint die Versionsinformation von Borland C++BuilderX. Über HILFE | HILFETHEMEN gelangen Sie zu umfassenden Referenzinformationen zu den C- und C++-Bibliotheken. Nehmen Sie sich etwas Zeit, um diese wertvolle Ressource zu erkunden.

Kontextabhängige Hilfe

Ein praktisches Feature dieser integrierten Entwicklungsumgebung ist die kontextabhängige Hilfe. Klicken Sie auf ein Schlüsselwort und drücken Sie F1. Daraufhin erscheint ein weiteres Fenster mit Informationen zum jeweiligen Befehl.

1.2.4 Compiler und Editor

Bei vielen Compilern ist ein integrierter Texteditor im Lieferumfang enthalten. Um die C++-Quelldateien zu erstellen, eignen sich der integrierte Editor, ein kommerzieller Text-Editor oder eine Textverarbeitung, die reine Textdateien erzeugen kann. Wie bereits erwähnt, verfügt der Borland-Compiler über einen integrierten Editor. Mit welchem Editor Sie Ihr Programm auch erstellen, er muss reine Textdateien ohne eingebettete Steuerbefehle liefern. Beispiele für »sichere« Editoren sind der zu Windows gehörende Editor (Notepad), der über den DOS-Befehl `Edit` aufrufbare Editor oder auch Brief, Epsilon, EMACS und vi. Kommerzielle Textverarbeitungen wie WordPerfect, Word und Dutzende andere bieten ebenfalls die Möglichkeit, das Dokument als einfache Textdatei zu speichern.

Seien Sie auf der Hut!

Achten Sie darauf, dass Sie keine Formatierungen speichern, wenn Sie Ihre Programme mit einer Textverarbeitung erstellen – die Datei können Sie dann zwar lesen, aber die Formatierungscodes bringen den Compiler durcheinander.

Die Dateien, die Sie mit Ihrem Editor erstellen, bezeichnet man als Quelldateien. Normalerweise erhalten diese Dateien unter C++ die Dateierweiterung `.cpp`, `.cxx`, `.cp` oder `.c`. In diesem Buch sind alle Quellcodedateien einheitlich mit der Erweiterung `.cpp` versehen, da das der Standard des Borland-Compilers ist.

Erweiterung der Quelldateien

Bei den meisten C++-Compilern hat die Erweiterung der Quelldateien keine besondere Bedeutung. Gibt man allerdings keine Erweiterung explizit an, verwenden viele Compiler per Vorgabe die Erweiterung `.cpp`.

Was Sie tun sollten

Erstellen Sie den Quellcode mit einem einfachen Texteditor, oder arbeiten Sie mit dem Editor, der zum Compiler gehört.

Nutzen Sie die Hilfe, die der sprachabhängige Editor (Language Sensitive Editor, LSE) oder die integrierte Entwicklungsumgebung bereitstellt.

Speichern Sie Ihre Dateien mit den Erweiterungen `.c`, `.cp` oder `.cpp`. (Der Borland-Compiler verwendet standardmäßig `.cpp`.)

Machen Sie sich anhand der Dokumentation mit den Eigenheiten Ihres Compilers und Linkers vertraut, damit Sie wissen, wie Programme zu kompilieren und zu linken sind.

Verwenden Sie keine Textverarbeitung, die spezielle Formatierungszeichen speichert. Sollten Sie doch mit einer Textverarbeitung arbeiten, speichern Sie die Datei als ASCII-Text.

1.3 Quellcode kompilieren und linken

Obwohl der Quellcode in Ihrer Datei etwas kryptisch aussieht und jeder, der sich mit C++ nicht auskennt, kaum versteht, für was das Ganze gut ist, haben wir es trotzdem mit einer für den Menschen verständlichen Form zu tun. Die Quellcodedatei ist kein Programm und lässt sich auch nicht wie ein Programm ausführen.

Kompilieren von der Befehlszeile

Für die Überführung des Quellcodes in ein Programm verwendet man einen Compiler. Wie man den Compiler aufruft und ihm den Standort der Quelldatei angibt, ist vom konkreten Compiler abhängig. Sehen Sie dazu in der entsprechenden Dokumentation nach.

Um den Quellcode von der Befehlszeile des Betriebssystems (statt von einer integrierten Entwicklungsumgebung) aus zu kompilieren, sind folgende Befehle – abhängig von Compiler und Betriebssystem – einzugeben:

- Beim Borland- Compiler: `bcc32 <Dateiname> -o <Name der ausführbaren Datei>`
- Bei GNU-Binärdateien: `gxx <Dateiname> -o <Name der ausführbaren Datei>`
- Beim Cygwin-Compiler: `g++ <Dateiname> -o <Name der ausführbaren Datei>`
- Bei den meisten Linux-Versionen: `g++ <Dateiname> -o <Name der ausführbaren Datei>`
- Bei UNIX-Versionen: `c++ <Dateiname> -o <Name der ausführbaren Datei>`

1.3.1 Kompilieren in einer integrierten Entwicklungsumgebung

Die meisten modernen Compiler bieten eine so genannte *integrierte Entwicklungsumgebung*. Hier wählt man normalerweise ERSTELLEN oder KOMPILIEREN aus einem Menü oder es steht eine Funktionstaste für das Erstellen der Anwendung zur Verfügung. In der integrierten Entwicklungsumgebung von C++BuilderX ist für diesen Zweck die Schaltfläche BUILD vorgesehen.

1.3.2 Ein Programm linken

Durch das Kompilieren des Quellcodes entsteht eine so genannte *Objektdatei*. Allerdings handelt es sich auch hier noch nicht um ein ausführbares Programm. Die Objektdatei überführt man mit einem Linker in ein ausführbares Programm.

C++-Programme werden in der Regel durch das Verknüpfen (Linken) einer oder mehrerer Objektdateien mit einer oder mehreren Bibliotheken erzeugt. Unter einer *Bibliothek* versteht man eine Sammlung verknüpfungsfähiger Dateien, die Sie selbst erzeugt haben, die der Compiler bereitstellt oder die Sie separat erworben haben. Zu allen C++-Compilern gehört eine Bibliothek nützlicher Funktionen (oder Prozeduren) und Klassen, die Sie in Ihr Programm aufnehmen können. Eine *Funktion* ist ein Codeblock, der eine bestimmte Aufgabe realisiert, beispielsweise das Addieren zweier Zahlen oder die Ausgabe auf den Bildschirm. Unter einer *Klasse* ist die Definition eines neuen Typs zu verstehen. Klassen werden als Daten mit den zugehörigen Funktionen implementiert. Auf Klassen gehen wir im weiteren Verlauf des Buches noch detailliert ein.

Eine ausführbare Datei erzeugt man normalerweise in folgenden Schritten:

1. Erstellen einer Quellcodedatei mit der Erweiterung `.cpp`.
2. Kompilieren der Quellcodedatei in eine Objektdatei.
3. Linken der Objektdatei mit allen erforderlichen Bibliotheken zu einem ausführbaren Programm.

Die Schaltfläche BUILD der integrierten Borland-Entwicklungsumgebung kompiliert und linkt das Programm automatisch in einem Schritt. Module können Sie einzeln kompilieren, indem Sie mit der rechten Maustaste auf den jeweiligen Namen im Projektfenster klicken, auf BUILD-TOOLS im Kontextmenü zeigen und BCC32 (den eigentlichen C++-Compiler) im Untermenü wählen. Um ein Programm manuell zu linken, klicken Sie mit der rechten Maustaste im Projektfenster auf den Projektnamen, zeigen auf BUILD-TOOLS im Kontextmenü und wählen ILINK32 (den mit der Borland-IDE gelieferten Linker) im Untermenü.

1.4 Der Entwicklungszyklus

Wenn jedes Programm sofort beim ersten Ausprobieren funktionieren würde, hätten wir es mit folgendem vollständigen Entwicklungszyklus zu tun: Schreiben des Programms, Kompilieren des Quellcodes, Linken des Programms und Ausführen des Programms. Leider enthält fast jedes Programm, wie trivial es auch sein mag, irgendwelche Fehler – so genannte *Bugs*. Einige Bugs verhindern bereits das Kompilieren, bei manchen Fehlern kann man das Programm nicht linken und einige Fehler zeigen sich erst bei der Ausführung des Programms.

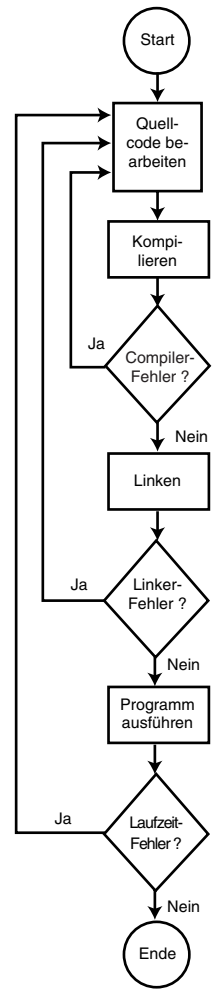


Abbildung 1.3: Die Schritte im Entwicklungsprozess eines C++-Programms

Ein »Syntaxfehler« wird in der Kompilierphase entdeckt: Es handelt sich um einen Fehler in der Verwendung der Sprache. Des Weiteren können Warnungen erscheinen, die einen Denkfehler nahe legen, wobei aber der Compiler in der Lage ist, eine binäre Objektdatei zu erstellen. Zu den Linker-Fehlern gehören in der Regel fehlende Bibliotheken oder falsch geschriebene Funktionsnamen. Fehler, die sich bei der Ausführung des Programms zeigen, gehören zur Kategorie der logischen Fehler.

Welchen Fehler Sie auch finden, er ist zu beseitigen. Und dazu gehört die Bearbeitung des Quellcodes, das erneute Kompilieren und Linken und schließlich ein neuer Start des Programms. Abbildung 1.3 zeigt diesen Zyklus mit einer Darstellung der einzelnen Schritte. Hier haben wir es mit einem der selten vorkommenden Programme zu tun, die sich bereits beim ersten Versuch kompilieren, linken und fehlerfrei ausführen lassen.

1.5 HELLO.CPP – Ihr erstes C++-Programm

Es ist schon Tradition, dass ein Programmierbuch für C-ähnliche Sprachen mit einem Programm beginnt, das die Worte `Hello World` auf den Bildschirm bringt oder eine Variante dieser Anweisung realisiert. Auch wir bleiben dieser Tradition treu. Alles hat mit dem ersten C-Programmierbuch angefangen, das die Autoren der Sprache C – Kernighan und Ritchie – geschrieben haben.

Ein neues Programm beginnen

Um ein neues Projekt anzulegen, klicken Sie in der C++BuilderX-IDE auf NEU. Für die Beispiele in diesem Buch wählen Sie *Neue Konsole*. Nachdem das Projekt erstellt ist, gelangen Sie zum Editor.

Tippen Sie das erste Programm direkt in Ihren Editor – genau wie dargestellt – ein. Nachdem Sie den Quelltext kritisch durchgesehen haben, speichern Sie die Datei mit dem Namen `hello.cpp`, kompilieren sie, linken sie (falls das der Compiler nicht ohnehin automatisch erledigt) und führen sie aus. Auf dem Bildschirm erscheinen die Worte `Hello World`. Kümmern Sie sich momentan noch nicht darum, wie das Ganze funktioniert, das Programm soll Sie nur mit dem Entwicklungszyklus vertraut machen. Die einzelnen Aspekte dieses Programms sind Gegenstand der nächsten Lektionen.

Ein paar Worte zu Listings

Das folgende Listing enthält an der linken Seite Zeilennummern. Diese dienen lediglich der Bezugnahme in diesem Buch und sind nicht als Teil des Quelltextes im Editor einzugeben. Beispielsweise geben Sie die Zeile 1 von Listing 1.1 in der folgenden Form ein:

```
#include <iostream.h>
```

Die Zeilennummern sind hier nur für die Analyse des Codes angegeben. Ihr Editor oder die integrierte Entwicklungsumgebung können Zeilennummern wie der Borland-Compiler anzeigen, dennoch gehören diese Nummern in keinem Fall zum Code.

Alle Listings in diesem Buch beginnen mit der Zeilennummer 1, wie es auch in der integrierten Entwicklung C++BuilderX der Fall ist. Falls Sie die Zeilennummernanzeige unterdrücken möchten, wählen Sie im Menü TOOLS den Befehl EDITOR-OPTIONEN und deaktivieren das Kontrollkästchen neben dem Eintrag *Zeilennummern anzeigen*.

Alle Listings in diesem Buch finden Sie auf der Begleit-CD zum Buch mit Namen, die in der Listingunterschrift angegeben sind und Ihnen etwas Tipparbeit sparen sollen.

Listing 1.1: hello.cpp – Das Programm »Hello World«

```
1: #include <iostream>
2:
3: int main()
4: {
5:     std::cout << "Hello World!\n";
6:     return 0;
7: }
```

Vergewissern Sie sich, dass Sie alles exakt wie dargestellt eingegeben haben. Achten Sie insbesondere auf die Satzzeichen. Das Symbol << in Zeile 5 ist ein Umleitungssymbol, das man auf deutschen Tastaturen rechts neben der linken ⇧-Taste findet. Zeile 5 endet mit einem Semikolon, das auf keinen Fall fehlen darf!

Namen und Leerzeichen

Der Befehl `std::cout` besteht aus zwei Teilen: `std` und `cout`. `cout` schreibt die Ausgabe auf die Konsole. Der Befehl ist Teil des Standardpakets. Da es aber so viele Programmierer gibt, die Funktionen und Objekte für C++ schreiben, kommt es häufig vor, dass zwei Programmierer den gleichen Namen für verschiedene Dinge verwenden. Um zwischen den beiden Versionen zu unterscheiden, müssen wir den gewünschten *Namensbereich* oder die Version spezifizieren. In diesem Fall möchten wir die `std`-Version verwenden und schreiben sie deshalb vor den Objektnamen `cout`.

Prüfen Sie auch, ob die Compiler-Direktiven entsprechend eingestellt sind. Die meisten Compiler einschließlich des Borland-Compilers linken automatisch. Trotzdem sollten Sie in der Dokumentation des jeweiligen Programms nachsehen. Wenn Sie Fehler erhalten, überprüfen Sie zunächst sorgfältig den eingegebenen Code und suchen nach Abweichungen von Listing 1.1.

Falls Sie einen anderen als den Borland-Compiler verwenden, erhalten Sie gegebenenfalls auch andere Fehler oder Fehlermeldungen. Erscheint beispielsweise für Zeile 1 ein Fehler wie `Include-Datei kann nicht geoeffnet werden: 'iostream.h'...`, sollten Sie sich in der Dokumentation zu Ihrem Compiler über die Einstellungen für den Include-Pfad und die Umgebungsvariablen informieren. Wenn Sie einen Fehler erhalten, dass es keinen Prototyp für `main` gibt, fügen Sie die Zeile `int main();` unmittelbar vor Zeile 3 ein. Diese Zeile müssen Sie jedem Programm in diesem Buch vor Beginn der Funktion `main()` hinzufügen. Die meisten Compiler, Borland eingeschlossen, erfordern das nicht, aber andere bestehen darauf.

Das fertige Programm sieht dann folgendermaßen aus:

```
1: #include <iostream>
2:
3: int main();
4: int main()
5: {
6:    std::cout << "Hello World!\n";
7:    return 0;
8: }
```

Kompilieren Sie nun `hello.cpp`, um die Datei `hello.exe` zu erstellen, und führen Sie die Datei aus. Auf dem Bildschirm sollte der Text

```
Hello World!
```

erscheinen. Sollte das so sein, Gratulation! Sie haben gerade Ihr erstes C++-Programm eingegeben, kompiliert und ausgeführt. Es ist vielleicht nicht gerade berauschend, aber nahezu jeder professionelle C++-Programmierer hat genau mit diesem Programm begonnen.

1.6 Fehler zur Kompilierzeit

Fehler zur Kompilierzeit können verschiedene Ursachen haben. Gewöhnlich sind sie das Ergebnis eines Schreibfehlers oder einer anderen Nachlässigkeit. Gute Compiler weisen nicht nur darauf hin, was Sie falsch gemacht haben, sie zeigen genau die Stelle im Code an, wo der Fehler aufgetreten ist. Die besten Compiler schlagen sogar eine Lösung vor!

Einen Compilerfehler können Sie sich in der Praxis ansehen, indem Sie absichtlich einen Fehler in das Programm einbauen. Nachdem Sie `hello.cpp` erst einmal richtig zum Laufen gebracht haben, bearbeiten Sie nun die Quelldatei (nach Listing 1.1) und entfernen die schließende geschweifte Klammer in Zeile 7. Das Programm entspricht nun Listing 1.2.

Listing 1.2: hellobad.cpp – Demonstration eines Compilerfehlers

```
1: #include <iostream>
2:
3: int main()
4: {
5:     std::cout << "Hello World!\n";
6:     return 0;
```

Kompilieren Sie Ihr Programm erneut. Es sollte folgende Fehlermeldung erscheinen:

```
"hellobad.cpp": E2134 Abschließendes } fehlt in Funktion main() in
Zeile 6
```

Dieser Meldung können Sie entnehmen, dass der Compiler einen Fehler am Ende des Quellcodes der Datei `hellobad.cpp` erkannt hat. (Zugegebenermaßen sieht das Ganze etwas kryptisch aus.) Beachten Sie, dass die Fehlermeldung auf Zeile 6 verweist. Der Compiler ist nicht immer in der Lage, den Fehler genau zu lokalisieren, und verweist gegebenenfalls auf eine Zeile, die gar nicht mehr zur Datei gehört. Manchmal bezieht sich die Fehlermeldung auch auf die unmittelbare Nachbarschaft des Problems. Könnte ein Compiler jedes Problem perfekt identifizieren, könnte er den Code auch selbst von Fehlern bereinigen.

1.7 Fragen und Antworten

F: Worin besteht der Unterschied zwischen einem Texteditor und einer Textverarbeitung?

A: Ein Texteditor erzeugt Dateien, die aus reinem Text bestehen und keinerlei Formatierungsbefehle oder andere spezielle Symbole, wie sie für eine Textverarbeitung erforderlich sind, enthalten. Bei reinen Textdateien gibt es weder automatische Zeilenumbrüche noch Fettdruck, kursive Zeichen usw. Textverarbeitungen können in der Regel Textdateien erzeugen, man muss die Datei aber explizit als Dateityp »Nur Text« speichern.

F: Mein Compiler verfügt über einen integrierten Editor. Ist das das geeignete Werkzeug?

A: Fast alle Compiler sind in der Lage, den von beliebigen Texteditoren erzeugten Code zu kompilieren. Der Vorteil des integrierten Editors besteht allerdings darin, dass man gegebenenfalls schnell zwischen den Entwicklungsschritten Bearbeiten und Kompilieren hin- und herschalten kann. Zu modernen Compilern gehört eine integrierte Entwicklungsumgebung: Hier kann der Programmierer sowohl auf Hilfedateien zugreifen als auch den Code bearbeiten und kompilieren – und das alles von einer zentralen Stelle aus. Außerdem lassen sich Compiler- und Linker-Fehler beseitigen, ohne dass man die Umgebung verlassen muss. Es gibt aber auch Gründe, einen anderen Editor zu verwenden: Manche Editoren sind funktionell reicher ausgestattet oder Sie bevorzugen einen Editor, mit dem Sie vertraut sind und der Ihnen ein zügiges Arbeiten erlaubt.

F: Kann ich Warnungen des Compilers ignorieren?

A: Auf keinen Fall. Machen Sie es sich zur Angewohnheit, Warnungen als Fehler zu behandeln. C++ verwendet den Compiler, um Sie vor etwas zu warnen, das Sie möglicherweise gar nicht vorhaben. Unternehmen Sie die erforderlichen Schritte, damit diese Warnungen nicht wieder auftreten. Eine Fehlermeldung besagt, dass der Compiler Ihren Quelltext nicht in Maschinensprache übersetzen kann. Bei einer Warnung lässt sich zwar der Quelltext konvertieren, allerdings ist nicht sicher, ob dies in der von Ihnen beabsichtigten Art und Weise geschieht.

F: Was versteht man unter der Kompilierzeit?

A: Die Kompilierzeit ist die Phase im Entwicklungszyklus, zu der man den Compiler ausführt. Weitere Phasen sind die Linkzeit (wenn man mit dem Linker den Objektcode verknüpft) oder die Laufzeit (wenn man das Programm ausführt).

1.8 Workshop

Nachdem Sie Ihr erstes Programm eingegeben, kompiliert, gelinkt und ausgeführt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen ausführen, um Ihr Wissen zum Compiler zu festigen.

1.8.1 Kontrollfragen

1. Warum ist ein Projekt erforderlich?
2. Was versteht man unter Debuggen?
3. Weshalb verwendet man einen Compiler, der entsprechend nationaler oder internationaler Standards kompiliert?
4. Mit welchen Tools können Sie Ihren Quellcode bearbeiten?

1.8.2 Übungen

Die Übungen in dieser Lektion sind zur reinen Selbstkontrolle gedacht. Es gibt für sie keine Standardantworten oder -lösungen. Später im Buch finden Sie auch Übungen, für die Lösungen auf der Begleit-CD enthalten sind.

1. Machen Sie sich mit den Schaltflächen und Menübefehlen des auf der Begleit-CD enthaltenen Borland-Compilers und dessen integrierter Entwicklungsumgebung vertraut. Was bewirken sie? Erleichtern sie Ihre Arbeit oder wird alles komplizierter?
2. Kompilieren, linken und starten Sie die Codebeispiele, die zum Lieferumfang des Borland-Compilers gehören. Sehen Sie sich den Code an und versuchen Sie zu erkennen, wie er die Ergebnisse produziert und warum.
3. Arbeiten Sie mit der Befehlszeilenversion des Borland-Compilers (oder eines anderen Compilers). Fällt es Ihnen leichter, Programme nach dem herkömmlichen Zyklus *Bearbeiten / Kompilieren und Linken / Ausführen* mit der Befehlszeilenversion (gegebenenfalls mit verschiedenen Tools) zu erstellen oder kommen Sie mit der integrierten Entwicklung besser zurecht?

1.8.3 Antworten zu den Kontrollfragen

1. Es teilt dem Compiler alle Komponenten mit, die für das Erstellen der ausführbaren Datei erforderlich sind. Beispielsweise gibt es Anwendungen, die eine riesige Anzahl unterschiedlicher Dateien benötigen; das Projekt organisiert sie alle.

2. Debuggen bezeichnet das Suchen und Beseitigen von Fehlern in einem Programm. Ein Bug ist ein logischer Fehler, den der Compiler nicht entdecken kann, weil er die Absichten des Programmierers nicht kennt.

3. Einfach gesagt: Weil man die Standardregeln befolgen will. Der Compiler soll sich genau wie erwartet verhalten – so wie es dieses Buch beschreibt. Es gibt doch auch nationale und internationale Vorschriften für die Benzinpistolen an Tankstellen. Ohne Standards würde jeder Hersteller seine eigenen Vorrichtungen entwickeln und das Betanken eines Fahrzeugs könnte sich schwierig gestalten.

4. Es eignet sich jedes Tool, das reinen ASCII-Text speichert. Beispielsweise können Sie die Editoren der Betriebssysteme (wie `edit` von DOS, `vi` und EMCACS von Unix/Linux oder Notepad von Windows) verwenden, einen speziellen Editor kaufen (beispielsweise Brief), in der integrierten Entwicklungsumgebung arbeiten oder sogar auf eine Textverarbeitung zurückgreifen. Wesentlich ist nur, dass Sie in der Datei keine Formatierungsbefehle speichern.

KAPITEL 2

jetzt lerne ich

Die Teile eines C++-Programms

In dieser Stunde lernen Sie,

- warum C++ der bevorzugte Standard in der Software-Entwicklung ist,
- aus welchen Teilen ein C++-Programm besteht,
- wie die Teile zusammenarbeiten,
- was eine Funktion ist und was sie bewirkt.

2.1 Warum C++ die richtige Wahl ist

C++ ist die bevorzugte Entwicklungssprache für die Mehrheit der professionellen Programmierer, da diese Sprache schnelle, kleine Programme liefert, die in einer robusten und portablen Umgebung entwickelt wurden. Die heutigen Werkzeuge von C++ machen die Entwicklung von komplexen und leistungsfähigen kommerziellen Anwendungen fast zum Kinderspiel. Damit man aber den größtmöglichen Nutzen aus C++ ziehen kann, muss man sich etwas tiefer gehend mit dieser leistungsfähigen Sprache beschäftigen.

C++ ist eine relativ neue Sprache. Die Programmierung im heutigen Sinne ist erst rund 60 Jahre alt. In dieser Zeit haben die Computersprachen einen drastischen Wandel durchgemacht. C++ betrachtet man dabei als evolutionäre Verbesserung von C. Die Sprache C selbst hat schon fast 30 Jahre auf dem Buckel.

In der Gründerzeit arbeiteten die Programmierer mit den elementarsten Computer-Anweisungen: *Maschinensprache*. Diese Anweisungen stellte man durch lange Folgen von Einsen und Nullen dar. Mit Einführung der Assembler ließen sich die Maschinenanweisungen durch besser verständliche Mnemonics wie `ADD` und `MOV` abbilden.

Im Laufe der Zeit entstanden höhere Programmiersprachen wie BASIC und COBOL. Diese Sprachen lehnen sich an natürlichen Sprachen (in der Regel Englisch) an. So lassen sich »Sätze« wie zum Beispiel `Let I = 100` bilden. Derartige Anweisungen übersetzen Interpreter und Compiler in die Maschinensprache. Ein *Interpreter* (wie er zum Beispiel für BASIC typisch ist) liest den Quelltext eines Programms, übersetzt dabei die Anweisungen – oder den Code – und wandelt sie direkt in Aktionen um.

Compiler übersetzen den Code in einen so genannten *Objektcode*. Den ersten Schritt dieser Umwandlung bezeichnet man als *Kompilieren*. Ein *Compiler* erzeugt eine Objektdatei. Der zweite Schritt ist das *Linken*. Ein *Linker* überführt die Objektdatei in ein ausführbares Programm. Ein ausführbares Programm »läuft« auf dem jeweiligen Betriebssystem.

Der Begriff *Objektcode* hat nichts mit dem Konzept der Objekte in der objektorientierten Programmierung zu tun (mehr dazu später).

Da Interpreter den Code genauso lesen, wie er niedergeschrieben wurde, und ihn unmittelbar ausführen, stellen sie ein leicht anzuwendendes Arbeitsmittel für den Programmierer dar. Bei Compilern sind zusätzliche Schritte zum Kompilieren und Linken des Codes erforderlich, was etwas umständlicher ist. Andererseits produzieren Compiler ein Programm, das im Vergleich zur interpretierten Ausführung wesentlich schneller läuft.

Über viele Jahre hinweg bestand das hauptsächliche Ziel der Programmierer darin, kurze und schnelle Codestücke zu schreiben. Das Programm musste klein sein, da Hauptspeicher teuer war, und es musste schnell sein, da Rechenzeit ebenfalls einen nicht unerheblichen Kostenfaktor darstellte. Computer wurden kleiner, billiger und schneller, die Speicherpreise sind gefallen, und somit haben sich die Prioritäten geändert. Heutzutage überwiegen die Kosten für einen Programmierer bei weitem die Kosten der meisten geschäftlich genutzten Computer. Gut geschriebener und leicht zu wartender Code steht nun an erster Stelle. Leicht zu warten bedeutet dabei, dass sich das Programm bei ändernden Anforderungen ohne großen Kostenaufwand erweitern und verbessern lässt.

2.1.1 Prozedurale, strukturierte und objektorientierte Programmierung

In der *prozeduralen Programmierung* stellt man sich die Programme als Folge auszuführender Anweisungen auf einem Satz von Daten vor. Mithilfe der *strukturierten Programmierung* bringt man Systematik in die Organisation dieser »Prozeduren« und die Verwaltung großer Datenmengen.

Der Grundgedanke der strukturierten Programmierung lässt sich auf die Formel »Teile und herrsche« bringen. Jede Aufgabe, deren Beschreibung zu komplex ist, wird in eine Menge kleinerer Teilaufgaben zerlegt. Das setzt man so lange fort, bis die Teilaufgaben eigenständig genug sind und sich leicht überblicken lassen.

Zum Beispiel ist die Berechnung des Durchschnittsgehalts aller Beschäftigten einer Firma eine ziemlich komplexe Aufgabe. Man kann diese allerdings in die folgenden Teilaufgaben zerlegen:

1. Verdienst jeder Person ermitteln.
2. Anzahl der Beschäftigten ermitteln.
3. Summe aller Gehälter bilden.
4. Dividieren der ermittelten Summe durch die Anzahl der Beschäftigten.

Der dritte Punkt lässt sich weiter in folgende Teilaufgaben gliedern:

1. Den Datensatz eines Beschäftigten holen.
2. Auf das Gehalt zugreifen.
3. Das Gehalt zur bisherigen Gesamtsumme addieren.
4. Den nächsten Datensatz eines Beschäftigten holen.

Wiederum kann man das Holen eines Beschäftigtendatensatzes in folgende Aufgaben gliedern:

1. Datei der Beschäftigten öffnen.
2. Zum entsprechenden Datensatz gehen.
3. Daten von der Festplatte lesen.

Die strukturierte Programmierung ist nach wie vor ein enorm erfolgreicher Lösungsansatz bei komplexen Aufgabenstellungen.

Trotzdem gibt es noch Probleme. Die Trennung der Daten von den Prozeduren, die die Daten manipulieren, ist mit zunehmenden Datenmengen schwieriger zu fassen und zu verwalten. Je mehr man mit den Daten anstellen will, desto unübersichtlicher gestaltet sich die Trennung.

Prozedurale Programmierer »erfinden das Rad immer wieder neu«, das heißt sie finden regelmäßig neue Lösungen für alte Probleme – das Gegenteil der *Wiederverwendbarkeit*. Der Gedanke hinter der Wiederverwendbarkeit ist der Aufbau von Komponenten mit bekannten Eigenschaften. Diese Komponenten fügt man dann bei Bedarf in ein Programm ein. Dieses Modell entspricht der Hardware-Welt – wenn ein Techniker einen neuen Transistor benötigt, erfindet er ihn in der Regel nicht neu, sondern sucht sich aus seinen Beständen den passenden heraus und modifiziert ihn vielleicht. Bis zur objektorientierten Programmierung gab es für den Software-Techniker keine vergleichbare Möglichkeit.

Das Wesen der *objektorientierten Programmierung* besteht in der Behandlung der Daten und der Prozeduren, die auf diesen Daten arbeiten, als geschlossenes »Objekt« – eine selbstständige Einheit mit einer Identität und eigenen Charakteristika.

2.1.2 C++ und objektorientierte Programmierung

C++ unterstützt voll die objektorientierte Programmierung (OOP), einschließlich der drei Säulen der objektorientierten Entwicklung: Kapselung, Vererbung und Polymorphie.

Kapselung

Wenn der Elektroniker ein neues Gerät entwickelt, verschaltet er einzelne Bauelemente – etwa Widerstände, Kondensatoren und Transistoren. Ein Transistor hat bestimmte Eigenschaften und kann bestimmte Verhaltensweisen realisieren. Man kann ihn ohne Kenntnis seiner eigentlichen Arbeitsweise einsetzen, man muss nur wissen, was er *bewirkt*.

Der Transistor muss zu diesem Zweck eigenständig sein. Er muss eine genau abgegrenzte Aufgabe haben und diese vollständig erfüllen. Die Realisierung einer in sich abgeschlossenen Aufgabe bezeichnet man als *Kapselung*.

Alle Eigenschaften des Transistors sind im Transistorobjekt verkapselt und nicht über die elektronische Schaltung verteilt. Einen Transistor kann man effektiv einsetzen, ohne mit seinen internen Abläufen vertraut zu sein.

C++ unterstützt die Eigenschaften der Kapselung über benutzerdefinierte Typen, die so genannten Klassen. Eine einmal definierte Klasse agiert als vollständig abgekapselte Einheit und wird als Ganzheit verwendet. Die eigentliche innere Funktionsweise der Klasse sollte nicht sichtbar sein. Die Benutzer einer gut konzipierten Klasse müssen nicht wissen, wie die Klasse funktioniert, sie müssen nur wissen, wie man sie verwendet. In Lektion 7 erfahren Sie, wie man Klassen erzeugt.

Vererbung und Wiederverwendbarkeit

In den späten 80ern arbeitete ich an einem Gerät für das Home-Banking für die Citibank. Das Ganze sollte nicht von Null beginnen, sondern auf Vorhandenem aufbauend schnell auf den Markt gelangen. Daher begannen wir mit dem Telefon und »erweiterten« es. Heraus kam ein Telefon, das gegenüber einem normalen Telefon lediglich ein paar neue Merkmale aufwies. Damit konnten wir alle Funktionen des guten alten Telefons übernehmen und seinen Einsatzbereich mit neuen Fähigkeiten erweitern.

C++ unterstützt den Grundgedanken der Wiederverwendbarkeit durch *Vererbung*. Man kann einen neuen Typ deklarieren, der eine Erweiterung eines vorhandenen Typs darstellt. Man sagt, dass diese neue Unterklasse von einem vorhandenen Typ abgeleitet ist, und spricht auch von einem *abgeleiteten Typ*. Das erweiterte Telefon ist vom guten alten Telefon abgeleitet und erbt damit dessen gesamte Eigenschaften. Bei Bedarf kann man aber neue hinzufügen. Auf die Vererbung und ihre Anwendung in C++ geht Stunde 16 ein.

Polymorphie

Bei einem eingehenden Anruf verhält sich das erweiterte Telefon (ET) nicht wie gewohnt. Es klingelt nicht, sondern der Bildschirm schaltet sich ein und eine Stimme sagt: »Sie werden angerufen.« Die Telefonvermittlung weiß hiervon jedoch nichts – sie sendet keine speziellen Signale für die einzelnen Telefonarten. Von der Vermittlungsstelle geht einfach der normale Ruf aus – das normale Telefon klingelt, das elektronische Telefon trillert, und ET spricht. Jedes Telefon unternimmt »genau das Richtige«, da es die von der Telefonvermittlung kommende Nachricht interpretiert.

C++ unterstützt den Gedanken, dass verschiedene Objekte »genau das Richtige tun« über die so genannte *Funktionspolymorphie* oder *Klassenpolymorphie*. Das aus dem Griechischen stammende Wort »polymorph« bedeutet »vielgestaltig«. *Polymorphie* bezeichnet also dasselbe Ding in verschiedenen Formen. Näheres dazu finden Sie in den Stunden 17 und 18.

2.2 Die Teile eines einfachen Programms

Das einfache Programm `hello.cpp` aus der ersten Stunde weist verschiedene interessante Teile auf. Der vorliegende Abschnitt untersucht dieses Programm eingehender. Listing 2.1 zeigt noch einmal die Originalversion des Programms `hello.cpp`.

Listing 2.1: hello.cpp – demonstriert die Teile eines C++-Programms

```
1: #include <iostream>
2:
3: int main()
4: {
5:     std::cout << "Hello World!\n";
6:     return 0;
7: }
```

```
Hello World!
```

Zeile 1 bindet die Datei `iostream` in die Datei `hello.cpp` ein. Für den Compiler stellt sich das so dar, als hätte man den gesamten Inhalt der Datei `iostream` unmittelbar am Anfang von `hello.cpp` eingegeben.

2.2.1 Die Anweisung #include im Detail

Das Symbol # ist ein Signal an ein besonderes Programm. Wenn Sie nämlich den Compiler starten, ruft dieser als Erstes einen so genannten *Präprozessor* auf. Dessen Aufgabe besteht darin, den Quellcode nach Zeilen zu durchsuchen, die mit dem Nummernzeichen (#) beginnen. Sobald eine derartige Zeile auftaucht, modifiziert der Präprozessor den Code und übergibt ihn in der geänderten Form an den Compiler.

Der Präprozessor ist praktisch ein Editor Ihres Codes – und zwar zur Kompilierzeit. Die Präprozessor-Direktiven sind die Befehle für diesen Editor.

Include (englisch: einbinden) ist eine Präprozessor-Anweisung in der Bedeutung: »Es folgt ein Dateiname. Suche die Datei und lies sie genau an diese Stelle ein.« Die spitzen Klammern um den Dateinamen weisen den Präprozessor an, »an allen üblichen Plätzen nach dieser Datei zu suchen«. Wenn Ihr Compiler korrekt eingerichtet ist, bewirken die spitzen Klammern, dass der Präprozessor nach der Datei `iostream` im Verzeichnis mit allen Header-Dateien für Ihren Compiler sucht. Diese Dateien heißen »H-Dateien« oder »Include-Dateien«, weil der Compiler sie in die Quellcodedateien einbindet (engl. include) und die Dateien üblicherweise die Erweiterung `.h` erhalten.

Erweiterungen bei Include-Dateien

Entsprechend dem neuen ANSI-Standard haben Include-Dateien oftmals überhaupt keine Erweiterung des Dateinamens mehr. Manche Compiler bringen sogar zwei Versionen für jede dieser Dateien mit. Beispielsweise kann eine Installation sowohl die ältere `iostream.h` als auch die neue – dem ANSI-Standard entsprechende – Datei `iostream` enthalten. In diesem Buch verwenden wir die neuen Dateien.

Falls Sie mit einem älteren Compiler arbeiten, müssen Sie gegebenenfalls das alte Format `#include <iostream.h>` einbinden.

Die Datei `iostream` (Input-Output-Stream, Eingabe-Ausgabe-Strom) ist für `cout` erforderlich und unterstützt das Schreiben auf den Bildschirm.

Beachten Sie, dass vor `cout` die Kennzeichnung `std::` steht, die den Compiler anweist, die Standardbibliothek für Ein-/Ausgabe zu verwenden. Die folgenden Lektionen erläutern im Detail, wie das funktioniert und was dabei passiert. Fürs Erste können Sie `std::cout` einfach als Name des Objekts betrachten, das die Ausgabe behandelt, und `std::cin` als Name des Objekts, das für die Eingabe zuständig ist.

Zeile 1 bindet die Datei `iostream` in dieses Programm genauso ein, als hätte man den entsprechenden Quelltext selbst eingetippt. Wenn der Compiler die Quelldatei zu sehen bekommt, ist die Include-Datei bereits an der richtigen Stelle eingebunden und der Compiler ist so klug wie zuvor.

2.2.2 Zeilenweise Analyse

In Zeile 3 beginnt das eigentliche Programm mit einer Funktion namens `main()`. Jedes C++-Programm verfügt über eine `main()`-Funktion. Im Allgemeinen ist eine Funktion ein Codeblock, der eine oder mehrere Aktionen ausführt. Funktionen werden von anderen Funktionen aufgerufen, wobei die Funktion `main()` eine Sonderstellung einnimmt. Bei Start des Programms erfolgt der Aufruf von `main()` automatisch.

Wie alle anderen Funktionen muss auch `main()` festlegen, welche Art von Rückgabewert sie liefert. Auch hier nimmt `main()` eine Sonderstellung ein: diese Funktion gibt immer `int` zurück. Das Schlüsselwort `int` bezeichnet einen Integer (eine ganze Zahl), worauf Stunde 3 näher eingeht. Die Rückgabe eines Wertes aus einer Funktion behandelt Stunde 4 ausführlich.

Die Funktion `main()` liefert ihren Übergabewert an das Betriebssystem, sodass man diesen Wert nutzen kann, um Fehler aufzuspüren. Dieses Verfahren ist in Produktions- und Stapelverarbeitungsumgebungen üblich. Ein anderes Tool oder Skript, das Ihr Programm aufruft, kann dann Fehlermeldungen auslösen (und Sie möglicherweise zu nächtlicher Stunde per Pager herbeirufen, wenn es sich um einen wirklich wichtigen Prozess handelt).

Alle Funktionen beginnen mit einer öffnenden geschweiften Klammer ({) und enden mit einer schließenden geschweiften Klammer (}). Die geschweiften Klammern für die Funktion `main()` stehen in den Zeilen 4 und 7. Alles zwischen der öffnenden und der schließenden Klammer ist Teil der Funktion.

Die geschweiften Klammern enthalten Blöcke – in diesem Fall handelt es sich um den äußeren Block einer Funktion. Andere Arten von Blöcken lernen Sie in Kürze kennen.

Zeile 5 leistet die »Knochenarbeit« dieses Programms. Das Objekt `cout` gibt eine Meldung auf dem Bildschirm aus. Beginnend mit Stunde 8 beschäftigen wir uns mit Objekten im Allgemeinen. Zum Lieferumfang des Compilers gehören `cout` und das verwandte Objekt `cin`. Diese Objekte realisieren im System die Ausgabe auf den Bildschirm (`cout`) und die Eingabe über die Tastatur (`cin`).

Im Anschluss an das Objekt `cout` steht der Umleitungsoperator (<<), den man durch zweimaliges Betätigen der [<]-Taste erzeugt. Alles was auf den Umleitungsoperator folgt, erscheint auf dem Bildschirm. Möchten Sie eine Zeichenfolge ausgeben, ist diese in Anführungszeichen (") wie in Zeile 5 zu setzen. Eine *Textzeichenfolge* ist eine Folge von druckbaren Zeichen.

Die beiden letzten Zeichen, `\n`, weisen `cout` an, eine neue Zeile nach dem Text `Hello World!` auszugeben.

Zeile 6 gibt den Wert `0` an das Betriebssystem zurück. Bei manchen Systemen dient der Rückgabewert als Signal, ob die Funktion fehlerfrei oder fehlerhaft ausgeführt wurde. Per Konvention kennzeichnet `0` eine erfolgreiche Operation, jede andere Zahl einen Fehler. Die heutigen fensterorientierten Systeme verwenden diesen Wert fast nie, sodass alle Programme in diesem Buch eine `0` zurückgeben.

Die Funktion `main()` endet in Zeile 7 mit der schließenden geschweiften Klammer.

2.3 Kommentare

Während man ein Programm schreibt, ist alles sonnenklar und bedarf keiner weiteren Erläuterung. Kehrt man aber einen Monat später zu diesem Programm zurück, sieht die Welt schon anders aus. Die selbst verfassten Zeilen geben nur noch Rätsel auf.

Um diese Unklarheiten von vornherein zu bekämpfen und anderen Personen das Verständnis des Codes zu erleichtern, schreibt man gewöhnlich Kommentare in das Programm. Dabei handelt es sich einfach um Text, den der Compiler ignoriert, der aber den Leser über die Wirkung des Programms an bestimmten Punkten informiert.

2.3.1 Arten von Kommentaren

In C++ unterscheidet man zwei Arten von *Kommentaren*. Der Kommentar mit doppelten Schrägstrichen (//) im Stil von C++ weist den Compiler an, alles nach den Schrägstrichen bis zum Ende der Zeile zu ignorieren.

Der durch Schrägstrich und Sternchen (/*) eingeleitete Kommentar weist den Compiler an, alles zu ignorieren, bis die abschließende Kommentarmarkierung (*/) erscheint. Diese Markierungen sind Kommentare im Stil von C und wurden in C++ übernommen. Achten Sie darauf, dass jedes /* durch ein entsprechendes */ abzuschließen ist.

Viele C++-Programmierer verwenden vorrangig Kommentare im Stil von C++ und heben sich die C-artigen Kommentare für das Ausklammern von großen Programmblöcken auf. Man kann C++-Kommentare innerhalb eines durch C-Kommentare »auskommentierten« Blocks einfügen. Zwischen den C-Kommentarmarken wird alles – einschließlich der C++-Kommentare – ignoriert.

Code auskommentieren

Gehen Sie sorgsam mit Kommentaren im Stil von C um, wenn Sie Code auskommentieren, der seinerseits Kommentare im Stil von C enthält. Dabei können Sie in eine Situation gelangen, in der weniger Code als erwartet auskommentiert wird, weil sich Kommentare im Stil von C nicht verschachteln lassen. Nur weil Sie zwei /*-Kommentarmarkierungen geschrieben haben, heißt das nicht, dass Sie zwei */-Kommentarmarkierungen brauchen. Die erste */-Kommentarmarkierung, die der Compiler antrifft, »schließt« jeden momentan geöffneten /*-Kommentar. Eine gute IDE versieht auskommentierten Code mit einer abweichenden Farbe, um ihn deutlich von echtem Code zu unterscheiden. Und der Compiler meldet einen Fehler, wenn er die zusätzliche */-Kommentarmarkierung findet.

2.3.2 Kommentare in einem einfachen Beispielprogramm

Kommentare kosten nichts, der Compiler übergeht sie und sie haben keinen Einfluss auf die Leistung des Programms. Listing 2.2 verdeutlicht die Verwendung von Kommentaren.

Listing 2.2: comments.cpp – demonstriert Kommentare

```
1: #include <iostream>
2:
3: int main()
4: {
5:     /* Das ist ein Kommentar im Stil von C, der
6:           bis zum schließenden Kommentarzeichen aus
7:           Sternchen und Schrägstrich geht. */
```

```
 8:     std::cout << "Hello World!\n";
 9:     // Dieser C++-Kommentar geht nur bis zum Zeilenende.
10:     std::cout << "Dieser Kommentar ist beendet!";
11:
12:     // Diese Kommentare können allein auf einer Zeile stehen
13:     /* genau wie diese Kommentare. */
14:     return 0;
15: }
```

```
Hello World!
Dieser Kommentar ist beendet!
```

Die Kommentare in den Zeilen 5 bis 7 ignoriert der Compiler gänzlich. Das Gleiche gilt für die Kommentare in den Zeilen 9, 12 und 13. Der Kommentar auf Zeile 9 geht nur bis zum Zeilenende, während die Kommentare in den Zeilen 5 und 13 ein schließendes Kommentarzeichen erfordern.

Das Schreiben guter Kommentare beherrschen nur wenige Programmierer. Einige Unverbesserliche verwenden überhaupt keine Kommentare! Andere kommentieren jede Zeile, was für den Quellcode wenig Wert hat und nur die Unübersichtlichkeit fördert.

Wie verwendet man nun Kommentare richtig? Gehen Sie am besten davon aus, dass der Leser einen C++-Quelltext ohne weiteres versteht, aber nicht die Absichten des Programms kennt. Lassen Sie den Quellcode erzählen, *was* Sie tun, und erläutern Sie mit Kommentaren, *warum* Sie etwas tun.

2.4 Funktionen

Die Funktion `main()` ist etwas ungewöhnlich. Beim Start des Programms ruft das Betriebssystem die Funktion `main()` auf. Andere Funktionen werden während des Programmlaufs von `main()` oder von einer anderen Funktion aufgerufen.

Die Funktion `main()` gibt immer einen `int`-Wert zurück. Wie die kommenden Stunden zeigen, können andere Funktionen andere Datentypen oder auch überhaupt nichts zurückgeben.

Ein Programm wird zeilenweise entsprechend der Reihenfolge im Quellcode ausgeführt. Bei einem Funktionsaufruf verzweigt das Programm, um die Funktion auszuführen. Nach Abarbeitung der Funktion geht die Steuerung des Programms an die nächste Zeile in der aufrufenden Funktion zurück.

Stellen Sie sich vor, dass Sie ein Bild von sich selbst zeichnen. Sie zeichnen den Kopf, die Augen, die Nase – und plötzlich bricht die Bleistiftspitze ab. Sie »verzweigen« nun in die Funktion »Bleistift spitzen«. Das heißt, Sie hören mit dem Zeichnen auf, stehen auf, gehen zur Spitzmaschine, spitzen den Stift, kehren an Ihre Arbeit zurück und setzen sie dort fort, wo Sie aufgehört haben (vielleicht zeichnen Sie gerade Ihr Kinn).

2.4.1 Funktionen aufrufen

Muss ein Programm einen Dienst ausführen lassen, ruft es eine Funktion auf, um diesen Dienst auszuführen. Ist die Funktion abgearbeitet, springt das Programm zurück und setzt an der Stelle unmittelbar nach dem Funktionsaufruf fort.

Listing 2.3 verdeutlicht dieses Konzept.

Listing 2.3: callfunc.cpp – Aufruf einer Funktion

```
 1: #include <iostream>
 2:
 3: // Demonstriert einen Funktionsaufruf
 4: // Funktion gibt eine Meldung aus
 5: void DemonstrationFunction()
 6: {
 7:     std::cout << "In DemonstrationFunction\n";
 8: }
 9:
10: // Funktion main - gibt eine Meldung aus, ruft
11: // dann DemonstrationFunction auf, gibt danach
12: // eine zweite Meldung aus.
13: int main()
14: {
15:     std::cout << "In main\n" ;
16:     DemonstrationFunction();
17:     std::cout << "Zurueck in main\n";
18:     return 0;
19: }
```

```
In main
In DemonstrationFunction
Zurueck in main
```

Die Zeilen 5 bis 8 enthalten die Definition der Funktion `DemonstrationFunction()`. Eine Definition zeigt lediglich, was zu tun ist, führt aber keinerlei Aktionen aus, bis die Funktion aufgerufen wird. Beim Aufruf in Zeile 16 gibt die Funktion dann eine Meldung auf dem Bildschirm aus und kehrt zum Aufrufer zurück.

In Zeile 13 beginnt das eigentliche Programm. In Zeile 15 gibt die Funktion `main()` eine Meldung aus, dass sich das Programm soeben in der Funktion `main()` befindet. Nach der Ausgabe der Meldung ruft Zeile 16 die Funktion `DemonstrationFunction()` auf. Dieser Aufruf bewirkt die Ausführung der Befehle in `DemonstrationFunction()`. In diesem Beispiel besteht die gesamte Funktion aus dem Code in Zeile 7, der eine weitere Meldung ausgibt. Nach vollständiger Abarbeitung der Funktion `DemonstrationFunction()` in Zeile 8 kehrt die Programmausführung an die Stelle zurück, wo der Aufruf der Funktion erfolgte. Im Beispiel kehrt das Programm zu Zeile 17 zurück und die Funktion `main()` gibt die abschließende Meldung aus.

2.4.2 Funktionen verwenden

Funktionen geben entweder einen Wert oder `void` (d.h. nichts) zurück. Eine Funktion zur Addition von zwei ganzen Zahlen liefert sicherlich die Summe zurück und man definiert diesen Rückgabewert vom Typ Integer. Eine Funktion, die lediglich eine Meldung ausgibt, hat nichts zurückzugeben und wird daher als `void` (zu deutsch: leer) deklariert.

Funktionen gliedern sich in Kopf und Rumpf. Der Kopf besteht wiederum aus dem Rückgabetyp, dem Funktionsnamen und den Parametern. Mit *Parametern* lassen sich Werte an eine Funktion übergeben. Soll eine Funktion zum Beispiel zwei Zahlen addieren, stellen die Zahlen die Parameter für die Funktion dar. Ein typischer Funktionskopf sieht folgendermaßen aus:

```
int Sum(int a, int b)
```

Als Parameter bezeichnet man im engeren Sinn nur die Deklaration des zu übergebenden Datentyps. Der eigentliche Wert, den die aufrufende Funktion übergibt, heißt *Argument*. Viele Programmierer machen keinen Unterschied zwischen den Begriffen »Parameter« und »Argument«, während andere genau auf diese technische Unterscheidung achten. Im Buch kommen beide Ausdrücke gleichberechtigt vor.

Der Name der Funktion und deren Parameter (das heißt, der Funktionskopf ohne den Rückgabewert) heißt *Signatur* der Funktion.

Der Rumpf einer Funktion besteht aus einer öffnenden geschweiften Klammer, einer beliebigen Zahl von Anweisungen (0 oder mehr) und einer schließenden geschweiften Klammer. Die Anweisungen erledigen die Arbeit der Funktion. Eine Funktion kann einen Wert mit der Anweisung `return` zurückgeben. Diese Anweisung bewirkt auch das Verlassen der Funktion. Wenn man keine `return`-Anweisung vorsieht, liefert die Funktion am Ende automatisch `void` zurück. Der zurückgegebene Wert muss dem im Funktionskopf deklarierten Typ entsprechen.

2.4.3 Parameter in Funktionen

Listing 2.4 zeigt eine Funktion, die zwei ganzzahlige Parameter übernimmt und einen ganzzahligen Wert zurückgibt.

Listing 2.4: func.cpp – demonstriert eine einfache Funktion

```
 1: #include <iostream>
 2:
 3: int Add (int x, int y)
 4: {
 5:     std::cout << "In Add(), erhalten " << x << " und " << y << "\n";
 6:     return (x+y);
 7: }
 8:
 9: int main()
10: {
11:     std::cout << "Ich bin in main()!\n";
12:     std::cout << "\nAufruf von Add()\n";
13:     std::cout << "Der Rueckgabewert ist: " << Add(3,4);
14:     std::cout << "\nZurueck in main().\n";
15:     std::cout << "\nBeenden...\n\n";
16:     return 0;
17: }
```

```
Ich bin in main()!

Aufruf von Add()
In Add(), erhalten 3 und 4
Der Rueckgabewert ist: 7

Zurueck in main().

Beenden...
```

Zeile 2 definiert die Funktion `Add()`, die zwei ganzzahlige Parameter übernimmt und einen ganzzahligen Wert zurückgibt. Das Programm selbst beginnt in Zeile 11 und gibt hier eine Meldung aus.

In Zeile 13 gibt die Funktion `main()` zuerst eine Meldung und anschließend den Rückgabewert des Aufrufs `Add(3,4)` aus. Durch diesen Aufruf verzweigt der Programmablauf in die Funktion `Add()`, die auf Zeile 3 beginnt. Die beiden an `Add()` übergebenen Werte werden als die Parameter `x` und `y` dargestellt. Die Funktion addiert die beiden Werte und Zeile 6 liefert das Ergebnis an den Aufrufer zurück.

Der in Zeile 13 ausgegebene Rückgabewert erscheint in der Ausgabe (7). Die Funktion `main()` gibt dann auf den Zeilen 14 und 15 eine Meldung aus und das Programm endet, wenn der Rücksprung aus der Funktion `main()` erfolgt. Die Steuerung geht daraufhin wieder an das Betriebssystem über.

2.5 Fragen und Antworten

F: Was bewirkt die Anweisung `#include`?

A: Es handelt sich um eine Direktive an den Präprozessor, der bei Ausführung des Compilers aufgerufen wird. Diese spezielle Direktive veranlasst, dass die nach dem Schlüsselwort `include` genannte Datei eingelesen wird, als würde man sie an dieser Stelle in den Quellcode eintippen.

F: Worin liegt der Unterschied zwischen den Kommentarformaten // und /*?

A: Der Wirkungsbereich eines Kommentars mit doppelten Schrägstrichen (//) erstreckt sich nur bis zum Zeilenende. Die Kommentare mit Schrägstrich und Sternchen gelten so lange, bis das Abschlusszeichen des Kommentars (*/) erscheint. Denken Sie daran, dass ein Kommentar mit /* nicht einfach durch das Ende einer Funktion abgeschlossen wird. Bei derartigen Kommentaren muss man immer das Abschlusszeichen schreiben, da andernfalls ein Fehler zur Kompilierzeit entsteht.

F: Was unterscheidet einen guten von einem schlechten Kommentar?

A: Ein guter Kommentar sagt dem Leser, warum die betreffende Codestelle etwas tut oder welcher Codeabschnitt wofür verantwortlich ist. Ein schlechter Kommentar wiederholt lediglich, was eine bestimmte Codezeile bewirkt. Die Codezeilen sollten so geschrieben sein, dass sie für sich selbst sprechen: Allein aus der niedergeschriebenen Anweisung sollte ohne weitere Kommentare ersichtlich sein, was der Code ausführt.

2.6 Workshop

Nachdem Sie sich mit den Teilen eines C++-Programms auseinandergesetzt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen ausführen, um Ihr Wissen zum Compiler zu vertiefen.

2.6.1 Kontrollfragen

1. Welchen Datentyp gibt `main()` zurück?
2. Was bewirken die geschweiften Klammern?
3. Worin besteht der Unterschied zwischen einem Compiler und einem Interpreter?
4. Warum ist Wiederverwendbarkeit so wichtig?

2.6.2 Übungen

1. Teilen Sie Zeile 5 von Listing 2.1 auf mehrere Zeilen auf (`Hello` auf eine Zeile und `World` auf eine andere). Hat sich die Art und Weise der Ergebnisausgabe in irgendeiner Form geändert? Wie lässt sich erreichen, dass die Wörter auf verschiedenen Ausgabezeilen erscheinen?
2. Wie verhält sich Ihr Compiler, wenn Sie Listing 2.1 mit den Direktiven `#include <iostream>` bzw. `#include <iostream.h>` kompilieren? Wenn Sie wissen, ob der Compiler nur einen oder beide Dateinamen akzeptiert, hilft Ihnen das später, Fehler in derartigen Anweisungen zu vermeiden bzw. zu erkennen.
3. Kompilieren und linken Sie einige Programme einmal von der integrierten Entwicklungsumgebung und einmal von der Befehlszeile aus. Unterscheiden sich die Ergebnisse bei bestimmten Programmen?

2.6.3 Antworten zu den Kontrollfragen

1. Die Funktion `main()` gibt den Typ `int` zurück – d.h. einen ganzzahligen Wert.
2. Geschweifte Klammern zeigen den Anfang und das Ende eines Blocks an. Sie umschließen den Inhalt der Funktion oder des Programms, die/das wir geschrieben haben.
3. Ein Compiler konvertiert das gesamte Programm in Maschinensprache, bevor es ausgeführt wird. Ein Interpreter konvertiert jede Zeile einzeln während der Programmausführung.
4. Weshalb greifen wir in einer Welt jenseits der Programmierung auf vorhandene Gedanken zurück? Weil es leichter und schneller sowie kostengünstiger ist etwas wiederzuverwenden, als alles von Grund auf neu zu entwickeln. Viele Programmierer möchten aber die Kontrolle über ihren Code und ihre Arbeit nicht aufgeben, sodass sie quasi die gemeinsame Nutzung und die damit verbundenen Denkprozesse erst erlernen müssen.

KAPITEL 3

jetzt lerne ich

Variablen und Konstanten

In dieser Stunde lernen Sie,

- wie man Variablen und Konstanten deklariert und definiert,
- wie man Werte an Variablen zuweist und diese Werte manipuliert,
- wie man den Wert einer Variablen auf dem Bildschirm ausgibt.

3.1 Was ist eine Variable?

Aus Sicht des Programmierers ist eine *Variable* eine Stelle im Hauptspeicher des Computers, an der man einen Wert ablegen und später wieder abrufen kann.

Um das zu verstehen, muss man zuerst etwas über die Arbeitsweise des Hauptspeichers wissen. Man kann sich den Hauptspeicher als eine Reihe von Fächern vorstellen, die in einer langen Reihe angeordnet sind. Jedes Fach – oder jede Speicherstelle – ist fortlaufend nummeriert. Diese Nummern bezeichnet man als Speicheradressen oder einfach als Adressen.

Variablen haben nicht nur Adressen, sondern auch Namen. Beispielsweise kann man eine Variable namens `myAge` erzeugen. Diese Variable ist ein Bezeichner für eines der Fächer, damit man es leicht finden kann, ohne seine Speicheradresse zu kennen.

Abbildung 3.1 verdeutlicht dieses Konzept. Wie die Abbildung zeigt, haben wir eine Variable namens `myVariable` deklariert, die an der Speicheradresse 103 beginnt.

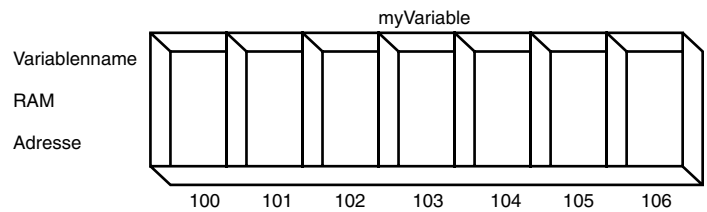


Abbildung 3.1: Schematische Darstellung des Hauptspeichers

Was ist RAM?

RAM steht für Random Access Memory – Speicher mit wahlfreiem Zugriff. Das ist der elektronische Speicher, den der Computer bei der Programmausführung verwendet. Der RAM gehört zur Kategorie der flüchtigen Speicher. Sämtliche Informationen im RAM gehen verloren, wenn der Computer ausgeschaltet wird.

Wenn Sie ein Programm ausführen, wird es von der Datei auf dem Datenträger (Festplatte) in den RAM geladen. Alle Variablen werden ebenso im RAM erzeugt. Spricht ein Programmierer vom Speicher, meint er damit gewöhnlich den RAM.

3.1.1 Speicher reservieren

Wenn man eine Variable in C++ definiert, muss man dem Compiler nicht nur ihren Namen, sondern auch den Typ der aufzunehmenden Informationen mitteilen, zum Beispiel Ganzzahl oder Zeichen. Wir nennen das den *Typ* der Variablen. Anhand dieser Information weiß der Compiler, wie viel Platz im Speicher für die Aufnahme des *Wertes* der Variablen zu reservieren ist.

Jedes Fach ist ein Byte groß. Wenn die erzeugte Variable zwei Bytes benötigt, muss man zwei Bytes im Speicher – oder zwei Fächer – reservieren. Der Typ der Variablen (zum Beispiel int) sagt dem Compiler, wie viel Speicher (oder wie viele Fächer) er für die Variable reservieren muss. Werte stellt der Computer letztlich in Form von Bits und Bytes dar und da man den Speicherbedarf in Byte misst, muss man diese Konzepte auch kennen und mit ihnen vertraut sein.

3.1.2 Größe von Integer-Werten

Eine Variable vom Typ `char` (zur Aufnahme von Zeichen) ist gewöhnlich ein Byte lang. Ein `short int` belegt auf den meisten Computern zwei Byte, ein `long int` ist normalerweise vier Byte lang, und ein `int` (ohne das Schlüsselwort `short` oder `long`) kann zwei oder vier Byte einnehmen. Unter Windows 95/98, Windows XP oder Windows NT/2000/2003 können Sie davon aus-

gehen, dass der Typ `int` vier Byte belegt, wenn Sie mit einem modernen Compiler arbeiten.

Allerdings sollten Sie sich nicht einfach darauf verlassen, dass die obige Aussage für Ihr System zutrifft. Prüfen Sie selbst! Dabei können Sie sich davon überzeugen, dass die Größe eines `short int` kleiner oder gleich der Größe eines `int` ist und dass ein `int` die gleiche Größe wie ein `long` hat oder kleiner ist. Gleitpunkttypen (`float`, `double`) sind eine ganz andere Spezies – wir gehen gleich näher darauf ein.

Mit dem Programm in Listing 3.1 lässt sich die genaue Größe dieser Typen auf Ihrem Computer bestimmen.

Listing 3.1: sizer.cpp – Die Größe von Variablentypen für einen Computer bestimmen

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:     std::cout << "Groesse eines int:\t\t";
 6:     std::cout << sizeof(int) << " Bytes.\n";
 7:     std::cout << "Groesse eines short int:\t";
 8:     std::cout << sizeof(short) << " Bytes.\n";
 9:     std::cout << "Groesse eines long int:\t";
10:     std::cout << sizeof(long) << " Bytes.\n";
11:     std::cout << "Groesse eines char:\t\t";
12:     std::cout << sizeof(char) << " Bytes.\n";
13:     std::cout << "Groesse eines bool:\t\t";
14:     std::cout << sizeof(bool) << " Bytes.\n";
15:     std::cout << "Groesse eines float:\t\t";
16:     std::cout << sizeof(float) << " Bytes.\n";
17:     std::cout << "Groesse eines double:\t";
18:     std::cout << sizeof(double) << " Bytes.\n";
19:
20:     return 0;
21: }
```

```
Groesse eines int:              4 Bytes.
Groesse eines short int:        2 Bytes.
Groesse eines long int:         4 Bytes.
Groesse eines char:             1 Bytes.
Groesse eines bool:             1 Bytes.
Groesse eines float:            4 Bytes.
Groesse eines double:           8 Bytes.
```

Die tatsächliche Anzahl der angezeigten Bytes kann auf Ihrem Computer abweichen.

Der größte Teil von Listing 3.1 sollte Ihnen bekannt vorkommen. Das Neue hier ist die Verwendung der Funktion `sizeof()` in den Zeilen 6 bis 18. Die Funktion `sizeof()` gehört zum Lieferumfang des Compilers und gibt die Größe des als Parameter übergebenen Objekts an. Beispielsweise wird in Zeile 6 das Schlüsselwort `int` an die Funktion `sizeof()` übergeben. Mittels `sizeof()` kann man feststellen, ob auf einem bestimmten Computer ein `int` gleich einem `long int` ist und 4 Byte belegt.

3.1.3 Mit und ohne Vorzeichen – signed und unsigned

Die ganzzahligen Typen kommen außerdem in zwei Versionen vor: mit Vorzeichen (`signed`) und ohne Vorzeichen (`unsigned`). Hier liegt der Gedanke zugrunde, dass man gelegentlich negative Zahlen benötigt, aber nicht immer. Ganze Zahlen (`short` und `long`) ohne das Wort `unsigned` werden als `signed` (das heißt vorzeichenbehaftet) angenommen. Vorzeichenbehaftete Ganzzahlen sind entweder negativ oder positiv, während ganze Zahlen ohne Vorzeichen (`unsigned int`) immer positiv sind. Merke: `signed` ist der Standard für ganzzahlige Variablen.

Da sowohl für vorzeichenbehaftete als auch vorzeichenlose Ganzzahlen dieselbe Anzahl von Bytes zur Verfügung steht, ist die größte Zahl, die man in einem `unsigned int` speichern kann, doppelt so groß wie die größte positive Zahl, die man in einem `signed int` unterbringt. Ein `unsigned short int` kann Zahlen von 0 bis 65535 speichern. Bei einem `signed short int` ist die Hälfte der Zahlen negativ. Daher kann ein `signed short` Zahlen im Bereich von –32768 bis 32767 darstellen.

Bit für signed und unsigned Ganzzahlen

Bei vorzeichenbehafteten Ganzzahlen ist ein Bit für das Vorzeichen reserviert; bei vorzeichenlosen Ganzzahlen steht dieses Bit für einen größeren Wertebereich zur Verfügung. Die Summe aller möglichen Zahlen ist bei `signed` und `unsigned` Variablen gleich. Der Unterschied besteht darin, wie die Werte dargestellt werden!

3.1.4 Grundlegende Variablentypen

In C++ gibt es weitere Variablentypen, die man zweckentsprechend in ganzzahlige Variablen (die bisher behandelten Typen), Gleitpunktvariablen und Zeichenvariablen einteilt.

Im Englischen verwendet man als Dezimalzeichen den Punkt, im Deutschen das Komma. Da C++ nur den Punkt akzeptiert, verwenden wir hier den Begriff *Gleitpunktzahlen* statt *Gleitkommazahlen* und geben Zahlenwerte in der entsprechenden Form mit Dezimalpunkt als Dezimalzeichen und mit Komma als Tausendertrennzeichen an.

Die Werte von Gleitpunktvariablen lassen sich als Zahlen mit gebrochenem Anteil ausdrücken – das heißt, als reelle Zahlen. Zeichenvariablen nehmen ein einzelnes Byte auf und dienen der Speicherung der 256 möglichen Zeichen und Symbole der ASCII- und erweiterten ASCII-Zeichensätze.

Der *ASCII-Zeichensatz* ist ein Standard, der die im Computer verwendeten Zeichen definiert. ASCII steht als Akronym für American Standard Code for Information Interchange (amerikanischer Standard-Code für den Informationsaustausch). Nahezu jedes Computer-Betriebssystem unterstützt ASCII. Daneben sind meistens weitere internationale Zeichensätze möglich.

Die in C++-Programmen verwendeten Variablentypen sind in Tabelle 3.1 aufgeführt. Diese Tabelle zeigt den Variablentyp, den belegten Platz im Speicher (Grundlage ist der Computer des Autors) und den möglichen Wertebereich, der sich aus der Größe des Variablentyps ergibt. Vergleichen Sie dazu die Ausgabe des Programms aus Listing 3.1.

Tabelle 3.1: Variablentypen

Typ	Größe	Wert
unsigned short int	2 Byte	0 bis 65,535
short int	2 Byte	–32,768 bis 32,767
unsigned long int	4 Byte	0 bis 4,294,967,295
long int	4 Byte	–2,147,483,648 bis 2,147,483,647
int	4 Byte	–2,147,483,648 bis 2,147,483,647
unsigned int	4 Byte	0 bis 4,294,967,295
char	1 Byte	256 Zeichenwerte
bool	1 Byte	`true` oder `false`
float	4 Byte	1.2*e*–38 bis 3.4e38
double	8 Byte	2.2*e*–308 bis 1.8e308

Auch wenn es viele als schlechten Programmierstil ansehen: Man kann eine `char`-Variable für sehr kleine Ganzzahlen verwenden.

3.2 Variablen definieren

Eine Variable erzeugt – oder definiert – man, indem man den Typ, mindestens ein Leerzeichen, den Variablennamen und ein Semikolon schreibt. Als Variablenname eignet sich nahezu jede Buchstaben-/Ziffernkombination, die allerdings keine Leerzeichen enthalten darf. Gültige Variablennamen sind zum Beispiel `x`, `J23qrsnf` und `meinAlter`. Geschickt gewählte Variablennamen sagen bereits etwas über den Verwendungszweck der Variablen aus und erleichtern damit das Verständnis für den Programmablauf. Die folgende Anweisung definiert eine Integer-Variable namens `meinAlter`:

```
int meinAlter;
```

In der Programmierpraxis hat es sich eingebürgert, möglichst aussagekräftige Namen für die Variable zu verwenden. Namen wie `meinAlter` oder `wieViele` sind leichter zu verstehen und zu merken als Namen wie `xJ4` oder `theInt`. Bei Verwendung treffender Variablennamen kommt man mit weniger Kommentaren zur Erläuterung des Codes aus.

Probieren Sie Folgendes aus: Versuchen Sie anhand der ersten Codezeilen zu ergründen, was die Codefragmente bewirken:

Beispiel 1

```
main()
{
     unsigned short x;
     unsigned short y;
     unsigned int z;
     z = x * y;
}
```

Beispiel 2

```
main ()
{
     unsigned short Breite;
     unsigned short Laenge;
     unsigned int Flaeche;
     Flaeche = Breite * Laenge;
}
```

3.2.1 Groß-/Kleinschreibung

C++ beachtet die *Groß-/Kleinschreibung* und behandelt demnach Großbuchstaben und Kleinbuchstaben als verschiedene Zeichen. Eine Variable namens `alter` unterscheidet sich von `Alter` und diese wiederum von `ALTER`.

Bestimmte Compiler gestatten es, die Abhängigkeit von der Groß-/Kleinschreibung zu deaktivieren. Das ist allerdings nicht zu empfehlen, da Ihre Programme bei anderen Compilern eventuell nicht funktionieren und andere C++-Programmierer mit Ihrem Code nicht klarkommen.

Viele Programmierer bevorzugen bei Variablennamen Kleinbuchstaben. Wenn der Name aus zwei Wörtern besteht (zum Beispiel `mein Auto`), gibt es zwei übliche Konventionen: `mein_Auto` oder `meinAuto`. Die zweite Form heißt auch »Kamelnotation«, weil die Großbuchstaben in den Namen an Kamelhöcker erinnern. Man kann sich die beiden Formen auch als »Unix« und »Microsoft« -Methoden vorstellen, basierend auf den gebräuchlichsten Konventionen. Ein Programmierer, der mit Unix groß geworden ist oder lange Zeit in einem Unix-Shop gearbeitet hat, verwendet die erste Form. Jemand mit Microsoft-Produkthintergrund greift eher zur zweiten. Den Compiler berührt das gar nicht, solange die Namen einheitlich sind.

3.2.2 Schlüsselwörter

In C++ sind bestimmte Wörter reserviert, die man unter anderem nicht als Variablennamen verwenden darf. Es handelt sich dabei um die Schlüsselwörter, mit denen der Compiler das Programm steuert. Zu den Schlüsselwörtern gehören zum Beispiel `if`, `while`, `for` und `main`. In der Dokumentation Ihres Compilers finden Sie eine vollständige Liste. Im Allgemeinen ist aber jeder aussagekräftige Name für eine Variable mit an Sicherheit grenzender Wahrscheinlichkeit kein Schlüsselwort.

Variablen können ein Schlüsselwort als Teil ihres Namens enthalten, allerdings nicht den gesamten Namen ausmachen. Variablennamen wie `main_flag` und `forEver` sind durchaus zulässig, während `main` und `for` nicht erlaubt sind.

Was Sie tun sollten

Definieren Sie eine Variable durch Angabe des Typs und dem sich anschließenden Variablennamen.

Verwenden Sie aussagekräftige Variablennamen.

Denken Sie daran, dass C++ die Groß-/Kleinschreibung berücksichtigt.

Informieren Sie sich über die Anzahl der Bytes für jeden Variablentyp im Speicher und die möglichen Werte, die sich mit dem jeweiligen Typ darstellen lassen.

Verwenden Sie keine Schlüsselwörter von C++ als Variablennamen.

Verwenden Sie keine `unsigned`-Variablen für negative Zahlen.

3.3 Mehrere Variablen gleichzeitig erzeugen

In einer Anweisung lassen sich mehrere Variablen desselben Typs gleichzeitig erzeugen, indem man den Typ schreibt und dahinter die Variablennamen durch Kommas getrennt angibt. Dazu ein Beispiel:

```
int meinAlter, meinGewicht;      // Zwei Variablen vom Typ int
long Flaeche, Breite, Laenge;    // Drei Variablen vom Typ long
```

Wie man sieht, werden `meinAlter` und `meinGewicht` gemeinsam als Variablen vom Typ `int` deklariert. Die zweite Zeile deklariert drei eigenständige Variablen vom Typ `long` mit den Namen `Flaeche`, `Breite` und `Laenge`. Der Typ (`long`) wird allen Variablen zugewiesen, sodass man in ein und derselben Definitionsanweisung keine unterschiedlichen Typen festlegen kann.

Guter Kommentierungsstil

Es gehört zum guten Programmierstil, zusammen mit den Variablendeklarationen aussagekräftige Kommentare auf derselben Zeile anzugeben. Zum einen erfahren andere Programmierer, welchen Zweck Sie mit den Variablen verfolgen, zum anderen dienen Ihnen diese Kommentare als Gedankenstütze, selbst wenn sich die Bedeutung der Variablen nicht aus dem Namen ableiten lässt.

3.4 Werte an Variablen zuweisen

Einen Wert weist man einer Variablen mithilfe des Zuweisungsoperators (`=`) zu. Zum Beispiel formuliert man die Zuweisung des Wertes `5` an die Variable `Breite` wie folgt:

```
unsigned short Breite;
Breite = 5;
```

Diese Schritte kann man zusammenfassen und die Variable `Breite` bei ihrer Definition initialisieren:

```
unsigned short Breite = 5;
```

Die Initialisierung sieht fast wie eine Zuweisung aus und bei Integer-Variablen gibt es auch kaum einen Unterschied. Bei der späteren Behandlung von Konstanten werden Sie sehen, dass man bestimmte Werte initialisieren muss, da Zuweisungen nicht möglich sind.

Listing 3.2 zeigt ein vollständiges Programm, das Sie sofort kompilieren können. Es berechnet die Fläche eines Rechtecks und schreibt das Ergebnis auf den Bildschirm.

Listing 3.2: usevar.cpp – Einsatz von Variablen

```
 1: // Einsatz von Variablen
 2: #include <iostream>
 3:
 4: int main()
 5: {
 6:     unsigned short int Width = 5, Length;
 7:     Length = 10;
 8:
 9:     // einen unsigned short erzeugen und mit dem Ergebnis der
Multiplikation
10:     // von Width und Length initialisieren
11:     unsigned short int Area = Width * Length;
12:
13:     std::cout << "Breite:  " << Width << "\n";
14:     std::cout << "Laenge:  "  << Length << std::endl;
15:     std::cout << "Flaeche: " << Area << std::endl;
16:     return 0;
17: }
```

```
Breite:  5
Laenge:  10
Flaeche: 50
```

Zeile 2 enthält die für `cout` erforderliche `include`-Anweisung für die Bibliothek von `iostream`. In Zeile 4 beginnt das Programm.

In Zeile 6 wird die Variable `Width` als `unsigned short int` definiert und mit dem Wert `5` initialisiert. Eine weitere Variable dieses Typs, `Length`, wird ebenfalls hier definiert, aber nicht initialisiert. In Zeile 7 erfolgt die Zuweisung des Wertes `10` an die Variable `Length`.

Zeile 11 definiert die Variable `Area` vom Typ `int` und initialisiert sie mit dem Wert, der sich aus der Multiplikation von `Width` und `Length` ergibt. In den Zeilen 13 bis 15 erfolgt die Ausgabe der Variablenwerte auf dem Bildschirm. Beachten Sie, dass das spezielle Wort `endl` eine neue Zeile erzeugt.

Eine Zeile beenden

Die Zeichenfolge `endl` steht für *end line* (Zeile beenden). Der letzte Buchstabe ist also ein (kleines) »L« und keine »1«, was man leicht verwechseln kann.

3.5 Das Schlüsselwort typedef

Es ist lästig, zeitraubend und vor allem fehleranfällig, wenn man häufig `unsigned short int` schreiben muss. In C++ kann man einen Alias für diese Wortfolge mithilfe des Schlüsselwortes `typedef` (für Typdefinition) erzeugen.

Mit diesem Schlüsselwort erzeugt man lediglich ein Synonym und keinen neuen Typ (wie es Stunde 7 erläutert). An das Schlüsselwort `typedef` fügt man einen vorhandenen Typ und danach den neuen Namen an. Beispielsweise erzeugt

```
typedef unsigned short int USHORT
```

den neuen Namen `USHORT`, den man an jeder Stelle verwenden kann, wo man sonst `unsigned short int` schreiben würde. Listing 3.3 verwendet als Neuauflage von Listing 3.2 die Typdefinition `USHORT` anstelle von `unsigned short int`.

Listing 3.3: typedefer.cpp – Demonstration von typedef

```
 1: // *****************
 2: // Zeigt die Verwendung des Schlüsselwortes typedef
 3: #include <iostream>
 4:
 5: typedef unsigned short int USHORT;    // mit typedef definiert
 6:
 7: int main()
 8: {
 9:    USHORT Width = 5;
10:    USHORT Length;
11:    Length = 10;
12:    USHORT Area = Width * Length;
13:    std::cout << "Breite : " << Width << "\n";
14:    std::cout << "Laenge : " << Length << std::endl;
15:    std::cout << "Flaeche: " << Area << std::endl;
16:    return 0;
17: }
```

```
Breite:  5
Laenge:  10
Flaeche: 50
```

Bei manchen Compilern führt dieser Code zur Warnung, dass die Umwandlung einen Genauigkeitsverlust bewirkt. Das hängt damit zusammen, dass das Produkt der beiden USHORTs in Zeile 12 größer als der Wertebereich einer Variablen vom Typ `unsigned short` werden und die Zuweisung dieses Produkts an `Area` zu einem Abschneiden führen kann. Im vorliegenden Fall kann man diese Warnung allerdings gefahrlos ignorieren.

Zeile 5 verwendet das mit `typedef` erzeugte Synonym `USHORT` für `unsigned short int`. Das Programm ist im Übrigen mit Listing 3.2 identisch und liefert auch die gleichen Ausgaben.

Man kann sich die Wirkung von `typedef` so vorstellen, dass die zugrunde liegende Definition (hier `unsigned short int`) an allen Stellen eingesetzt wird, wo man den neuen Typ (hier `USHORT`) eingetippt hat.

3.6 Wann verwendet man short und wann long?

Neueinsteiger in die C++-Programmierung wissen oft nicht, wann man eine Variable als `long` und wann als `short` deklarieren sollte. Die Regel ist einfach: Wenn der in der Variablen zu speichernde Wert zu groß für seinen Typ werden kann, wählt man einen größeren Typ.

Wie Tabelle 3.1 zeigt, können ganzzahlige Variablen vom Typ `unsigned short` (vorausgesetzt, dass sie aus 2 Bytes bestehen) nur Werte bis zu 65535 aufnehmen. Variablen vom Typ `signed short` können nur halb so große Zahlen speichern. Obwohl Integer-Zahlen vom Typ `unsigned long` sehr große Ganzzahlen aufnehmen können (`4,294,967,295`), hat auch dieser Typ einen begrenzten Wertebereich. Benötigt man größere Zahlen, muss man auf `float` oder `double` ausweichen und einen gewissen Genauigkeitsverlust in Kauf nehmen. Variablen vom Typ `float` oder `double` können zwar extrem große Zahlen speichern, allerdings sind auf den meisten Computern nur die ersten 7 bzw. 19 Ziffern signifikant. Das bedeutet, dass die Zahl nach dieser Stellenzahl gerundet wird.

3.6.1 Bereichsüberschreitung bei Integer-Werten vom Typ unsigned

Die Tatsache, dass Ganzzahlen vom Typ `unsigned long` nur einen begrenzten Wertebereich aufnehmen können, ist nur selten ein Problem. Aber was passiert, wenn der Platz im Verlauf des Programms zu klein wird?

Wenn eine Ganzzahl vom Typ `unsigned` ihren Maximalwert erreicht, schlägt der Zahlenwert um und die Zählung beginnt von vorn. Vergleichbar ist das mit einem Kilometerzähler. Listing 3.4 demonstriert den Versuch, einen zu großen Wert in einer Variablen vom Typ `short int` abzulegen.

Listing 3.4: toobigu.cpp – Speichern eines zu großen Wertes in einer Variablen vom Typ unsigned int

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:     unsigned short int smallNumber;
 6:     smallNumber = 65535;
 7:     std::cout << "Kleine Zahl:" << smallNumber << std::endl;
 8:     smallNumber++;
 9:     std::cout << "Kleine Zahl:" << smallNumber << std::endl;
10:     smallNumber++;
11:     std::cout << "Kleine Zahl:" << smallNumber << std::endl;
12:     return 0;
13: }
```

```
Kleine Zahl: 65535
Kleine Zahl: 0
Kleine Zahl: 1
```

Zeile 5 deklariert `smallNumber` vom Typ `unsigned short int` (auf dem Computer des Autors 2 Byte für einen Wertebereich zwischen 0 und 65535). Zeile 6 weist den Maximalwert an `smallNumber` zu und gibt ihn in Zeile 7 aus.

Die Anweisung in Zeile 8 inkrementiert `smallNumber`, das heißt, addiert den Wert 1. Das Symbol für das Inkrementieren ist ++ (genau wie der Name C++ eine Inkrementierung von C symbolisieren soll). Der Wert in `smallNumber` sollte nun 65536 lauten. Da aber Ganzzahlen vom Typ `unsigned short` keine Zahlen größer als 65535 speichern können, schlägt der Wert zu 0 um. Die Ausgabe dieses Wertes findet in Zeile 9 statt.

Die Anweisung in Zeile 10 inkrementiert `smallNumber` erneut. Es erscheint nun der neue Wert 1.

3.6.2 Bereichsüberschreitung bei Integer-Werten vom Typ signed

Im Gegensatz zu `unsigned` Integer-Zahlen besteht bei einer Ganzzahl vom Typ `signed` die Hälfte des Wertebereichs aus negativen Werten. Den Vergleich mit einem Kilometerzähler stellen wir nun so an, dass er bei einem positiven Überlauf vorwärts und bei negativen Zahlen rückwärts läuft. Vom Zählerstand 0 ausgehend erscheint demnach die Entfernung 1 Kilometer entweder als 1 oder –1. Wenn man den Bereich der positiven Zahlen verlässt, gelangt man zur (dem Betrag nach) größten negativen Zahl und zählt dann weiter herauf bis Null. Listing 3.5 zeigt die Ergebnisse, wenn man auf die maximale positive Zahl in einem (`signed`) `short int` eine 1 addiert.

Listing 3.5: toobigs.cpp – Addieren einer zu großen Zahl auf eine Zahl vom Typ signed int

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:     short int smallNumber;
 6:     smallNumber = 32767;
 7:     std::cout << "Kleine Zahl:" << smallNumber << std::endl;
 8:     smallNumber++;
 9:     std::cout << "Kleine Zahl:" << smallNumber << std::endl;
10:     smallNumber++;
11:     std::cout << "Kleine Zahl:" << smallNumber << std::endl;
12:     return 0;
13: }
```

```
Kleine Zahl: 32767
Kleine Zahl: -32768
Kleine Zahl: -32767
```

Zeile 5 deklariert `smallNumber` dieses Mal als `signed short int` (wenn man nicht explizit `unsigned` festlegt, gilt per Vorgabe `signed`). Das Programm läuft fast genau wie das vorherige, liefert aber eine andere Ausgabe. Um diese Ausgabe zu verstehen, muss man die Bit-Darstellung vorzeichenbehafteter (`signed`) Zahlen in einer Integer-Zahl von 2 Byte Länge kennen.

Letztlich verhält es sich genau wie mit einem `unsigned int`: Ein Wert vom Typ `signed int` schlägt vom größten positiven Wert zu seinem kleinsten negativen (oder: dem Betrag nach größten negativen) Wert um. Inkrementiert man ausgehend von `32767`, erhält man erst `–32768` und dann `–32767`, `–32766` usw.

3.7 Konstanten

Konstanten sind ebenso wie Variablen benannte Speicherstellen. Während sich Variablen aber ändern können, behalten Konstanten – wie der Name bereits sagt – immer ihren Wert.

Man muss eine Konstante bei ihrer Erzeugung initialisieren, spätere Wertzuweisungen sind nicht möglich. Eine einmal initialisierte Konstante behält diesen Wert während der gesamten Programmausführung bei.

C++ kennt zwei Arten von Konstanten: *literale* und *symbolische*.

3.7.1 Literale Konstanten

Eine literale Konstante ist ein Wert, den man direkt in das Programm an der Stelle des Vorkommens eintippt. In der Anweisung

```
int meinAlter = 39;
```

ist `meinAlter` eine Variable vom Typ `int`, während `39` eine literale Konstante bezeichnet. Man kann 39 keinen Wert zuweisen oder diesen Wert ändern.

3.7.2 Symbolische Konstanten

Eine symbolische Konstante wird genau wie eine Variable durch einen Namen repräsentiert. Allerdings lässt sich im Gegensatz zu einer Variablen der Wert einer Konstanten nach deren Initialisierung nicht ändern.

Wenn Ihr Programm eine Integer-Variable namens `Studenten` und eine weitere namens `Klassen` enthält, kann man die Anzahl der Studenten berechnen, wenn die Anzahl der Klassen bekannt ist und man weiß, dass 15 Studenten zu einer Klasse gehören:

```
Studenten = Klassen * 15;
```

Ihr erster Operator

Das Symbol * bezeichnet eine Multiplikation.

In diesem Beispiel ist `15` eine literale Konstante. Der Code wäre leichter zu lesen, zu verstehen und zu warten, wenn man für diesen Wert eine symbolische Konstante setzt:

```
Studenten = Klassen * StudentenProKlasse
```

Wenn man später die Anzahl der Studenten pro Klasse ändern möchte, braucht man das nur in der Definition der Konstanten `StudentenProKlasse` vorzunehmen, ohne dass man alle Stellen ändern muss, wo man diesen Wert

verwendet hat. Andernfalls besteht nämlich das Risiko, dass man eine Stelle beim Ändern übersieht. Dann ist es sehr schwer, den logischen Fehler aufzuspüren.

Konstanten mit `#define` definieren

Um eine Konstante auf die herkömmliche Weise zu definieren, schreibt man:

```
#define StudentenProKlasse 15
```

Beachten Sie, dass `StudentenProKlasse` keinen besonderen Typ (etwa `int` oder `char`) aufweist. `#define` nimmt eine einfache Textersetzung vor. Der Präprozessor schreibt an alle Stellen, wo `StudentenProKlasse` vorkommt, die Zeichenfolge `15` in den Quelltext.

Da der Präprozessor vor dem Compiler ausgeführt wird, kommt Ihr Compiler niemals mit der symbolischen Konstanten in Berührung, sondern erhält immer die Zahl `15` übergeben.

Konstanten mit `const` definieren

Obwohl `#define` funktioniert, gibt es in C++ eine neue, bessere und elegantere Lösung zur Definition von Konstanten:

```
const unsigned short int StudentenProKlasse = 15;
```

Dieses Beispiel deklariert ebenfalls eine symbolische Konstante namens `StudentenProKlasse`, dieses Mal ist aber `StudentenProKlasse` als Typ `unsigned short int` definiert.

Diese Version erfordert zwar mehr Tipparbeit, bietet aber verschiedene Vorteile. Der größte Unterschied ist der, dass diese Konstante einen Typ hat und der Compiler die zweckmäßige – sprich typgerechte – Verwendung der Konstanten prüfen kann.

3.8 Aufzählungskonstanten

Aufzählungskonstanten (`enum`) erzeugen einen Satz von Konstanten mit einem Bereich von Werten. Beispielsweise kann man `FARBE` als Aufzählung deklarieren und dafür fünf Werte definieren: `ROT`, `BLAU`, `GRUEN`, `WEISS` und `SCHWARZ`.

Die Syntax für Aufzählungskonstanten besteht aus dem Schlüsselwort `enum`, gefolgt vom Typennamen, einer öffnenden geschweiften Klammer, allen zulässigen – durch Kommas getrennten – Werten, einer schließenden geschweiften Klammer und einem Semikolon. Dazu ein Beispiel:

```
enum FARBE { ROT, BLAU, GRUEN, WEISS, SCHWARZ };
```

Diese Anweisung realisiert zwei Aufgaben:

1. FARBE wird zum Namen einer Aufzählung, das heißt, zu einem neuen Typ.
2. ROT wird zu einer symbolischen Konstanten mit dem Wert 0, BLAU zu einer symbolischen Konstanten mit dem Wert 1, GRUEN zu einer symbolischen Konstanten mit dem Wert 2 usw.

Jeder Aufzählungskonstanten ist ein Integer-Wert zugeordnet. Wenn man nichts anderes festlegt, weist der Compiler der ersten Konstanten den Wert 0 zu und nummeriert die restlichen Konstanten fortlaufend. Jede einzelne Konstante lässt sich mit einem bestimmten Wert initialisieren, wobei die nicht initialisierten Konstanten aufsteigende Werte zu den jeweils vorhergehenden erhalten. Schreibt man daher

```
enum FARBE { ROT=100, BLAU, GRUEN=500, WEISS, SCHWARZ=700 };
```

erhält ROT den Wert 100, BLAU den Wert 101, GRUEN den Wert 500, WEISS den Wert 501 und SCHWARZ den Wert 700.

Aufzählungen haben den Vorteil, dass man mit symbolischen Namen wie SCHWARZ oder WEISS anstelle nichts sagender Zahlen wie 1 oder 700 arbeiten kann.

Es ist üblich, normale und Aufzählungskonstanten durchgehend groß zu schreiben. Dadurch ist auf einen Blick erkennbar, dass es sich um eine Konstante handelt und nicht um eine Variable, deren Wert man ändern darf. Dem Compiler ist die Schreibweise egal, solange man die Namen jeweils einheitlich schreibt.

3.9 Fragen und Antworten

F: Wenn der Wertebereich einer Variablen vom Typ `short int` eventuell nicht ausreicht, warum verwendet man dann nicht immer Ganzzahlen vom Typ `long`?

A: Sowohl bei ganzen Zahlen vom Typ `short` als auch bei `long` können Platzprobleme auftreten, auch wenn der Wertebereich bei Zahlen vom Typ `long` wesentlich größer ist. Auf den meisten Maschinen nimmt aber eine Variable vom Typ `long` doppelt soviel Speicher ein wie eine Variable vom Typ `short`. Allerdings stellt das heutzutage kaum noch ein Problem dar, da die meisten PCs über mehrere Megabyte Hauptspeicher verfügen.

F: Was passiert, wenn ich eine reelle Zahl an eine Variable vom Typ `int` statt vom Typ `float` zuweise? Als Beispiel dazu die folgende Codezeile:

```
int aNumber = 5.4;
```

A: Ein guter Compiler erzeugt eine Warnung, aber die Zuweisung ist durchaus zulässig. Die zugewiesene Zahl wird auf eine ganze Zahl abgeschnitten. Wenn man daher `5.4` an eine ganzzahlige Variable zuweist, erhält diese Variable den Wert `5`. Allerdings gehen Informationen verloren und wenn man später den Wert der Integer-Variablen an eine Variable vom Typ `float` zuweist, erhält die `float`-Variable ebenfalls nur den Wert `5`.

F: Warum sollte man auf literale Konstanten verzichten und sich die Mühe machen, symbolische Konstanten zu verwenden?

A: Wenn man einen Wert an vielen Stellen im gesamten Programm hindurch verwendet, kann man bei einer symbolischen Konstanten alle Werte ändern, indem man einfach die nur einmal vorhandene Definition der Konstanten ändert. Symbolische Konstanten sprechen auch für sich selbst. Es kann schwer zu verstehen sein, warum eine Zahl mit 360 multipliziert wird. Einfacher ist dagegen eine Anweisung zu lesen, in der die Multiplikation mit `GradImVollkreis` realisiert wird.

3.10 Workshop

Nachdem Sie sich mit Variablen und Konstanten beschäftigt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zum Compiler zu festigen.

3.10.1 Kontrollfragen

1. Warum sollte man `unsigned`-Ganzzahlen den `signed`-Ganzzahlen vorziehen?
2. Worin besteht der Unterschied zwischen der Initialisierung und der Definition einer Variablen?
3. Sind die Variablen `DOG`, `dog`, `Dog` und `doG` gleich?
4. Worin besteht der Unterschied zwischen einer mit `#define` und einer mit `const` definierten Konstante?

3.10.2 Übungen

1. Erweitern Sie das Programm von Listing 3.1, um alle in Tabelle 3.1 aufgeführten Variablentypen einzubinden. Die Werte müssen nicht unbedingt den angegebenen entsprechen.

2. Suchen Sie sich einige Zahlen und Informationen, mit denen Sie im täglichen Leben zu tun haben. Stellen Sie sich vor, welche Variablentypen am besten geeignet sind, um die Werte zu speichern. Denken Sie sich aussagekräftige Variablennamen aus.
3. Falls Sie einen professionellen Softwareentwickler kennen, fragen Sie ihn doch mal nach seinem bevorzugten Benennungsformat. Und sollten Sie mehrere Entwickler kennen, werden Sie feststellen, dass jeder einen anderen Ansatz verfolgt. Erkundigen Sie sich nach den Gründen für das eine oder andere Format. Die Antworten werden Sie überraschen.

3.10.3 Antworten zu den Kontrollfragen

1. Vorzeichenlose (`unsigned`) Ganzzahlen erlauben mehr positive Werte als vorzeichenbehaftete (`signed`) Typen. Außerdem nehmen sie keine negativen Werte an. Es liegt bei Ihnen als Programmierer zu bestimmen, welche Typen für ein Problem am besten geeignet sind.
2. Mit einer Definition deklarieren Sie den Datentyp und den Namen einer Variablen. Die Initialisierung weist der Variablen den ersten (oder anfänglichen) Wert zu. In ein und demselben Befehl kann man sowohl die Definition als auch die Initialisierung vornehmen.
3. Die Variablen sind nicht gleich. C++ berücksichtigt die Groß-/Kleinschreibung und für den Compiler sehen die angegebenen Variablen vollkommen unterschiedlich aus. Welches Namensformat Sie wählen, hängt von persönlichen Vorlieben oder von Team-Richtlinien ab.
4. Die Präprozessor-Direktive `#define` setzt den angegebenen Wert (die literale Konstante) an allen Stellen im Code ein, an denen der Wert vorkommt. Eine als `const` deklarierte Variable belegt nur einen einzigen Speicherplatz und hat den spezifizierten Datentyp.

KAPITEL 4

jetzt lerne ich

Ausdrücke und Anweisungen

In dieser Stunde lernen Sie,

- was Anweisungen sind,
- was Ausdrücke sind,
- wie man mit Operatoren arbeitet,
- was Wahrheitswerte sind.

4.1 Anweisungen

Eine *Anweisung* steuert die Reihenfolge der Ausführung, wertet einen Ausdruck aus oder bewirkt nichts (die `null`-Anweisung). Alle C++-Anweisungen enden mit einem Semikolon.

Eine häufig gebrauchte einfache Anweisung ist die Zuweisung:

```
x = a + b;
```

Diese Anweisung bedeutet im Gegensatz zur Algebra nicht `x` gleich `a + b`, sondern ist wie folgt zu interpretieren: »Weise den Wert der Summe aus `a` und `b` an `x` zu.« Auch wenn diese Anweisung zwei Dinge bewirkt, handelt es sich um eine einzige Anweisung und ist daher mit einem Semikolon abzuschließen. Der Zuweisungsoperator nimmt die Zuweisung des auf der rechten Seite stehenden Ausdrucks an die linke Seite vor.

4.1.1 Whitespaces

Leerzeichen gehören zusammen mit Tabulatoren und den Zeilenschaltungen zu den so genannten *Whitespaces*. Derartige Zeichen ignoriert der Compiler. An jeder Stelle, wo ein einzelnes Leerzeichen zu sehen ist, kann man auch einen Tabulator oder einen Zeilenvorschub schreiben. Whitespaces fügt man ein, damit sich das Programm leichter lesen lässt. Der Compiler nimmt keine Notiz davon.

Die oben behandelte Zuweisung lässt sich auch wie folgt schreiben:

```
x=a+b;
```

oder

```
x                          =a
+          b          ;
```

Die zweite Variante ist zwar zulässig, aber kompletter Blödsinn. Durch Whitespaces sollen Programme leichter zu lesen und zu warten sein. Man kann damit aber auch unleserlichen Code produzieren. C++ stellt die Möglichkeiten bereit, für den sinnvollen Einsatz ist der Programmierer verantwortlich. Wichtig ist, dass Whitespaces innerhalb von Variablennamen, Funktionsnamen oder Zeichenfolgen signifikant sind. So wird die Variable `sekundenProMinute` zu etwas völlig anderem, wenn man `sekunden Pro Minute` schreibt (der Compiler fasst das als drei verschiedene Variablen auf).

4.1.2 Verbundanweisungen

An allen Stellen, wo eine einzelne Anweisung stehen kann, sind auch Verbundanweisungen (zusammengesetzte Anweisungen) zulässig. Eine *Verbundanweisung* beginnt mit einer öffnenden geschweiften Klammer ({) und endet mit einer schließenden geschweiften Klammer (}).

In einer Verbundanweisung ist zwar jede Anweisung mit einem Semikolon abzuschließen, die Verbundanweisung selbst endet aber nicht mit einem Semikolon. Dazu ein Beispiel:

```
{
     temp = a;
     a = b;
     b = temp;
}
```

Diese Verbundanweisung tauscht die Werte in den Variablen `a` und `b` aus.

Was Sie tun sollten

Schreiben Sie immer eine schließende geschweifte Klammer, wenn eine öffnende geschweifte Klammer vorhanden ist.

Schließen Sie Anweisungen mit einem Semikolon ab.

Setzen Sie Whitespaces sinnvoll ein, um den Code deutlicher zu präsentieren.

Machen Sie Verbundanweisungen nicht länger als eine oder zwei Bildschirmseiten, um den Überblick nicht zu verlieren.

4.2 Ausdrücke

Alles, was einen Wert zurückgibt, nennt man in C++ einen *Ausdruck*.

Ganz einfach also. Wenn etwas einen Wert liefert, ist es ein Ausdruck. Alle Ausdrücke sind Anweisungen.

Die Vielfalt der Codeabschnitte, die sich als Ausdruck entpuppen, mag Sie vielleicht überraschen. Hier drei Beispiele:

```
3.2                     // liefert den Wert 3.2
PI                      // Gleitpunktkonstante, die den Wert 3.14
zurückgibt
SekundenProMinute       // int-Konstante, die 60 liefert
```

Vorausgesetzt, dass `PI` eine Konstante mit dem Wert `3.14` und `SekundenProMinute` eine Konstante mit dem Wert `60` ist, stellen alle drei Anweisungen gültige Ausdrücke dar.

Der komplexere Ausdruck

```
x = a + b;
```

addiert nicht nur `a` und `b` und weist das Ergebnis an `x` zu, sondern liefert auch den Wert dieser Zuweisung (den Wert in `x`). Daher ist diese Anweisung ebenfalls ein Ausdruck und kann somit auch auf der rechten Seite eines Zuweisungsoperators stehen:

```
y = x = a + b;
```

Der *Zuweisungsoperator* (`=`) bewirkt, dass der Wert des Operanden auf der linken Seite des Operators in den Wert auf der rechten Seite des Operators geändert wird.

Operand ist ein mathematischer Begriff, der den Teil eines Ausdrucks bezeichnet, auf den ein Operator wirkt.

Die obige Zeile wird in der folgenden Reihenfolge ausgewertet:

1. Addiere `a` zu `b`.
2. Weise das Ergebnis des Ausdrucks `a + b` an `x` zu.
3. Weise das Ergebnis des Zuweisungsausdrucks `x = a + b` an `y` zu.

Wenn `a`, `b`, `x` und `y` ganze Zahlen sind, `a` den Wert `2` und `b` den Wert `5` hat, enthält sowohl `x` als auch `y` nach Ausführung dieser Anweisung den Wert `7`.

Listing 4.1 demonstriert die Auswertung komplexer Ausdrücke.

Listing 4.1: express.cpp – Auswertung komplexer Ausdrücke

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:     int a=0, b=0, x=0, y=35;
 6:     std::cout << "vorher a: " << a << " b: " << b;
 7:     std::cout << " x: " << x << " y: " << y << std::endl;
 8:     a = 9;
 9:     b = 7;
10:     y = x = a+b;
11:     std::cout << "nachher a: " << a << " b: " << b;
12:     std::cout << " x: " << x << " y: " << y << std::endl;
13:     return 0;
14: }
```

```
vorher a: 0 b: 0 x: 0 y: 35
nachher a: 9 b: 7 x: 16 y: 16
```

Zeile 5 deklariert und initialisiert die vier Variablen. Die Ausgabe ihrer Werte erfolgt in den Zeilen 6 und 7. Zeile 8 weist den Wert `9` an die Variable `a`, Zeile 9 den Wert `7` an die Variable `b` zu. Zeile 10 summiert die Werte von `a` und `b` und weist das Ergebnis an `x` zu. Dieser Ausdruck (`x = a+b`) ergibt einen Wert (die Summe aus `a` und `b`), der wiederum an `y` zugewiesen wird.

4.3 Operatoren

Ein *Operator* ist ein Symbol, das den Compiler zur Ausführung einer Aktion veranlasst.

4.3.1 Zuweisungsoperator

Einen Operanden, der auf der linken Seite eines Zuweisungsoperators zulässig ist, bezeichnet man als L-Wert (linker Wert). Ein dementsprechender Operand auf der rechten Seite heißt R-Wert.

Konstanten und R-Werte

Konstanten sind R-Werte und können nicht als L-Werte vorkommen. Demzufolge ist die Anweisung

```
x = 35;     // OK
```

zulässig, während die Anweisung

```
35 = x;     // Fehler, kein L-Wert!
```

nicht erlaubt ist.

Ein *L-Wert* (englisch L-Value) ist ein Operand, der auf der linken Seite eines Ausdrucks stehen kann. Als *R-Wert* (englisch R-Value) bezeichnet man einen Operanden, der auf der rechten Seite eines Ausdrucks vorkommen kann. Während alle L-Werte auch als R-Werte zulässig sind, dürfen nicht alle R-Werte auch als L-Werte verwendet werden. Ein Literal ist zum Beispiel ein R-Wert, der nicht als L-Wert erlaubt ist. Demzufolge kann man `x = 5;` schreiben, `5 = x;` jedoch nicht.

Warum muss ich mich mit L-Wert und R-Wert befassen?

Vielleicht fragen Sie sich, warum Sie sich mit einer so seltsamen Terminologie herumschlagen müssen. Diese Begriffe finden Sie in Fehlermeldungen wieder und Sie müssen verstehen, warum sich der Compiler beschwert. Zudem können Sie erkennen, wieso bestimmte Codekonstruktionen zulässig sind und andere nicht.

4.3.2 Mathematische Operatoren

C++ kennt die fünf mathematischen Operatoren für Addition (+), Subtraktion (-), Multiplikation (*), Division (/) und Modulo-Operation (%). Wie C gehört auch C++ zu den wenigen Sprachen, die keinen vordefinierten Exponential-

operator (x zur Potenz von y erheben) haben. Allerdings gibt es für diese Aufgabe eine entsprechende Funktion.

Die drei Grundrechenarten Addition, Subtraktion und Multiplikation arbeiten wie gewohnt. Bei der Division gibt es dagegen ein paar Besonderheiten zu beachten.

Die Integer-Division unterscheidet sich etwas von der gewohnten Division. Dividiert man 21 durch 4, ist das Ergebnis eine reelle Zahl (eine Zahl mit gebrochenem Anteil). Ganze Zahlen (Integer) haben keinen gebrochenen Anteil, sodass der »Rest« abgeschnitten wird. Der aus `21 / 4` zurückgegebene Wert ist also `5`.

Der Modulo-Operator (`%`) gibt den Rest einer Ganzzahldivision zurück. Der Ausdruck `21 % 4` liefert `1`, da `21 / 4` gleich `5` mit Rest `1` ist.

Die Modulo-Anweisung lesen

Der Ausdruck `21 % 4` ist als »21 modulo 4« zu lesen. Das Ergebnis der Modulo-Operation heißt auch Modulus. Somit ist der Rest `1` des angegebenen Ausdrucks der Modulus.

In der Praxis setzt man Modulo-Berechnungen zum Beispiel ein, um eine Statusmeldung bei jedem zehnten Schleifendurchlauf auszugeben.

Eine Zahl modulo `10` liefert `0`, wenn die Zahl ein Vielfaches von 10 ist. Daher ist `20 % 10` wie auch `30 % 10` usw. gleich `0`.

Die Gleitpunktdivision läuft genauso ab, wie man sie beispielsweise vom Taschenrechner gewohnt ist: `21 / 4.0` liefert das Ergebnis `5.25`. Wenn mindestens ein Operand ein Gleitpunktwert ist (Literal oder Variable), ist auch das Ergebnis ein Gleitpunktwert.

4.4 Zusammengesetzte Operatoren

Häufig muss man einen Wert zu einer Variablen addieren und dann das Ergebnis an dieselbe Variable zuweisen. Im folgenden Beispiel wird der Wert der Variablen `myAge` um 2 erhöht:

```
int myAge = 5;
int temp;
temp = myAge + 2;  // 5 und 2 addieren und in temp ablegen
myAge = temp;      // nach myAge zurückschreiben
```

Dieses Verfahren ist allerdings recht umständlich. In C++ kann man dieselbe Variable auf beiden Seiten des Zuweisungsoperators einsetzen und das obige Codefragment eleganter formulieren:

```
myAge = myAge + 2;
```

In der Algebra wäre dieser Ausdruck unzulässig, während ihn C++ als »addiere 2 zum Wert in `myAge` und weise das Ergebnis an `myAge` zu« interpretiert.

Das Ganze lässt sich noch einfacher schreiben, auch wenn es im ersten Moment etwas unverständlich aussieht:

```
myAge += 2;
```

Der zusammengesetzte Additionsoperator (`+=`) addiert den R-Wert zum L-Wert und führt dann eine erneute Zuweisung des Ergebnisses an den L-Wert durch. Der Operator heißt »Plus-Gleich«. Die Anweisung ist dann als »`myAge` plus gleich 2« zu lesen. Wenn `myAge` zu Beginn den Wert `4` enthält, steht nach Ausführung dieser Anweisung der Wert `6` in `myAge`.

Weitere zusammengesetzte Operatoren gibt es für Subtraktion (`-=`), Division (`/=`), Multiplikation (`*=`) und Modulo-Operation (`%=`).

4.5 Inkrementieren und Dekrementieren

Am häufigsten hat man den Wert `1` zu einer Variablen zu addieren (bzw. von einer Variablen zu subtrahieren). In C++ spricht man beim Erhöhen des Wertes um 1 von *Inkrementieren* und beim Verringern um 1 von *Dekrementieren*. Für diese Operationen stehen spezielle Operatoren bereit.

Der Inkrement-Operator (`++`) erhöht den Wert der Variablen um 1, der Dekrement-Operator (`--`) verringert ihn um 1. Möchte man die Variable C inkrementieren, schreibt man die folgende Anweisung:

```
C++;      // Beginne mit C und inkrementiere den enthaltenen Wert.
```

Diese Anweisung ist äquivalent zur ausführlicher geschriebenen Anweisung

```
C = C + 1;
```

Das gleiche Ergebnis liefert die verkürzte Darstellung

```
C += 1;
```

Warum die Sprache C++ heißt

Sehen Sie, warum die Sprache »C++« heißt? Sie wurde als inkrementelle Verbesserung gegenüber der Sprache C und nicht als vollkommen neue Sprache entwickelt.

4.5.1 Präfix und Postfix

Sowohl der Inkrement-Operator (++) als auch der Dekrement-Operator (--) existiert in zwei Spielarten: *Präfix* und *Postfix*. Die Präfix-Version wird vor den Variablennamen geschrieben (`++myAge`), die Postfix-Version danach (`myAge++`).

In einer einfachen Anweisung spielt es keine Rolle, welche Version man verwendet. In einem komplexen Ausdruck ist es allerdings entscheidend, ob man eine Variable zuerst inkrementiert (oder dekrementiert) und dann das Ergebnis einer anderen Variablen zuweist. Der Präfix-Operator wird vor der Zuweisung ausgewertet, der Postfix-Operator nach der Zuweisung.

Die Semantik von Präfix bedeutet: inkrementiere den Wert und rufe ihn dann ab. Die Bedeutung des Postfix-Operators lautet dagegen: rufe den Wert ab und inkrementiere dann das Original.

Das folgende Beispiel verdeutlicht diese Vorgänge. Es sei angenommen, dass `x` eine ganze Zahl mit dem Wert `5` ist. Bei der Anweisung

```
int a = ++x;
```

inkrementiert der Compiler den Wert in `x` (und macht ihn damit zu 6), holt dann diesen Wert und weist ihn der Variablen `a` zu. Daher ist jetzt sowohl `a` als auch `x` gleich `6`.

Schreibt man anschließend

```
int b = x++;
```

weist man den Compiler an, den Wert aus `x` (6) zu holen, ihn der Variablen `b` zuzuweisen und dann den Wert von `x` zu inkrementieren. Demzufolge ist nun `b` gleich 6, während `x` gleich 7 ist. Listing 4.2 zeigt Verwendung und Besonderheiten beider Typen.

Listing 4.2: prepostfix.cpp – Präfix- und Postfix-Operatoren

```
 1: // Listing 4.2 - Zeigt die Verwendung der
 2: // Inkrement- und Dekrement-Operatoren in
 3: // Präfix- und Postfix-Notation
 4: #include <iostream>
 5:
 6: int main()
 7: {
 8:     int myAge = 39;      // initialisiert zwei Integer-Variablen
 9:     int yourAge = 39;
10:     std::cout << "Ich bin:\t" << myAge << "\tJahre alt.\n";
11:     std::cout << "Du bist:\t" << yourAge << "\tJahre alt.\n";
12:     myAge++;         // Postfix-Inkrement
13:     ++yourAge;       // Präfix-Inkrement
14:     std::cout << "Ein Jahr ist vergangen...\n";
```

```
15:     std::cout << "Ich bin:\t" << myAge << "\tJahre alt.\n";
16:     std::cout << "Du bist:\t" << yourAge << "\tJahre alt.\n";
17:     std::cout << "Noch ein Jahr ist vergangen.\n";
18:     std::cout << "Ich bin:\t" << myAge++ << "\tJahre alt.\n";
19:     std::cout << "Du bist:\t" << ++yourAge << "\tJahre alt.\n";
20:     std::cout << "Und noch einmal ausgeben.\n";
21:     std::cout << "Ich bin:\t" << myAge << "\tJahre alt.\n";
22:     std::cout << "Du bist:\t" << yourAge << "\tJahre alt.\n";
23:     return 0;
24: }
```

```
Ich bin:        39      Jahre alt.
Du bist:        39      Jahre alt.
Ein Jahr ist vergangen...
Ich bin:        40      Jahre alt.
Du bist:        40      Jahre alt.
Noch ein Jahr ist vergangen.
Ich bin:        40      Jahre alt.
Du bist:        41      Jahre alt.
Und noch einmal ausgeben.
Ich bin:        41      Jahre alt.
Du bist:        41      Jahre alt.
```

Die Zeilen 8 und 9 deklarieren zwei Integer-Variablen und initialisieren sie jeweils mit dem Wert 39. Die Ausgabe der Werte findet in den Zeilen 10 und 11 statt.

Zeile 12 inkrementiert `myAge` mit dem Postfix-Operator, und Zeile 13 inkrementiert `yourAge` mit dem Präfix-Operator. Die Ergebnisse werden in den Zeilen 15 und 16 ausgegeben – sie sind identisch (beide 40).

In Zeile 18 wird `myAge` mit dem Postfix-Operator als Teil einer Ausgabeanweisung inkrementiert. Durch den Postfix-Operator erfolgt die Inkrementierung nach der Ausgabe und es erscheint auch hier in der Ausgabe der Wert 40. Im Gegensatz dazu inkrementiert Zeile 18 die Variable `yourAge` mit dem Präfix-Operator. Das Inkrementieren findet jetzt vor der Ausgabe statt und es erscheint der Wert 41 in der Anzeige.

Schließlich werden die Werte in den Zeilen 21 und 22 erneut ausgegeben. Da die Inkrement-Anweisung vollständig abgearbeitet ist, lautet der Wert in `myAge` jetzt 41, genau wie der Wert in `yourAge`.

Drei Arten das Gleiche zu tun. Welche soll ich wählen?

In C++ gibt es drei Arten, den Wert 1 zu einer Variablen zu addieren: `a=a+1`, `a+=1` und `a++`. Es erhebt sich nun die Frage, welche Version man wählen sollte.

Es war einmal so, dass durch die zugrunde liegende Hardware-Architektur der CPU `A+=1` effizienter als `A=A+1` und `A++` effizienter als `A+=1` gewesen ist. Möglicherweise gilt das auch heute noch, doch optimierende Compiler können jeweils den effizientesten Maschinencode erzeugen.

Persönlich ziehe ich den Operator ++ vor, wenn auf den ersten Blick klar sein soll, dass ich 1 addiere. Um irgendeinen anderen Wert (2, 3 oder eine Variable) zu addieren, greife ich auf += zurück, und beim Addieren mehrerer Variablen verwende ich die regulären Operatoren = und +.

Die unterschiedlichen Codeformate machen für mich die Bedeutung klarer.

4.6 Rangfolge: die Reihenfolge der Operationen

Unter der *Rangfolge* oder dem *Vorrang* versteht man die Reihenfolge, nach der die Operatoren abgearbeitet werden.

Welche Operation wird in der komplexen Anweisung

```
x = 5 + 3 * 8;
```

zuerst ausgeführt, die Addition oder die Multiplikation? Führt man die Addition zuerst aus, lautet das Ergebnis 8 * 8 oder 64. Bei vorrangiger Multiplikation heißt das Ergebnis 5 + 24 oder 29.

Jeder Operator besitzt einen festgelegten *Vorrang*. Die vollständige Liste finden Sie auf dem Einlageblatt. Die Multiplikation hat gegenüber der Addition einen höheren Vorrang. Der Wert des Ausdrucks ist demnach 29.

Wenn zwei mathematische Operatoren den gleichen Vorrang haben, werden sie in der Reihenfolge von links nach rechts ausgeführt. Demzufolge werden in

```
x = 5 + 3 + 8 * 9 + 6 * 4;
```

die Multiplikationen zuerst – und zwar von links nach rechts – ausgeführt. Es ergeben sich die beiden Terme zu 8*9 = 72 und 6*4 = 24. Damit wird der Ausdruck zu

```
x = 5 + 3 + 72 + 24;
```

Daran schließen sich die Additionen von links nach rechts an: 5 + 3 = 8, 8 + 72 = 80, 80 + 24 = 104.

Die Rangfolge ist unbedingt zu beachten. Bestimmte Operatoren wie der Zuweisungsoperator werden von rechts nach links ausgeführt! Was passiert nun, wenn die Rangfolge nicht Ihren Vorstellungen entspricht? Sehen Sie sich dazu folgenden Ausdruck an:

```
TotalSeconds = NumMinutesToThink + NumMinutesToType * 60
```

In diesem Ausdruck soll nicht `NumMinutesToType` zuerst mit 60 multipliziert und dann zu `NumMinutesToThink` addiert werden. Beabsichtigt ist zuerst die Addition der beiden Variablen, um die Summe der Minuten zu ermitteln, und anschließend soll diese Zahl mit 60 multipliziert werden, um die Anzahl der Sekunden zu berechnen.

In diesem Fall setzt man Klammern, um die Rangfolge zu ändern. Elemente in Klammern werden mit höherem Vorrang ausgeführt als irgendein mathematischer Operator. Das gewünschte Ergebnis erhält man also mit

```
TotalSeconds = (NumMinutesToThink + NumMinutesToType) * 60
```

Klammern verwenden

Verwenden Sie im Zweifelsfall Klammern, um Ihre Absichten deutlich zu machen. Das Programm wird dadurch nicht langsamer, nur der Compiler braucht eine winzige Spur länger, um das Programm zu kompilieren.

4.7 Verschachtelte Klammern

Bei komplexen Ausdrücken müssen Sie gegebenenfalls Klammern verschachteln, d.h. ein Klammernpaar innerhalb eines anderen unterbringen. Beispielsweise ist zuerst die Anzahl der Sekunden zu berechnen und danach die Anzahl der Mitarbeiter, bevor die Multiplikation der Mitarbeiter mit den Sekunden erfolgt (um etwa die gesamte Arbeitszeit in Sekunden zu erhalten):

```
TotalPersonSeconds = ( ( (NumMinutesToThink + NumMinutesToType) * 60)
* (PeopleInTheOffice + PeopleOnVacation) )
```

Dieser zusammengesetzte Ausdruck ist von innen nach außen zu lesen. Zuerst erfolgt die Addition von `NumMinutesToThink` zu `NumMinutesToType`, da dieser Ausdruck in den innersten Klammern steht. Anschließend wird diese Summe mit 60 multipliziert. Es schließt sich die Addition von `PeopleInTheOffice` zu `PeopleOnVacation` an. Schließlich wird die berechnete Mitarbeiterzahl mit der Gesamtzahl der Sekunden multipliziert.

Dieses Beispiel weist auf einen wichtigen Punkt hin. Für einen Computer ist der Ausdruck leicht zu interpretieren, für einen Menschen aber schwer zu lesen, zu verstehen oder zu modifizieren. Der gleiche Ausdruck in einer anderen Form mit Variablen zur Zwischenspeicherung sieht folgendermaßen aus:

```
TotalMinutes = NumMinutesToThink + NumMinutesToType;
TotalSeconds = TotalMinutes * 60;
TotalPeople = PeopleInTheOffice + PeopleOnVacation;
TotalPersonSeconds = TotalPeople * TotalSeconds;
```

Dieses Beispiel verwendet zwar temporäre Variablen und erfordert mehr Schreibarbeit als das vorherige Beispiel, ist aber leichter zu verstehen. Fügen Sie am Beginn einen Kommentar ein, um die Absichten hinter diesem Codeabschnitt zu erläutern, und ändern Sie die `60` in eine symbolische Konstante. Damit erhalten Sie Code, der leicht zu verstehen und zu warten ist.

Wahrheitswerte

Frühere Versionen von C++ haben die Wahrheitswerte durch ganze Zahlen dargestellt. Mit dem neuen ISO/ANSI-Standard ist der Datentyp `bool` hinzugekommen. Dieser Typ hat zwei mögliche Werte: falsch (`false`) und wahr (`true`).

Jeder Ausdruck lässt sich als Wahrheitswert auswerten. Ausdrücke, die im mathematischen Sinne 0 sind, liefern den Wahrheitswert `false`, alle anderen Ergebnisse den Wahrheitswert `true`.

Ein Teil der bisher auf dem Markt verfügbaren Compiler kannte bereits einen booleschen Datentyp, der intern durch `long int` mit einer Größe von 4 Bytes dargestellt wurde. ANSI-kompatible Compiler bieten jetzt den Datentyp `bool` mit einer Länge von 1 Byte.

4.8 Vergleichsoperatoren

Mit den *Vergleichsoperatoren* (oder *relationalen Operatoren*) ermittelt man, ob zwei Zahlen gleich sind oder ob eine größer oder kleiner als die andere ist. Jeder Vergleichsausdruck liefert entweder 1 (`true`) oder 0 (`false`) zurück. Die Vergleichsoperatoren fasst Tabelle 4.1 zusammen.

Hat die Integer-Variable `myAge` den Wert `39` und die Integer-Variable `yourAge` den Wert `40`, prüft man diese Werte mit dem relationalen Operator »Gleich« auf Gleichheit:

```
myAge == yourAge;  // ist der Wert in myAge der gleiche wie in
yourAge?
```

Dieser Ausdruck ergibt 0 oder `false`, da die Variablen nicht gleich sind. Der Ausdruck

```
myAge > yourAge;  // ist myAge größer als yourAge?
```

liefert 0 oder `false`.

Zuweisung ist keine Gleichheit

Viele Neueinsteiger in die C++-Programmierung verwechseln den Zuweisungsoperator (=) mit dem Gleich-Operator (==). Das kann zu gemeinen Fehlern im Programm führen.

Man darf das Ergebnis eines Vergleichs einer Variablen zuweisen! Es ist absolut zulässig, `a = b > c` oder `a = b == c` zu kodieren. Der Compiler gibt möglicherweise eine Warnung aus, wenn Sie den Zuweisungsoperator an einer Stelle verwenden, an der der Gleich-Operator sinnvoller erscheint – der Compiler ist sozusagen nicht sicher, ob Sie wirklich das beabsichtigen, was Sie kodiert haben.

Zu den Vergleichsoperatoren gehören: Gleich (==), Kleiner als (<), Größer als (>), Kleiner oder gleich (<=), Größer oder gleich (>=) und Ungleich (!=). Tabelle 4.1 zeigt die Vergleichsoperatoren, ein Codebeispiel und den Rückgabewert.

Tabelle 4.1: Die Vergleichsoperatoren

Operator	Symbol	Beispiel	Rückgabewert
Gleich	==	`100 == 50;`	`false`
		`50 == 50;`	`true`
Ungleich	!=	`100 != 50;`	`true`
		`50 != 50`	`false`
Größer als	>	`100 > 50;`	`true`
		`50 > 50;`	`false`
Größer oder gleich	>=	`100 >= 50;`	`true`
		`50 >= 50`	`true`
Kleiner als	<	`100 < 50;`	`false`
		`50 < 50;`	`false`
Kleiner oder gleich	<=	`100 <= 50;`	`false`
		`50 <= 50;`	`true`

4.9 Die if-Anweisung

Normalerweise verläuft der Programmfluss zeilenweise in der Reihenfolge, in der die Anweisungen in Ihrem Quellcode stehen. Mit der `if`-Anweisung kann man auf eine Bedingung testen (beispielsweise ob zwei Variablen einander gleich sind) und zu unterschiedlichen Teilen des Codes in Abhängigkeit vom Testergebnis verzweigen.

Die einfachste Form der `if`-Anweisung sieht folgendermaßen aus:

```
if (Ausdruck)
    Anweisung;
```

Der Ausdruck in den Klammern kann jeder beliebige Ausdruck sein, er enthält aber in der Regel einen der Vergleichsausdrücke. Wenn der Ausdruck den Wert `0` liefert, entspricht das dem booleschen Wert `false`, und die Anweisung wird übersprungen. Ergibt sich ein Wert ungleich 0, nimmt C++ den booleschen Wert `true` an, und die Anweisung wird ausgeführt. Sehen Sie sich dazu folgendes Beispiel an:

```
if (bigNumber > smallNumber)
    bigNumber = smallNumber;
```

Dieser Code vergleicht `bigNumber` mit `smallNumber`. Wenn `bigNumber` größer ist, setzt die zweite Zeile den Wert von `bigNumber` auf den Wert von `smallNumber`.

4.9.1 Die `else`-Klausel

Oftmals soll ein Programm bei erfüllter Bedingung (`true`) den einen Zweig durchlaufen und bei nicht erfüllter Bedingung (`false`) einen anderen.

Die bisher gezeigte Form, bei der man zuerst die eine Bedingung testet und dann die andere, funktioniert zwar, ist aber etwas umständlich. Das Schlüsselwort `else` trägt hier zur besseren Lesbarkeit des Codes bei:

```
if (Ausdruck)
    Anweisung;
else
    Anweisung;
```

Listing 4.3 demonstriert die Verwendung des Schlüsselwortes `else`.

Listing 4.3: ifelse.cpp – Einsatz der else-Klausel

```
1: // Listing 4.3 - Zeigt die if-Anweisung mit der else-Klausel
2: #include <iostream>
3:
4: int main()
5: {
6:     int firstNumber, secondNumber;
7:     std::cout << "Bitte eine grosse Zahl eingeben: ";
```

```
 8:     std::cin >> firstNumber;
 9:     std::cout << "\nBitte eine kleinere Zahl eingeben: ";
10:     std::cin >> secondNumber;
11:     if (firstNumber > secondNumber)
12:         std::cout << "\nDanke!\n";
13:     else
14:         std::cout << "\nDie zweite Zahl ist groesser!";
15:
16:     return 0;
17: }
```

```
Bitte eine grosse Zahl eingeben: 10
Bitte eine kleinere Zahl eingeben: 12
Die zweite Zahl ist groesser!
```

Mit anderen Zahlen erhalten Sie beispielsweise folgende Ergebnisse:

```
Bitte eine grosse Zahl eingeben: 12
Bitte eine kleinere Zahl eingeben: 10
Danke!
```

Liefert die Bedingung der `if`-Anweisung in Zeile 11 das Ergebnis `true`, wird die Anweisung in Zeile 12 ausgeführt. Ergibt sich der Wert `false`, führt das Programm die Anweisung in Zeile 14 aus. Wenn man die `else`-Klausel in Zeile 13 entfernt, wird die Anweisung in Zeile 14 unabhängig davon ausgeführt, ob die `if`-Anweisung `true` ist oder nicht. Denken Sie daran, dass die `if`-Anweisung nach Zeile 12 endet. Fehlt die `else`-Klausel, wäre Zeile 14 einfach die nächste Zeile im Programm.

Für die einzelnen Anweisungen kann man auch Codeblöcke schreiben, die in geschweifte Klammern eingeschlossen sind.

In C++ lassen sich mit der Anweisung `std::cin` Daten in eine Variable lesen.

Wenn der `Ausdruck` in

```
if (Ausdruck)
    Anweisung;
nächste_Anweisung;
```

das Ergebnis `true` liefert, wird die `Anweisung` ausgeführt und das Programm setzt mit `nächste_Anweisung` fort. Ergibt der `Ausdruck` nicht `true`, wird `Anweisung` ignoriert und das Programm springt sofort zu `nächste_Anweisung`.

Als Anweisung kann man eine einzelne, durch Semikolon abgeschlossene Anweisung oder eine in geschweifte Klammern eingeschlossene Verbundanweisung schreiben.

4.9.2 Erweiterte `if`-Anweisungen

In einer `if`- oder `else`-Klausel kann jede beliebige Anweisung stehen, sogar eine andere `if`- oder `else`-Anweisung. Daher lassen sich auch komplexe `if`-Anweisungen der folgenden Form erstellen:

```
if (Ausdruck1)
{
    if (Ausdruck2)
        Anweisung1;
    else
    {
        if (Ausdruck3)
            Anweisung2;
        else
            Anweisung3;
    }
}
else
    Anweisung4;
```

Diese umständliche `if`-Anweisung sagt aus: »Wenn `Ausdruck1` gleich `true` ist und `Ausdruck2` gleich `true` ist, führe `Anweisung1` aus. Wenn `Ausdruck1` gleich `true`, aber `Ausdruck2` nicht `true` ist, dann führe `Anweisung2` aus, wenn `Ausdruck3` `true` ist. Wenn `Ausdruck1` gleich `true`, aber `Ausdruck2` und `Ausdruck3` gleich `false` sind, führe `Anweisung3` aus. Wenn schließlich `Ausdruck1` nicht `true` ist, führe `Anweisung4` aus.« Wie man sieht, können komplexe `if`-Anweisungen einiges zur Verwirrung beitragen!

Listing 4.4 zeigt ein Beispiel einer derartigen komplexen `if`-Anweisung.

Listing 4.4: nestedif.cpp – Verschachtelte if-Anweisung

```
 1: // Listing 4.4 - Verschachtelte if-Anweisung
 2: #include <iostream>
 3:
 4: int main()
 5: {
 6:     // Zwei Zahlen abfragen
 7:     // Die Zahlen an bigNumber und littleNumber zuweisen
 8:     // Wenn bigNumber größer als littleNumber ist,
 9:     // testen, ob sie ohne Rest teilbar sind.
10:     // Wenn ja, testen, ob die Zahlen gleich sind.
11:
12:     int firstNumber, secondNumber;
13:     std::cout << "Bitte zwei Zahlen eingeben.\nDie erste: ";
14:     std::cin >> firstNumber;
15:     std::cout << "\nDie zweite: ";
16:     std::cin >> secondNumber;
17:     std::cout << "\n\n";
18:
```

```
 19:     if (firstNumber >= secondNumber)
 20:     {
 21:         if ( (firstNumber % secondNumber) == 0) // ganzzahlig
teilbar?
 22:         {
 23:             if (firstNumber == secondNumber)
 24:                 std::cout << "Beide Zahlen sind gleich!\n";
 25:             else
 26:                 std::cout << "Zahlen ohne Rest teilbar!\n";
 27:         }
 28:         else
 29:             std::cout << "Zahlen nicht ohne Rest teilbar!\n";
 30:     }
 31:     else
 32:         std::cout << "Die zweite Zahl ist groesser!\n";
 33:     return 0;
 34: }
```

```
Bitte zwei Zahlen eingeben.
Die erste: 10
Die zweite: 2
Zahlen ohne Rest teilbar!
```

Mit anderen Zahlen erhalten Sie beispielsweise folgende Ergebnisse:

```
Bitte zwei Zahlen eingeben.
Die erste: 2
Die zweite: 2
Beide Zahlen sind gleich!
```

Das Programm fordert zur Eingabe von zwei Zahlen auf und vergleicht sie dann. Die erste `if`-Anweisung in Zeile 19 prüft, ob die erste Zahl größer oder gleich der zweiten ist. Falls nicht, kommt die `else`-Klausel in Zeile 31 zur Ausführung.

Wenn die erste `if`-Bedingung gleich `true` ist, wird der in Zeile 20 beginnende Codeblock ausgeführt und die zweite `if`-Anweisung in Zeile 21 getestet. Hier erfolgt die Prüfung, ob die erste Zahl modulo der zweiten keinen Rest liefert. Wenn das so ist, sind die Zahlen entweder ohne Rest teilbar oder einander gleich. Die `if`-Anweisung in Zeile 23 prüft auf Gleichheit und zeigt die entsprechende Meldung an.

Wenn die `if`-Anweisung in Zeile 21 `false` ergibt, wird die `else`-Anweisung in Zeile 28 ausgeführt.

4.10 Geschweifte Klammern in verschachtelten if-Anweisungen

Obwohl es zulässig ist, die geschweiften Klammern in `if`-Anweisungen mit nur einer einzelnen Anweisung wegzulassen und es ebenfalls zulässig ist, `if`-Anweisungen wie

```
if (x > y)              // Wenn x größer als y ist,
    if (x < z)          // und wenn x kleiner als z ist,
        x = y;          // dann setze x auf den Wert in z.
```

zu verschachteln, kann das bei umfangreich verschachtelten Anweisungen kaum noch zu durchschauen sein. Denken Sie daran, dass man mit Whitespaces und Einzügen die Lesbarkeit des Quelltextes erhöhen kann und der Compiler von diesen gestalterischen Elementen keine Notiz nimmt. Man kann leicht die Logik durcheinanderbringen und versehentlich eine `else`-Anweisung der falschen `if`-Anweisung zuordnen. Listing 4.5 verdeutlicht dieses Problem.

Listing 4.5: bracesif.cpp – Mit Klammern verdeutlichen, welche else-Anweisung zu welcher if-Anweisung gehört

```
 1: // Listing 4.5 - Demonstriert, warum Klammern in verschachtelten
 2: // if-Anweisungen so wichtig sind
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     int x;
 8:     std::cout << "Bitte eine Zahl kleiner als 10 oder groesser
als 100 eingeben: ";
 9:     std::cin >> x;
10:     std::cout << "\n";
11:
12:     if (x > 10)
13:         if (x > 100)
14:             std::cout << "Groesser als 100. Danke!\n";
15:     else                          // nicht das beabsichtigte else!
16:         std::cout << "Kleiner als 10. Danke!\n";
17:
18:     return 0;
19: }
```

```
Bitte eine Zahl kleiner als 10 oder groesser als 100 eingeben: 20
Kleiner als 10. Danke!
```

Der Programmierer hat beabsichtigt, den Benutzer zur Eingabe einer Zahl kleiner als 10 oder größer als 100 aufzufordern, auf den korrekten Wert zu testen und dann eine Danksagung auszugeben. Mit dem angegebenen Code funktioniert das aber nicht.

Wenn die `if`-Anweisung in Zeile 12 den Wert `true` liefert, wird die folgende Anweisung (Zeile 13) ausgeführt. Hier erfolgt die Ausführung von Zeile 13, wenn die eingegebene Zahl größer als 10 ist. Zeile 13 enthält ebenfalls eine `if`-Anweisung, die `true` liefert, wenn die eingegebene Zahl größer als 100 ist. In diesem Fall wird die Anweisung in Zeile 14 ausgeführt.

Ist die eingegebene Zahl kleiner als oder gleich 10, liefert die `if`-Anweisung in Zeile 12 das Ergebnis `false`. Die Programmsteuerung geht an die auf `if` folgende Zeile, in diesem Fall Zeile 18, über. Wenn man eine Zahl kleiner als 10 eingibt, sieht die Ausgabe wie folgt aus:

```
Bitte eine Zahl kleiner als 10 oder groesser als 100 eingeben: 9
```

Die `else`-Klausel in Zeile 15 sollte offensichtlich zur `if`-Anweisung von Zeile 12 gehören, was auch der Einzug im Quelltext dokumentiert. Leider ist die `else`-Anweisung tatsächlich mit `if` von Zeile 13 verbunden. Damit enthält dieses Programm einen nicht auf Anhieb erkennbaren Fehler, da ihn der Compiler nicht bemängelt. Wir haben es hier mit einem zulässigen C++-Programm zu tun, das aber nicht wie vorgesehen arbeitet. Dazu kommt noch, dass bei den meisten Tests, die der Programmierer durchführt, das Programm zu laufen scheint. Solange man eine Zahl größer als 100 eingibt, funktioniert alles wunderbar.

Für Sie und nicht für den Computer

Denken Sie daran, dass Whitespaces und Einzüge für Sie als Programmierer gedacht sind und nicht für den Compiler! Der Compiler interessiert sich nicht im geringsten dafür, wie Sie die `if`-Anweisungen optisch gestalten. Verwenden Sie ein ansprechendes Format, um sich die Wirkung zu verdeutlichen. Wenn aber die Einzüge nicht mit dem übereinstimmen, was der Compiler tut, kann das zu Verwirrungen führen.

In Listing 4.6 ist das Problem durch Einfügen der erforderlichen Klammern beseitigt.

Listing 4.6: properbraces.cpp – Richtige Verwendung von geschweiften Klammern bei einer if-Anweisung

```
 1: // Listing 4.6 - Demonstriert die richtige Verwendung
 2: // von Klammern in verschachtelten if-Anweisungen
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     int x;
 8:     std::cout << "Bitte eine Zahl kleiner als 10 oder groesser
als 100 eingeben: ";
 9:     std::cin >> x;
10:     std::cout << "\n";
11:
12:     if (x > 10)
13:     {
14:         if (x > 100)
15:             std::cout << "Groesser als 100. Danke!\n";
16:     }
17:     else        // arbeitet jetzt wie vorgesehen!
18:         std::cout << "Kleiner als 10. Danke!\n";
19:     return 0;
20: }
```

```
Bitte eine Zahl kleiner als 10 oder groesser als 100 eingeben: 20
```

Die geschweiften Klammern in den Zeilen 13 und 16 schließen alle dazwischen liegenden Anweisungen zu einem Block zusammen. Nunmehr gehört die `else`-Klausel in Zeile 17 wie beabsichtigt zur `if`-Anweisung in Zeile 12.

Der Benutzer hat die Zahl `20` eingegeben, sodass die `if`-Anweisung in Zeile 12 den Wert `true` liefert. Die `if`-Anweisung in Zeile 14 liefert `false`, sodass keine Ausgabe erfolgt. Besser wäre es, wenn der Programmierer eine weitere `else`-Klausel nach Zeile 15 vorgesehen hätte, um fehlerhafte Eingaben abzufangen und eine entsprechende Meldung anzuzeigen.

Defensive Programmierung

Die im Buch gezeigten Programme sind bewusst einfach gehalten und dienen nur dazu, das jeweils behandelte Thema zu verdeutlichen. Es wurde auf »narrensicheren« Code verzichtet, der einen Schutz gegen fehlerhafte Eingaben bietet. In professionellen Programmen muss man mit jedem möglichen Benutzerfehler rechnen und diese Fehler entsprechend »sanft« abfangen.

4.11 Logische Operatoren

Oftmals muss man mehrere Vergleiche auf einmal anstellen. »Ist es richtig, dass x größer ist als y, und ist es ebenso richtig, dass y größer ist als z?« Ein Programm muss ermitteln, dass beide Bedingungen `true` sind, oder dass irgendeine andere Bedingung `true` ist, um eine Aktion auszuführen.

Stellen Sie sich ein intelligentes Alarmsystem vor, das nach folgender Logik arbeitet: »Wenn der Türalarm ausgelöst wird UND es später als 18:00 Uhr ist UND es sich NICHT um die Urlaubszeit handelt ODER ein Wochenende ist, dann rufe die Polizei.« Eine derartige Auswertung lässt sich mit den drei logischen Operatoren von C++ realisieren. Tabelle 4.2 zeigt diese Operatoren.

Operator	Symbol	Beispiel
`AND`	`&&`	`Ausdruck1 && Ausdruck2`
`OR`	`||`	`Ausdruck1 || Ausdruck2`
`NOT`	`!`	`!Ausdruck`

Tabelle 4.2: Logische Operatoren

4.11.1 Logisches AND

Eine logische `AND`-Anweisung wertet zwei Ausdrücke aus. Sind beide Ausdrücke `true`, liefert die logische `AND`-Anweisung ebenfalls `true`. Wenn es wahr (`true`) ist, dass Sie hungrig sind, UND (AND) es wahr ist, dass Sie Geld haben, dann ist es wahr, dass Sie ein Mittagessen kaufen können. In diesem Sinn wird

```
if ( (x == 5) && (y == 5) )
```

zu `true` ausgewertet, wenn sowohl `x` als auch `y` gleich 5 sind. Der Ausdruck liefert `false`, wenn mindestens eine der beiden Variablen nicht gleich 5 ist. Beachten Sie, dass beide Seiten `true` sein müssen, damit der gesamte Ausdruck zu `true` wird.

Achten Sie auch darauf, dass das logische `AND` aus zwei Et-Zeichen (`&&`) besteht.

4.11.2 Logisches OR

Eine logische `OR`-Anweisung wertet zwei Ausdrücke aus. Wenn einer der beiden `true` ist, ergibt sich der Ausdruck zu `true`. Wenn man Geld ODER eine Kreditkarte hat, kann man die Rechnung bezahlen. Beide Zahlungsmittel sind nicht gleichzeitig erforderlich, man braucht nur eines, auch wenn es nichts schadet, beides zur Verfügung zu haben.

Der Ausdruck

```
if ( (x == 5) || (y == 5) )
```

liefert `true`, wenn `x` oder `y` gleich 5 ist oder wenn beide gleich 5 sind. Wenn bereits `x` gleich 5 ist, prüft der Compiler gar nicht mehr auf `y`!

Beachten Sie, dass das logische `OR` aus zwei senkrechten Strichen (`||`) besteht.

4.11.3 Logisches NOT

Eine logische `NOT`-Anweisung liefert `true`, wenn der zu testende Ausdruck `false` ergibt. Der Ausdruck

```
if ( !(x == 5) )
```

ist nur dann `true`, wenn `x` nicht gleich 5 ist. Den gleichen Test kann man auch als

```
if (x != 5)
```

schreiben.

4.12 Rangfolge der Vergleichsoperatoren

C++-Ausdrücke mit Vergleichsoperatoren und logischen Operatoren liefern einen Wert zurück: 1 (`true`) oder 0 (`false`). Wie bei allen Ausdrücken gibt es auch hier eine Rangfolge, die bestimmt, welcher Vergleich zuerst ausgewertet wird. Diese Tatsache ist wichtig, wenn man den Wert einer Anweisung wie

```
if ( x > 5 && y > 5 || z > 5)
```

bestimmt. Der Programmierer könnte beabsichtigen, dass dieser Ausdruck nur zu `true` ausgewertet wird, wenn sowohl `x` als auch `y` größer als 5 sind oder wenn `z` größer als 5 ist. Andererseits kann der Programmierer auch wollen, dass dieser Ausdruck nur dann zu `true` wird, wenn `x` größer als 5 ist und wenn es auch wahr ist, dass entweder `y` oder `z` größer als 5 ist.

Wenn `x` gleich 3 ist und sowohl `y` als auch `z` gleich 10 sind, liefert die erste Interpretation das Ergebnis `true` (`z` ist größer als 5, sodass `x` und `y` ignoriert werden), während die zweite Auslegung `false` ergibt (es ist nicht wahr, dass sowohl `x` als auch `y` größer als 5 sind, und es ist nicht wahr, dass `z` größer als 5 ist).

Die von der Rangfolge bestimmte Reihenfolge der Vergleiche kann man mit Klammern sowohl ändern als auch deutlicher darstellen:

```
if ( (x > 5) && (y > 5 || z > 5) )
```

Mit den obigen Werten liefert diese Anweisung das Ergebnis `false`. Da es nicht wahr ist, dass `x` größer als 5 ist, ergibt die linke Seite der `AND`-Anweisung

false, und damit wird die gesamte Anweisung zu false. Denken Sie daran, dass bei der AND-Anweisung beide Seiten true sein müssen – irgend etwas ist nicht sowohl »gutschmeckend« UND »gut für Dich«, wenn es nicht gut schmeckt.

Gruppierungen

Es empfiehlt sich, mit zusätzlichen Klammern klarzustellen, was man gruppieren möchte. Denken Sie immer daran: Das Ziel ist ein funktionsfähiges Programm, das zudem leicht zu lesen und zu verstehen ist.

4.13 Mehr über Wahr und Unwahr

C++ behandelt 0 als false und jeden anderen Wert als true. Da Ausdrücke immer einen Wert haben, nutzen viele C++-Programmierer diese Tatsache in ihren if-Anweisungen aus. Eine Anweisung wie

```
if (x)           // wenn x gleich true (ungleich Null)
    x = 0;
```

kann man lesen als »wenn x einen Wert ungleich 0 hat, setze x auf 0«. Hier besteht »Verdunklungsgefahr«! Klarer formuliert man

```
if (x != 0)      // wenn x ungleich 0
    x = 0;
```

Beide Anweisungen sind zulässig, wobei aber die zweite verständlicher ist. Es gehört zum guten Programmierstil, die erste Methode für echte logische Tests zu reservieren, statt auf Werte ungleich 0 zu testen.

Die folgenden Anweisungen sind ebenfalls äquivalent:

```
if (!x)          // wenn x gleich false ist (Null)
if (x == 0)      // wenn x gleich Null ist
```

Die zweite Anweisung ist allerdings deutlicher und etwas einfacher zu verstehen.

4.14 Fragen und Antworten

F: Warum verwendet man unnötige Klammern, wenn die Rangfolge bestimmt, welche Operatoren zuerst auszuwerten sind?

A: Obwohl es stimmt, dass der Compiler den Vorrang kennt und dass ein Programmierer bei Bedarf die Vorrangregeln nachschlagen kann, ist verständlicherer Code auch leichter zu warten.

F: Wenn die Vergleichsoperatoren immer 1 oder 0 zurückgeben, warum werden dann andere Werte als `true` angesehen?

A: Die Vergleichsoperatoren geben 1 oder 0 zurück, aber jeder Ausdruck liefert einen Wert und solche Werte lassen sich ebenfalls in einer `if`-Anweisung auswerten. Dazu ein Beispiel:

```
if ( (x = a + b) == 35 )
```

Das ist eine durchaus zulässige C++-Anweisung, die auch dann einen Wert liefert, wenn die Summe von `a` und `b` nicht gleich 35 ist. Beachten Sie auch, dass `x` in jedem Fall den Wert erhält, der sich aus der Addition von `a` und `b` ergibt.

F: Welche Wirkung haben Tabulatoren, Leerzeichen und Zeilenschaltungen auf das Programm?

A: Tabulatoren, Leerzeichen und Zeilenschaltungen (so genannte Whitespaces) haben auf das Programm keinen Einfluss. Allerdings lässt sich mit diesen Zeichen der Quelltext gestalten und damit die Lesbarkeit verbessern.

F: Sind negative Zahlen `true` oder `false`?

A: Alle Zahlen ungleich 0 – sowohl positive als auch negative – sind `true`.

4.15 Workshop

Nachdem Sie sich mit Ausdrücken und Anweisungen beschäftigt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr eben erworbenes Wissen zu festigen.

4.15.1 Kontrollfragen

1. Worin besteht der Unterschied zwischen `x++` und `++x`?
2. Welcher Operator ist in den meisten Sprachen vorhanden, in C++ jedoch nicht?
3. Worin besteht der Unterschied zwischen einem L-Wert und einem R-Wert?
4. Was bewirkt der Modulo-Operator?

4.15.2 Übungen

1. Schreiben Sie drei Programme, um einen Wert zu einer Variablen zu addieren. Verwenden Sie jeweils eine der Formen `a=a+1`, `a+=1` und `a++`. Kompilieren, linken und starten Sie diese Programme. Unterscheiden sich die Größen? Lassen sich Unterschiede in der Ausführungsgeschwindigkeit feststellen?
2. Geben Sie einen mathematischen Ausdruck mithilfe der Borland-IDE in ein Programm ein, setzen Sie den Cursor auf einen der mathematischen Operatoren und drücken Sie F1. Wie nützlich sind die gelieferten Informationen?
3. Versuchen Sie ein Programm zu erstellen, das im selben Ausdruck sowohl Präfix- als auch Postfix-Inkrement oder -Dekrement auf eine Variable anwendet (in der Art wie `++x++`). Lässt sich das Programm kompilieren? Ist es überhaupt sinnvoll? Wie sieht es aus, wenn Sie Inkrement- und Dekrement-Operatoren kombinieren?

4.15.3 Antworten zu den Kontrollfragen

1. Die erste Form ist ein Postfix-Inkrement, wobei zuerst der gesamte Ausdruck ausgewertet (der Wert bestimmt) und dann erst `x` inkrementiert wird. Die zweite Form ist ein Präfix-Inkrement, wobei `x` zuerst inkrementiert und dann erst der Wert des Ausdrucks bestimmt wird.
2. Der Potenz- oder Exponentialoperator (der Operator, der einen Wert zu einem Exponenten erhebt) ist in C++ nicht als einfacher Operator vorhanden; für diese Operation muss man eine Funktion aufrufen.
3. Ein L-Wert kann auf der linken Seite eines Gleichheitszeichens erscheinen, während ein R-Wert auf der rechten Seite steht. Alle L-Werte können auch auf der rechten Seite eines Gleichheitszeichens vorkommen (sodass sie wie R-Werte wirken), während die meisten R-Werte nicht als L-Werte fungieren können.
4. Er berechnet den Rest nach einer Ganzzahldivision der angegebenen Operanden.

KAPITEL 5

jetzt lerne ich

Funktionen

In dieser Stunde lernen Sie,

- was eine Funktion ist und aus welchen Teilen sie besteht,
- wie man Funktionen deklariert und definiert,
- wie man Parameter an Funktionen übergibt,
- wie man einen Wert aus einer Funktion zurückgibt.

5.1 Was ist eine Funktion?

Eine *Funktion* ist praktisch ein Unterprogramm, das Daten verarbeitet und einen Wert zurückgibt. Jedes C++-Programm hat zumindest eine Funktion: `main()`. Beim Start des Programms wird `main()` automatisch aufgerufen. In `main()` können andere Funktionsaufrufe stehen, die wiederum andere Funktionen aufrufen können.

Jede Funktion hat einen eigenen Namen. Gelangt die Programmausführung zu einer Anweisung mit diesem Namen, verzweigt das Programm in den Rumpf der betreffenden Funktion. Bei Rückkehr aus der Funktion setzt sich die Programmausführung auf der nächsten Zeile nach dem Funktionsaufruf fort. Abbildung 5.1 verdeutlicht diesen Ablauf.

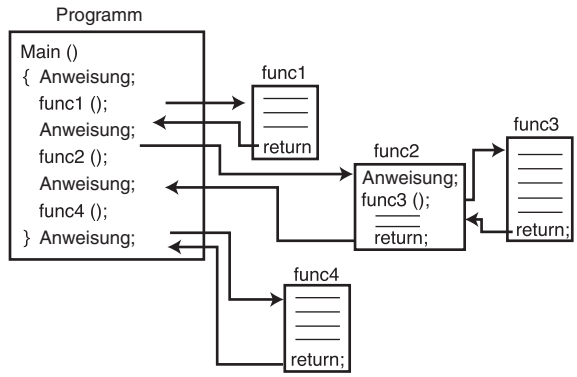


Abbildung 5.1: Ruft ein Programm eine Funktion auf, schaltet die Ausführung zur Funktion um und fährt nach Rückkehr aus der Funktion auf der Zeile nach dem Funktionsaufruf fort.

Eine gut konzipierte Funktion führt eine klar abgegrenzte Aufgabe aus – nicht mehr und nicht weniger – und kehrt dann zurück.

Komplexere Aufgaben sollte man auf mehrere Funktionen aufteilen und diese Funktionen dann aufrufen. Damit lässt sich der Code besser verstehen und leichter warten.

Das ganze Buch hindurch finden Sie Hinweise, wie Sie ein Programm wartungsfreundlicher gestalten könne. Den Löwenanteil der Projektkosten bildet nicht das Schreiben des Programms, sondern die Aufrechterhaltung seiner Brauchbarkeit und Zuverlässigkeit während der gesamten Lebenszeit des Programms.

5.2 Funktionen deklarieren und definieren

Bevor man eine Funktion verwenden kann, muss man sie zuerst *deklarieren* und dann *definieren*.

Die *Funktionsdeklaration* teilt dem Compiler den Namen, den Rückgabetyp und die Parameter der Funktion mit. Die Deklaration einer Funktion bezeichnet man als Prototyp. Er enthält keinen ausführbaren Code, d.h. keinen Funktionsrumpf.

Die *Funktionsdefinition* teilt dem Compiler die Arbeitsweise der Funktion mit. Keine Funktion lässt sich von irgendeiner anderen Funktion aufrufen, wenn sie nicht zuerst deklariert wurde. Die Definition enthält den eigentlichen Code – den Rumpf – der Funktion.

5.2.1 Funktionen deklarieren

Für die zum Lieferumfang des Compilers gehörenden Funktionen ist der Prototyp bereits geschrieben. Mit der Anweisung `#include` binden Sie einfach die erforderliche Datei ein, und schon ist alles eingerichtet.

Der *Prototyp* einer Funktion ist eine Anweisung, die demzufolge auch mit einem Semikolon endet. Er besteht aus dem Rückgabetyp, dem Namen und der Parameterliste der Funktion.

Die Parameterliste führt – durch Komma getrennt – alle Parameter zusammen mit deren Typen auf. Abbildung 5.2 verdeutlicht die Teile des Funktionsprototyps.

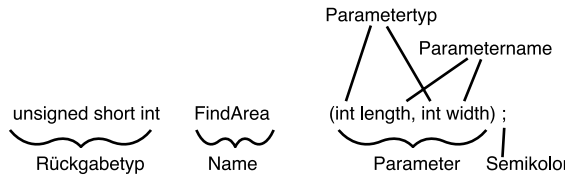


Abbildung 5.2: Teile des Prototyps einer Funktion

Der Prototyp und die Definition einer Funktion müssen hinsichtlich Rückgabetyp, Name und Parameterliste genau übereinstimmen. Andernfalls erhält man einen Compiler-Fehler. Allerdings müssen im Prototyp der Funktion nur die Typen und nicht die Namen der Parameter erscheinen. Das folgende Beispiel zeigt einen zulässigen Prototyp:

```
long FindArea(int, int);
```

Dieser Prototyp deklariert eine Funktion namens `FindArea`, die einen Wert vom Typ `long` zurückgibt und zwei ganzzahlige Parameter übernimmt. Diese Anweisung ist zwar zulässig, aber nicht empfehlenswert. Mit hinzugefügten Parameternamen ist der Prototyp verständlicher. Dieselbe Funktion könnte mit benannten Parametern folgendes Aussehen haben:

```
long FindArea(int length, int width);
```

Daraus lässt sich ohne weiteres erkennen, welche Aufgabe die Funktion realisiert und welche Parameter zu übergeben sind.

Beachten Sie, dass alle Funktionen einen Rückgabetyp haben. Listing 5.1 zeigt ein Programm, das den Funktionsprototyp für die Funktion `FindArea()` einbindet.

Listing 5.1: declarefunction.cpp – Deklaration, Definition und Verwendung einer Funktion

```
 1: // Listing 5.1 - Zeigt die Verwendung von Funktionsprototypen
 2: #include <iostream>
 3:
 4: int FindArea(int length, int width); // Funktionsprototyp
(Deklaration)
 5:
 6: int main()
 7: {
 8:     int lengthOfYard;
 9:     int widthOfYard;
10:     int areaOfYard;
11:
12:     std::cout << "\nWie breit ist Ihr Garten? ";
13:     std::cin >> widthOfYard;
14:     std::cout << "\nWie lang ist Ihr Garten? ";
15:     std::cin >> lengthOfYard;
16:
17:     areaOfYard= FindArea(lengthOfYard,widthOfYard);
18:
19:     std::cout << "\nDie Flaeche Ihres Gartens betraegt ";
20:     std::cout << areaOfYard;
21:     std::cout << " Quadratmeter\n\n";
22:     return 0;
23: }
24:
25: int FindArea(int l, int w) // Funktionsdefinition
26: {
27:      return l * w;
28: }
```

```
Wie breit ist Ihr Garten? 100
Wie lang ist Ihr Garten? 200
Die Flaeche Ihres Gartens betraegt 20000 Quadratmeter
```

Der Prototyp für die Funktion `FindArea()` steht in Zeile 4. Vergleichen Sie den Prototyp mit der Definition der Funktion in Zeile 25. Beachten Sie, dass Name, Rückgabetyp und die Parametertypen gleich sind. Andernfalls erhält man einen Compiler-Fehler. Praktisch besteht der einzige Unterschied darin, dass der Prototyp einer Funktion mit einem Semikolon endet und keinen Rumpf aufweist.

Weiterhin fällt auf, dass die Parameternamen im Prototyp mit `length` und `width` angegeben sind, während die Definition `l` und `w` enthält. Die Namen im Prototyp sind nicht erforderlich und dienen nur als Information für den Programmierer. Die Argumente übergibt man an die Funktion in der Reihenfolge, in der sie deklariert und definiert sind.

Die Definition (Zeilen 25 bis 28) können Sie auch vor den ersten Aufruf der Funktion setzen, d.h. vor Zeile 17 verschieben. In diesem Fall brauchen Sie den Prototyp nicht mehr. Diese Lösung ist zwar für kleinere Programme durchaus geeignet, es gehört aber nicht zum guten Programmierstil, von der Anordnung der Funktionsdefinitionen abhängig zu sein. Wird das Programm komplexer und rufen Funktionen andere Funktionen auf, lässt es sich nur noch schwer sicherstellen, dass alle Programme in der richtigen Reihenfolge definiert sind. Im Allgemeinen ist es vorzuziehen, alle Funktionen mit Prototypen zu deklarieren. Damit umgeht man eine Abhängigkeit von der Reihenfolge der Definitionen.

5.2.2 Funktionen definieren

Die Definition einer Funktion besteht aus dem Funktionskopf und ihrem Rumpf. Der Kopf entspricht genau dem Prototyp der Funktion, außer dass die Parameter benannt sein müssen und kein abschließendes Semikolon vorhanden ist.

Der Rumpf der Funktion ist eine Gruppe von Anweisungen, die in geschweifte Klammern eingeschlossen sind. Abbildung 5.3 zeigt Kopf und Rumpf einer Funktion.

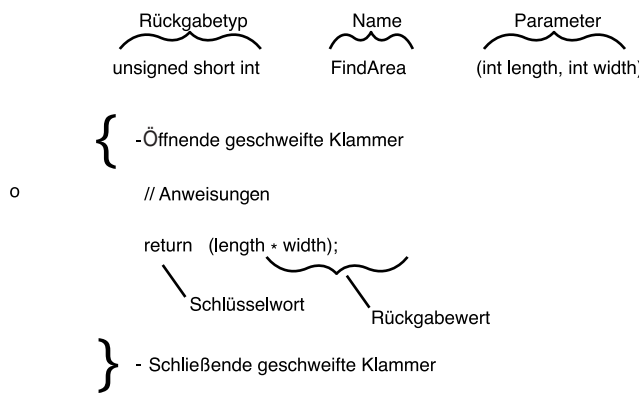


Abbildung 5.3: Kopf und Rumpf einer Funktion

5.2.3 Funktionen

Ein Funktionsprototyp gibt dem Compiler den Namen, den Rückgabewert und die Parameter der Funktion an.

```
rueckgabetyp funktionsname ( [typ [parameterName]]...);
```

Die Funktionsdefinition sagt dem Compiler, wie die Funktion arbeitet.

```
rueckgabetyp funktionsname ( [typ parameterName]...)
{
    Anweisungen;
}
```

Funktionen müssen keine Parameter aufweisen. Falls Parameter vorkommen, müssen sie nicht als Namen im Prototyp erscheinen, sondern nur als Typ. Ein Prototyp endet immer mit einem Semikolon.

Eine Funktionsdefinition muss hinsichtlich Rückgabetyp und Parameterliste mit ihrem Prototyp übereinstimmen. Sie muss die Namen für alle Parameter bereitstellen und der Rumpf der Funktionsdefinition ist in geschweifte Klammern einzuschließen. Alle Anweisungen innerhalb des Rumpfes der Funktion müssen jeweils mit einem Semikolon abgeschlossen sein, während die Funktion selbst nicht mit einem Semikolon, sondern mit einer schließenden geschweiften Klammer beendet wird.

Wenn die Funktion einen Rückgabewert liefert, sollte sie mit einer `return`-Anweisung enden. `return`-Anweisungen sind aber auch an beliebigen Stellen innerhalb des Funktionsrumpfes zulässig.

Jede Funktion hat einen Rückgabetyp. Gibt man keinen Typ explizit an, wird `int` angenommen. Man sollte für jede Funktion explizit einen Rückgabetyp festlegen. Gibt eine Funktion keinen Wert zurück, lautet der Rückgabetyp `void`.

Die folgenden Beispiele zeigen weitere Funktionsprototypen:

```
long FindArea(long length, long width);  // gibt long zurück, hat
zwei Parameter
void PrintMessage(int messageNumber);    // gibt void zurück, hat
einen Parameter
int GetChoice();                         // gibt int zurück, hat
keine Parameter
BadFunction();                           // gibt int zurück, hat
keine Parameter
```

Hier zwei Beispiele für Funktionsdefinitionen:

```
long Area(long l, long w)
{
     return l * w;
}
```

```
void PrintMessage(int whichMsg)
{
     if (whichMsg == 0)
          cout << "Hello.\n";
     if (whichMsg == 1)
          cout << "Goodbye.\n";
     if (whichMsg > 1)
          cout << "Was ist los?\n";
}
```

Es gibt praktisch keine Grenze für die Anzahl oder die Typen von Anweisungen, die man in einem Funktionsrumpf unterbringen kann.

Gut konzipierte Funktionen sind allerdings in der Regel relativ klein. Die große Mehrheit Ihrer Funktionen wird lediglich aus einer Hand voll Codezeilen bestehen.

5.3 Variablen in Funktionen

Variablen lassen sich innerhalb von Funktionen auf verschiedene Weise behandeln. Wir haben bereits die Parameter erwähnt, die der Aufrufer an die Funktion übergibt. Außerdem kann eine Funktion Variablen enthalten, deren Gültigkeitsbereich auf die Funktion beschränkt ist (lokale Variablen) oder die dem gesamten Programm zugänglich sind (globale Variablen).

5.3.1 Lokale Variablen

Man kann an eine Funktion nicht nur Variablen übergeben, sondern auch Variablen innerhalb des Funktionsrumpfes deklarieren. Diese so genannten *lokalen Variablen* existieren nur innerhalb der Funktion selbst. Wenn die Funktion zurückkehrt, sind die lokalen Variablen nicht mehr zugänglich.

Lokale Variablen definiert man wie jede andere Variable. Die an eine Funktion übergebenen Parameter gelten ebenfalls als lokale Variablen und lassen sich genauso verwenden, als hätte man sie innerhalb des Funktionsrumpfes definiert. Listing 5.2 zeigt ein Beispiel für die Verwendung von Parametern und lokal definierten Variablen innerhalb einer Funktion. Die Ausgaben stammen von drei Programmstarts, wobei bei jedem Aufruf andere Werte – 212, 32 und 85 – bereitgestellt werden.

Listing 5.2: localvar.cpp – Lokale Variablen und Parameter

```
1: #include <iostream>
2:
3: float Convert(float);
4:
5: int main()
6: {
```

```
 7:     float TempFer;
 8:     float TempCel;
 9:
10:     std::cout << "Bitte Temperatur in Fahrenheit eingeben: ";
11:     std::cin >> TempFer;
12:     TempCel = Convert(TempFer);
13:     std::cout << "\nDie Temperatur in Grad Celsius ist: ";
14:     std::cout << TempCel << std::endl;
15:     return 0;
16: }
17:
18: float Convert(float TempFer)
19: {
20:     float TempCel;
21:     TempCel = ((TempFer - 32) * 5) / 9;
22:     return TempCel;
23: }
```

```
Bitte Temperatur in Fahrenheit eingeben: 212
Die Temperatur in Grad Celsius ist: 100

Bitte Temperatur in Fahrenheit eingeben: 32
Die Temperatur in Grad Celsius ist: 0

Bitte Temperatur in Fahrenheit eingeben: 85
Die Temperatur in Grad Celsius ist: 29.4444
```

Die Zeilen 7 und 8 deklarieren zwei Variablen vom Typ `float`. Eine nimmt die Temperatur in Fahrenheit auf und eine die Temperatur in Grad Celsius. Die Anweisung in Zeile 10 fordert den Benutzer auf, eine Temperatur in Fahrenheit einzugeben. Dieser Wert wird an die Funktion `Convert` übergeben.

Die Ausführung springt zur ersten Zeile der Funktion `Convert()` in Zeile 20. Die Funktion deklariert hier eine lokale Variable, die ebenfalls mit `TempCel` benannt ist. Beachten Sie, dass diese lokale Variable nichts mit der in Zeile 8 deklarierten Variablen `TempCel` zu tun hat, sondern nur innerhalb der Funktion `Convert()` existiert. Der als Parameter `TempFer` übergebene Wert ist ebenfalls lediglich eine lokale Kopie der durch `main()` übergebenen Variablen.

Man könnte den Parameter der Funktion mit `FerTemp` und die lokale Variable mit `CelTemp` benennen, ohne dass sich an der Arbeitsweise des Programms irgendetwas ändert. Überzeugen Sie sich davon, indem Sie diese Namen eingeben und das Programm erneut kompilieren.

Die lokale Variable `TempCel` der Funktion erhält den Wert zugewiesen, der aus der Subtraktion des Wertes 32 vom Parameter `TempFer`, der Multiplikation mit 5 und der Division durch 9 resultiert. Diesen Wert liefert die Funktion als Rückgabewert und Zeile 12 weist ihn der Variablen `TempCel` in der Funktion `main()` zu. Die Ausgabe erfolgt in Zeile 14.

Das Programm wird dreimal ausgeführt. Beim ersten Mal übergibt man den Wert 212, um zu prüfen, dass der Siedepunkt des Wassers in Grad Fahrenheit (212) die korrekte Antwort in Grad Celsius (100) generiert. Der zweite Test bezieht sich auf den Gefrierpunkt des Wassers. Der dritte Test verwendet eine willkürliche Zahl, um ein gebrochenes Ergebnis zu erzeugen.

Zur Übung können Sie das Programm erneut mit anderen Namen wie in Listing 5.3 eingeben:

Listing 5.3: localvar2.cpp – Lokale Variablen und Parameter (mit Variablennamen, die sich zwischen main und Convert unterscheiden)

```
 1: #include <iostream>
 2:
 3: float Convert(float);
 4:
 5: int main()
 6: {
 7:     float TempFer;
 8:     float TempCel;
 9:
10:     std::cout << "Bitte Temperatur in Fahrenheit eingeben: ";
11:     std::cin >> TempFer;
12:     TempCel = Convert(TempFer);
13:     std::cout << "\nDie Temperatur in Grad Celsius ist: ";
14:     std::cout << TempCel << std::endl;
15:     return 0;
16: }
17:
18: float Convert(float Fer)
19: {
20:     float Cel;
21:     Cel = ((Fer - 32) * 5) / 9;
22:     return Cel;
23: }
```

Die Ergebnisse sollten dieselben sein.

Zu einer Variablen gehört ein *Gültigkeitsbereich*. Dieser bestimmt, wie lange eine Variable in einem Programm zugänglich ist und wo man auf sie zugreifen kann. Der Gültigkeitsbereich der in einem Block deklarierten Variablen beschränkt sich auf diesen Block. Nur innerhalb dieses Blocks kann man auf die Variablen zugreifen, außerhalb des Blocks »verschwinden« sie. Globale Variablen besitzen einen globalen Gültigkeitsbereich und sind an allen Stellen eines Programms zugänglich.

Wo immer Sie wollen!

Variablen können Sie innerhalb jedes beliebigen Blocks deklarieren, so etwa in den geschweiften Klammern von `if () { }`. Denken Sie daran, dass Variablen nur innerhalb des Blocks existieren, in dem sie deklariert wurden.

5.3.2 Globale Variablen

Variablen, die außerhalb aller Funktionen definiert sind, weisen einen *globalen* Gültigkeitsbereich auf und sind daher für jede Funktion im Programm einschließlich der Funktion `main()` verfügbar.

In C++ vermeidet man globale Variablen, da sich sonst schwer zu überblickender und zu wartender Code ergeben kann. Weder in diesem Buch noch in den Programmen des Autors werden Sie globale Variablen finden.

5.4 Funktionsargumente

Funktionsargumente müssen nicht alle vom selben Typ sein. Es ist durchaus sinnvoll, eine Funktion zu schreiben, die eine Ganzzahl, zwei Zahlen vom Typ `long` und ein Zeichen als Argumente übernimmt.

Als Funktionsausdruck ist jeder gültige C++-Ausdruck zulässig. Dazu gehören Konstanten, mathematische und logische Ausdrücke und andere Funktionen, die einen Wert zurückgeben.

Wenn Sie zum Beispiel eine Funktion als

```
int MyFunction(int theIntegerParam, bool the Boolean);
```

deklariert haben, dürfen Sie diese Funktion mit den folgenden Argumenten aufrufen:

```
int z, x = 3, y = 5;          // die Variablen deklarieren
z = MyFunction(x,y);          // eine int- und eine bool-Variable
übergeben
z = MyFunction(32,true);      // zwei Konstanten übergeben
z = MyFunction(23+9, 100>5); // Ausdrücke mit den Ergebnissen 32 bzw.
true
```

Der letzte Aufruf ist in seiner Wirkung mit dem zweiten Funktionsaufruf identisch. Und wenn Sie die Funktionen

```
int MyIntFunction(int x, int y);
bool MyBoolFunction(int x, int y);
```

deklariert haben, können Sie `MyFunction` wie folgt aufrufen:

```
z = MyFunction(MyIntFunction(3,5), MyBoolFunction(2,4));
```

Praktisch übergeben Sie in diesem Aufruf als Parameter den Integer-Wert, den der Aufruf von `MyIntFunction(3,5)` zurückgibt, und den booleschen Wert, den die Funktion `MyBoolFunction(2,4)` liefert.

5.4.1 Funktionen als Parameter in Funktionen

Obwohl es zulässig ist, eine Funktion zu verwenden, die einen Wert als Parameter an eine andere Funktion zurückgibt, kann man derartigen Code schwer lesen und kaum auf Fehler untersuchen.

Nehmen wir zum Beispiel die Funktionen `doubler()`, `tripler()`, `square()` und `cube()`, die alle einen Wert zurückgeben. Man kann schreiben

```
Answer = (doubler(tripler(square(cube(myValue)))));
```

Diese Anweisung übergibt die Variable `myValue` als Argument an die Funktion `cube()`, deren Rückgabewert als Argument an die Funktion `square()` dient. Die Funktion `square()` wiederum liefert ihren Rückgabewert an `tripler()` und diesen Rückgabewert übernimmt die Funktion `doubler()`. Der Rückgabewert dieser verdoppelten, verdreifachten, quadrierten und zur dritten Potenz erhobenen Zahl wird jetzt an `Answer` zugewiesen.

Man kann nur schwer erkennen, was dieser Code bewirkt. (Findet das Verdreifachen vor oder nach dem Quadrieren statt?) Falls diese Anweisung ein falsches Ergebnis liefert, lässt sich kaum die »schuldige« Funktion ermitteln.

Als Alternative kann man das Ergebnis jedes einzelnen Schrittes an eine Variable zur Zwischenspeicherung zuweisen:

```
unsigned long myValue = 2;
unsigned long cubed  =   cube(myValue);        // zur dritten Potenz
= 8
unsigned long squared = square(cubed);         // quadriert = 64
unsigned long tripled = tripler(squared);      // verdreifacht = 192
unsigned long Answer =  doubler(tripled);      // Answer = 384
```

Jetzt lässt sich jedes Zwischenergebnis untersuchen und die Reihenfolge der Ausführung ist klar erkennbar.

5.5 Parameter sind auch lokale Variablen

Die an eine Funktion übergebenen Argumente sind zur Funktion lokal. An den Argumenten vorgenommene Änderungen beeinflussen die Werte in der aufrufenden Funktion nicht. Man spricht hier von einer *Übergabe als Wert*. Das bedeutet, dass die Funktion von jedem Argument eine lokale Kopie anlegt. Die lokalen Kopien lassen sich wie andere lokale Variablen behandeln. Listing 5.4 verdeutlicht dieses Konzept.

Listing 5.4: passbyvar.cpp – Übergabe als Wert

```
// Listing 5.4 - Zeigt die Übergabe als Wert
#include <iostream>

void swap(int x, int y);

int main()
{
    int x = 5, y = 10;
    std::cout << "Main. Vor Vertauschung, x: " << x
                                    << " y: " << y << "\n";
    swap(x,y);
    std::cout << "Main. Nach Vertauschung, x: " << x
                                    << " y: " << y << "\n";
    return 0;
}

void swap (int x, int y)
{
    int temp;
    std::cout << "Swap. Vor Vertauschung, x: " << x
                                    << " y: " << y << "\n";
    temp = x;
    x = y;
    y = temp;
    std::cout << "Swap. Nach Vertauschung, x: " << x
                                    << " y: " << y << "\n";
}
```

```
Main. Vor Vertauschung, x: 5 y: 10
Swap. Vor Vertauschung, x: 5 y: 10
Swap. Nach Vertauschung, x: 10 y: 5
Main. Nach Vertauschung, x: 5 y: 10
```

Dieses Programm initialisiert zwei Variablen in `main()` und übergibt sie dann an die Funktion `swap()`, die scheinbar eine Vertauschung vornimmt. Wenn man die Werte allerdings erneut in `main()` untersucht, ist keine Veränderung festzustellen!

Zeile 8 initialisiert die Variablen, die Zeilen 9 und 10 zeigen die Werte an. In Zeile 11 steht der Aufruf der Funktion `swap()` mit der Übergabe der Variablen.

Die Ausführung des Programms verzweigt in die Funktion `swap()`, die in den Zeilen 20 und 21 die Werte erneut ausgibt. Wie erwartet, befinden sie sich in derselben Reihenfolge wie in `main()`. In den Zeilen 22 bis 24 findet die Vertauschung statt. Die Ausgabe in den Zeilen 25 und 26 bestätigt diese Aktion. Solange wir uns in der Funktion `swap()` befinden, sind die Werte tatsächlich vertauscht.

Die Programmausführung kehrt zu Zeile 12 (in der Funktion `main()`) zurück, wo die Werte nicht mehr vertauscht sind.

Wie Sie bemerkt haben, findet die Übergabe an die Funktion `swap()` als Wert statt. Folglich legt die Funktion `swap()` lokale Kopien der Werte an. Die Vertauschung in den Zeilen 22 bis 24 betrifft nur die lokalen Kopien und hat keinen Einfluss auf die Variablen in der Funktion `main()`.

Später im Buch lernen Sie Alternativen zur Übergabe als Wert kennen. Damit lassen sich dann auch die Werte in `main()` ändern.

5.6 Rückgabewerte

Funktionen geben entweder einen Wert oder `void` zurück. Das Schlüsselwort `void` ist ein Signal an den Compiler, dass kein Wert zurückgegeben wird.

Die Syntax für die Rückgabe eines Wertes aus einer Funktion besteht aus dem Schlüsselwort `return` gefolgt vom zurückzugebenden Wert. Der Wert selbst kann ein Ausdruck sein, der einen Wert liefert. Unter der Voraussetzung, dass die Funktion `MyFunction` selbst einen Rückgabewert liefert, handelt es sich bei den folgenden Beispielen um zulässige `return`-Anweisungen:

```
return 5;
return (x > 5);
return (MyFunction());
```

Der Wert in der zweiten Anweisung `return (x > 5)` ist `false`, wenn `x` nicht größer als 5 ist. Andernfalls lautet der Rückgabewert `true`. Es wird hier der Wert des Ausdrucks – `false` oder `true` – zurückgegeben und nicht der Wert von `x`.

Gelangt die Programmausführung zum Schlüsselwort `return`, wird der auf `return` folgende Ausdruck als Wert der Funktion zurückgegeben. Die Programmausführung kehrt dann sofort zur aufrufenden Funktion zurück, und alle auf `return` folgenden Anweisungen gelangen nicht mehr zur Ausführung.

In ein und derselben Funktion dürfen mehrere `return`-Anweisungen vorkommen. Allerdings sollte man bedenken, dass die Ausführung einer Funktion unmittelbar nach der Ausführung einer `return`-Anweisung beendet ist. Listing 5.5 zeigt dazu ein Beispiel.

Listing 5.5: manyreturns.cpp – Eine Funktion mit mehreren return-Anweisungen

```
// Listing 5.5 - Verwendung mehrerer
// return-Anweisungen
#include <iostream>

int Doubler(int AmountToDouble);

int main()
{
    int result = 0;
    int input;

    std::cout << "Zu verdoppelnde Zahl zwischen 0 und "
              << "10,000 eingeben: ";
    std::cin >> input;

    std::cout << "\nVor Aufruf von Doubler...";
    std::cout << "\nEingabe: " << input
              << " Verdoppelt: " << result << "\n";

    result = Doubler(input);

    std::cout << "\nZurueck aus Doubler...";
    std::cout << "\nEingabe: " << input
              << " Verdoppelt: " << result << "\n\n";

    return 0;
}

int Doubler(int original)
{
    if (original <= 10000)
        return original * 2;
    else
        return -1;
    std::cout << "Diese Stelle wird nie erreicht!\n";
}
```

```
Zu verdoppelnde Zahl zwischen 0 und 10000 eingeben: 9000
Vor Aufruf von Doubler...
Eingabe: 9000 Verdoppelt: 0
Zurueck aus Doubler...
Eingabe: 9000 Verdoppelt: 18000

Zu verdoppelnde Zahl zwischen 0 und 10000 eingeben: 11000
Vor Aufruf von Doubler...
Eingabe: 11000 Verdoppelt: 0
Zurueck aus Doubler...
Eingabe: 11000 Verdoppelt: -1
```

In den Zeilen 12 bis 14 wird eine Zahl angefordert und in den Zeilen 17 und 18 zusammen mit dem Ergebnis der lokalen Variablen ausgegeben. Bei Aufruf der Funktion `Doubler()`in Zeile 20 erfolgt die Übergabe des eingegebenen Wertes als Parameter. Die zu `main()` lokale Variable `result` erhält das Ergebnis dieses Aufrufs, das die Zeilen 22 bis 24 erneut anzeigen.

Zeile 31 – in der Funktion `Doubler()` – testet, ob der Parameter größer als 10000 ist. Sollte das nicht der Fall sein, gibt die Funktion das Doppelte des Originalwertes zurück. Bei einem Wert größer als 10000 liefert die Funktion –1 als Fehlerwert.

Das Programm erreicht niemals Zeile 35, da der Rücksprung aus der Funktion entweder in Zeile 32 oder in Zeile 34 – je nach übergebenem Wert – erfolgt. Ein guter Compiler erzeugt hier eine Warnung, dass diese Anweisung nicht ausgeführt werden kann, und ein guter Programmierer nimmt eine derartige Anweisung gar nicht erst auf!

Compiler-Meldungen für manyreturns.cpp

Wenn Sie mit `manyreturns.cpp` arbeiten, gibt Ihr Compiler gegebenenfalls Warnungen wie die folgenden aus (hier für C++BorlandX):

```
"manyreturns.cpp": W8066 Code wird nie erreicht in Funktion
Doubler(int) in Zeile 35

"manyreturns.cpp": W8070 Funktion sollte einen Wert zurückgeben in
Funktion Doubler(int) in Zeile 36
```

Die erste Meldung besagt, dass `std::cout` niemals zum Zuge kommt, weil alle möglichen Ausführungspfade in `return`-Anweisungen enden, bevor das Programm zur angegebenen Zeile gelangt. Und weil der Compiler mit der Zeile `std::cout` nichts anfangen kann, drückt er mit der zweiten Meldung aus, dass er ausführbaren Code ohne `return`-Anweisung vor der schließen-

den geschweiften Klammer gefunden hat. Wohlgemerkt: Es handelt sich hier um Warnungen; der Compiler kann aus Ihrem Quelltext Maschinencode erzeugen, der sich auch ausführen lässt. Möglicherweise erhalten Sie aber nicht die erwarteten Ergebnisse.

Natürlich bekommen wir in diesem Beispiel genau die Ergebnisse, auf die wir abzielen – das Programm soll ja demonstrieren, dass bestimmte Codeabschnitte nicht ausgeführt werden.

5.7 Standardparameter

Für jeden Parameter, den man im Prototyp und in der Definition einer Funktion deklariert, muss die aufrufende Funktion einen Wert übergeben, der vom deklarierten Typ sein muss. Wenn man daher eine Funktion als

```
long myFunction(int);
```

deklariert hat, muss die Funktion tatsächlich eine Integer-Variable übernehmen. Wenn die Definition der Funktion davon abweicht oder man keinen Integer-Wert übergibt, erhält man einen Compiler-Fehler.

Die einzige Ausnahme von dieser Regel: Der Prototyp der Funktion deklariert für den Parameter einen Standardwert, den die Funktion verwendet, wenn man keinen anderen Wert bereitstellt. Die obige Deklaration lässt sich wie folgt neu schreiben:

```
long myFunction (int x = 50);
```

Dieser Prototyp sagt aus: »`myFunction` gibt einen Wert vom Typ `long` zurück und übernimmt einen Integer-Parameter. Wenn kein Argument angegeben ist, verwende den Standardwert 50.« Da in Funktionsprototypen keine Parameternamen erforderlich sind, könnte man diese Deklaration auch als

```
long myFunction (int = 50);
```

schreiben. Durch die Deklaration eines Standardparameters ändert sich die Funktionsdefinition nicht. Der Kopf der Funktionsdefinition für diese Funktion lautet:

```
long myFunction (int x)
```

Wenn die aufrufende Funktion keinen Parameter einbindet, füllt der Compiler den Parameter `x` mit dem Standardwert `50`. Der Name des Standardparameters im Prototyp muss nicht mit dem Namen im Funktionskopf übereinstimmen. Die Zuweisung des Standardwertes erfolgt nach der Position in der Parameterliste und nicht nach dem Namen.

Man kann allen Parametern einer Funktion oder nur einem Teil davon Standardwerte zuweisen. Die einzige Einschränkung: Wenn für einen der Parameter kein Standardwert angegeben ist, kann kein vorheriger Parameter dieser Parameterliste einen Standardwert haben.

Sieht der Prototyp einer Funktion zum Beispiel wie folgt aus

```
long myFunction (int Param11, int Param2, int Param3);
```

kann man `Param2` nur dann einen Standardwert zuweisen, wenn man für `Param3` einen Standardwert festgelegt hat. An `Param1` lässt sich nur dann ein Standardwert zuweisen, wenn sowohl für `Param2` als auch `Param3` Standardwerte festgelegt wurden. Listing 5.6 demonstriert die Verwendung von Standardwerten.

Listing 5.6: default-parm.cpp – Standardwerte für Parameter

```
 1: // Listing 5.6 - Demonstriert die Verwendung
 2: // von Standardwerten für Parameter
 3: #include <iostream>
 4:
 5: int AreaCube(int length, int width = 25, int height = 1);
 6:
 7: int main()
 8: {
 9:     int length = 100;
10:     int width = 50;
11:     int height = 2;
12:     int area;
13:
14:     area = AreaCube(length, width, height);
15:     std::cout << "Erstes Volumen gleich: " << area << "\n";
16:
17:     area = AreaCube(length, width);
18:     std::cout << "Zweites Volumen gleich: " << area << "\n";
19:
20:     area = AreaCube(length);
21:     std::cout << "Drittes Volumen gleich: " << area << "\n";
22:     return 0;
23: }
24:
25: int AreaCube(int length, int width, int height)
26: {
27:     return (length * width * height);
28: }
```

```
Erstes Volumen gleich: 10000
Zweites Volumen gleich: 5000
Drittes Volumen gleich: 2500
```

In Zeile 5 spezifiziert der Prototyp von `AreaCube()`, dass die Funktion `AreaCube()` drei Integer-Parameter übernimmt. Die letzten beiden weisen Standardwerte auf.

Die Funktion berechnet das Volumen des Quaders für die übergebenen Abmessungen. Fehlt die Angabe der Breite (`width`), nimmt die Funktion eine Breite von 25 und eine Höhe (`height`) von 1 an. Übergibt man die Breite, aber nicht die Höhe, verwendet die Funktion eine Höhe von 1. Ohne die Übergabe der Breite kann man keine Höhe übergeben.

Länge (`length`), Höhe (`height`) und Breite (`width`) werden in den Zeilen 9 bis 11 initialisiert und in Zeile 14 an die Funktion `AreaCube()` übergeben. Nach der Berechnung der Werte gibt Zeile 15 das Ergebnis aus.

Die Programmausführung setzt mit Zeile 17 fort, wo ein weiterer Aufruf von `AreaCube()` steht, diesmal aber ohne Wert für `height`. Damit läuft die Berechnung mit dem Standardwert ab. Zeile 18 gibt das Ergebnis aus.

Nach Abarbeitung dieses Funktionsaufrufs gelangt das Programm zu Zeile 19. Im dritten Aufruf von `AreaCube()` werden weder Breite noch Höhe übergeben. Die Ausführung verzweigt nun zum dritten Mal zu Zeile 27. Die Berechnung erfolgt jetzt mit beiden Standardwerten. Zeile 22 zeigt das Ergebnis an.

5.8 Funktionen überladen

In C++ lassen sich mehrere Funktionen mit demselben Namen erzeugen. Man bezeichnet das als *Überladen* von Funktionen. Die Funktionen müssen sich in ihrer Parameterliste unterscheiden, wobei andere Parametertypen und/oder eine abweichende Anzahl von Parametern möglich sind. Dazu ein Beispiel:

```
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);
```

Die Funktion `myFunction()` wird mit drei unterschiedlichen Parameterlisten überladen. Die erste Version unterscheidet sich von der zweiten durch die Parametertypen, während die dritte Version eine abweichende Anzahl von Parametern aufweist.

Die Rückgabetypen der überladenen Funktionen können gleich oder verschieden sein, solange ein Unterschied in der Parameterliste vorhanden ist. Ein Überladen nur bezüglich des Rückgabetyps ist nicht möglich.

Das Überladen von Funktionen bezeichnet man auch als *Funktionspolymorphie*. Das aus dem Griechischen stammende Wort *polymorph* bedeutet vielgestaltig – eine polymorphe Funktion weist viele Formen auf.

Funktionspolymorphie bezieht sich auf die Fähigkeit, eine Funktion mit mehreren Bedeutungen zu »überladen«. Indem man die Anzahl oder den Typ der Parameter ändert, kann man zwei oder mehreren Funktionen denselben Funktionsnamen geben. Anhand der Parameterliste wird dann die richtige Funktion aufgerufen. Damit kann man eine Funktion erzeugen, die den Mittelwert von ganzen (`int`, `long`) und reellen (`float`, `double`) Zahlen sowie anderen Werten bilden kann, ohne dass man für jede Funktion einen separaten Namen wie `AverageInts()`, `AverageDoubles()` usw. angeben muss.

Nehmen wir eine Funktion an, die den übergebenen Wert verdoppelt. Man möchte dabei Zahlen vom Typ `int`, `long`, `float` oder `double` übergeben können. Ohne das Überladen von Funktionen müsste man vier Funktionsnamen erzeugen:

```
int DoubleInt(int);
long DoubleLong(long);
float DoubleFloat(float);
double DoubleDouble(double);
```

Mit Überladen von Funktionen lassen sich folgende Deklarationen formulieren:

```
int Double(int);
long Double(long);
float Double(float);
double Double(double);
```

Das ist leichter zu lesen und einfacher zu warten. Man braucht sich nicht darum zu kümmern, welche Funktion aufzurufen ist. Man übergibt einfach eine Variable und die richtige Funktion wird automatisch aufgerufen.

5.8.1 Inline-Funktionen

Wenn Sie eine Funktion definieren, erzeugt der Compiler normalerweise genau einen Anweisungssatz im Speicher. Beim Aufruf der Funktion springt die Programmausführung zu diesen Anweisungen und geht bei der Rückkehr aus der Funktion zur nächsten Zeile in der aufrufenden Funktion. Wenn Sie die Funktion zehnmal aufrufen, springt das Programm jedes Mal zum selben Anweisungssatz. Das bedeutet, dass es nur ein Exemplar der Funktion gibt und nicht zehn Kopien dieser Funktion.

Der Sprung in eine Funktion und das Verlassen der Funktion bedeutet für den Prozessor zusätzliche Arbeit. Gerade bei Funktionen, die nur aus einer oder zwei Zeilen Code bestehen, lässt sich die Programmausführung effizienter ge-

stalten, wenn man die erforderlichen Sprünge in die und aus der Funktion vermeiden kann. Effizienz ist für den Programmierer gewöhnlich gleichbedeutend mit Geschwindigkeit: das Programm läuft ohne den Funktionsaufruf schneller.

Deklariert man eine Funktion mit dem Schlüsselwort `inline`, erzeugt der Compiler keine Funktion im engeren Sinne, sondern kopiert den Code für die Inline-Funktion direkt in die aufrufende Funktion. Damit fallen die Sprünge weg, und das Ganze stellt sich genauso dar, als hätte man die Anweisungen der Funktion selbst in die aufrufende Funktion geschrieben.

Allerdings ist der Geschwindigkeitsvorteil nicht umsonst zu haben. An allen Stellen, an denen Sie die Inline-Funktion aufrufen, erscheint der vollständige Code der Funktion, bei zehn Aufrufen eben zehn Mal. Bei umfangreichen Programmen wird somit die ausführbare Datei wesentlich größer. Dennoch kann der Geschwindigkeitszuwachs nur marginal sein. Die heutigen optimierenden Compiler arbeiten intelligent genug, sodass Sie aus der Deklaration einer Funktion als `inline` fast nie einen großen Gewinn schlagen. Wichtiger ist aber, dass die Vergrößerung des Codes seinerseits zulasten der Gesamtleistung gehen kann.

inline anstelle von #define

Inline-Funktionen treten an die Stelle der `#define`-Makros der Sprache C.

Als Faustregel gilt, dass eine kleine Funktion mit einer oder zwei Anweisungen für eine Inline-Deklaration infrage kommt. Wenn Sie allerdings im Zweifel sind, verzichten Sie einfach darauf. Listing 5.7 demonstriert den Einsatz einer Inline-Funktion.

Listing 5.7: inliner.cpp – Eine Inline-Funktion

```
1: // Listing 5.7 - Zeigt den Einsatz von Inline-Funktionen
2: #include <iostream>
3:
4: inline int Doubler(int);
5:
6: int main()
7: {
8:     int target;
9:
10:    std::cout << "Bitte eine Zahl eingeben: ";
11:    std::cin >> target;
12:    std::cout << "\n";
13:
14:    target = Doubler(target);
15:    std::cout << "Ziel: " << target << std::endl;
16:
```

```
17:     target = Doubler(target);
18:     std::cout << "Ziel: " << target << std::endl;
19:
20:     target = Doubler(target);
21:     std::cout << "Ziel: " << target << std::endl;
22:     return 0;
23: }
24:
25: int Doubler(int target)
26: {
27:     return 2*target;
28: }
```

```
Bitte eine Zahl eingeben: 20

Ziel: 40
Ziel: 80
Ziel: 160
```

Zeile 4 deklariert die Inline-Funktion `Doubler()`, die einen Parameter vom Typ `int` übernimmt und einen Wert des gleichen Typs zurückgibt. Die Deklaration sieht wie bei jedem anderen Prototyp aus, außer dass das Schlüsselwort `inline` unmittelbar vor dem Rückgabetyp steht.

Der Compiler erzeugt daraus den gleichen Code, als hätten Sie die Anweisung

```
target = 2 * target;
```

an allen Stellen eingegeben, an denen im obigen Listing

```
target = Doubler(target);
```

steht.

Bei Programmausführung befinden sich die Anweisungen bereits an Ort und Stelle, d.h. der Compiler hat sie in die Objektdatei eingebaut (sprich: kompiliert). Das spart zwar einen Sprung zum Aufruf der Funktion, geht aber zulasten der Programmgröße.

Nur eine Empfehlung

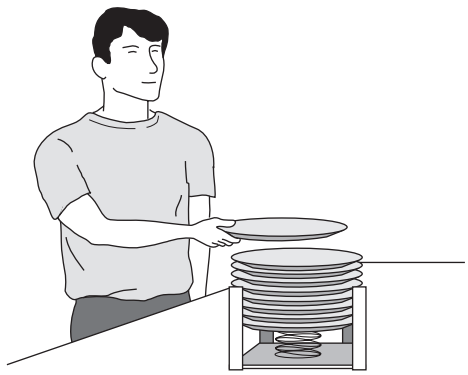
Das Schlüsselwort `inline` ist ein Hinweis an den Compiler, dass Sie die Funktion als `inline` deklarieren möchten. Dem Compiler steht es aber frei, den Hinweis zu ignorieren und einen echten Funktionsaufruf zu generieren.

5.8.2 Stack und Funktionen

Beim Aufruf einer Funktion verzweigt der Code in die aufgerufene Funktion, die Parameter werden übergeben und der Rumpf der Funktion wird ausgeführt. Nach abgeschlossener Abarbeitung gibt die Funktion einen Wert zurück (außer wenn die Funktion als `void` deklariert ist) und die Steuerung geht an die aufrufende Funktion über. Wie funktioniert das Ganze?

Beim Start eines Programms legt der Compiler einen speziellen Bereich im Hauptspeicher an: den so genannten *Stack* (Stapelspeicher). Der Stack nimmt Daten auf, die für die einzelnen Funktionen im Programm gedacht sind. Die Bezeichnung Stapelspeicher lässt Ähnlichkeiten mit einem Geschirrstapel vermuten, wie ihn Abbildung 5.4 zeigt. Was man zuletzt auf den Stapel gelegt hat, entnimmt man auch zuerst wieder.

Abbildung 5.4: Ein Stack (Stapel)



Der Stack wächst, wenn man Daten auf den Stack »legt«. Entnimmt man Daten vom Stack, schrumpft er. Einen Geschirrstapel kann man auch nicht wegräumen, ohne die zuletzt obenauf gelegten Teller als erste wieder wegzunehmen.

Die Analogie zum Geschirrstapel eignet sich zwar zur anschaulichen Darstellung, versagt aber bei der grundlegenden Arbeitsweise des Stacks. Eine genauere Vorstellung liefert eine Folge von Fächern, die von oben nach unten angeordnet sind. Auf der Spitze des Stacks befindet sich das Fach, auf das der Stack-Pointer (Zeiger auf den Stack, ein spezielles Register des Prozessors) zeigt.

Alle Fächer haben eine fortlaufende Adresse und eine dieser Adressen wird im Stack-Register abgelegt. Alles unterhalb dieser magischen Adresse, die man als Spitze des Stacks bezeichnet, wird als »auf dem Stack befindlich« betrachtet. Alles oberhalb des Stack-Pointers liegt außerhalb des Stacks und ist ungültig. Abbildung 5.5 verdeutlicht dieses Konzept.

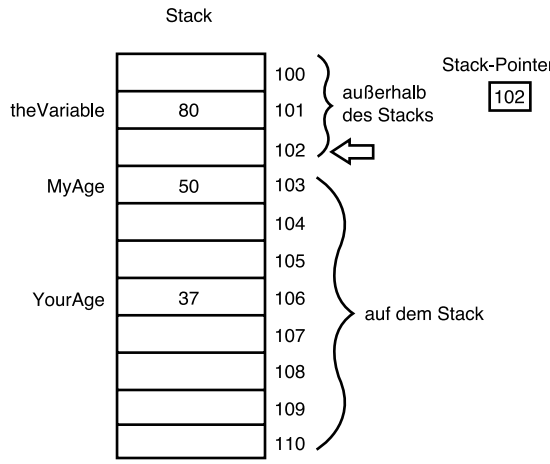


Abbildung 5.5: Der Stack-Pointer

Wenn man Daten auf den Stack legt, kommen sie in ein Fach oberhalb des Stack-Pointers. Anschließend wird der Stack-Pointer zu den neuen Daten verschoben. Entnimmt man Daten aus dem Stack, passiert weiter nichts, als dass die Adresse des Stack-Pointers auf dem Stack nach unten geschoben wird. Abbildung 5.6 macht diese Regel deutlich.

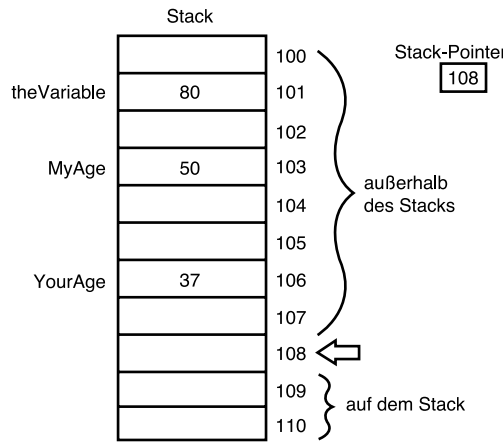


Abbildung 5.6: Verschieben des Stack-Pointers

Ruft ein Programm eine Funktion auf, richtet es einen so genannten »Stack-Rahmen« ein. Dabei handelt es sich um einen Bereich auf dem Stack, den das Programm für die Verwaltung der aufgerufenen Funktion reserviert. Das Ganze ist ziemlich kompliziert und wird noch dazu auf verschiedenen Computern unterschiedlich realisiert. Die wesentlichen Schritte sehen so aus:

1. Die Rückkehradresse der Funktion wird auf dem Stack abgelegt. Wenn die Funktion zurückkehrt, setzt die Programmausführung an dieser Adresse fort.
2. Auf dem Stack wird für den deklarierten Rückgabetyp Platz geschaffen.
3. Alle Argumente an die Funktion werden auf dem Stack platziert.
4. Das Programm verzweigt in die Funktion.
5. Die Funktion legt lokale Variablen auf dem Stack ab, wenn sie definiert werden.

Beim Rücksprung aus der Funktion laufen die Vorgänge in der umgekehrten Reihenfolge ab.

5.9 Fragen und Antworten

F: Warum arbeitet man nicht generell mit globalen Variablen?

A: In den Anfangszeiten der Programmierung wurde genau das gemacht. Durch die zunehmende Komplexität der Programme ließen sich Fehler immer schwerer finden, da jede beliebige Funktion die Daten zerstören konnte – globale Daten lassen sich an beliebigen Stellen im Programm verändern. Im Lauf der Jahre hat sich bei den Programmierern die Erkenntnis durchgesetzt, dass Daten so lokal wie möglich zu halten sind und der Zugriff auf diese Daten sehr eng abgesteckt sein sollte.

F: Warum werden Änderungen am Wert von Funktionsargumenten nicht in der aufrufenden Funktion widergespiegelt?

A: Die Übergabe der Argumente an eine Funktion erfolgt als Wert. Das bedeutet, dass die Argumente innerhalb der Funktion tatsächlich als Kopien der Originalwerte vorliegen.

F: Was passiert, wenn ich die folgenden beiden Funktionen in ein und demselben Programm deklariere?

```
int Area (int width, int length = 1);
int Area (int size);
```

Werden sie überladen? Die Anzahl der Parameter ist unterschiedlich, aber der erste Parameter hat einen Standardwert.

A: Die Deklarationen werden zwar kompiliert, wenn man aber `Area` mit einem Parameter aufruft, erhält man einen Fehler zur Kompilierzeit:

```
... Mehrdeutigkeit zwischen 'Area(int,int)' und 'Area(int)'...
```

5.10 Workshop

Nachdem Sie sich mit Funktionen beschäftigt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zum Compiler zu festigen.

5.10.1 Kontrollfragen

1. Nach welchem Standardmechanismus werden Variablen an eine aufgerufene Funktion übergeben?
2. Woher weiß C++, welche Version einer überladenen Funktion aufzurufen ist?
3. Weshalb sind Funktionen nützlich? Warum schreibt man nicht ein großes Programm aus einem Stück?
4. Wie viele Werte kann eine `return`-Anweisung zurückgeben?

5.10.2 Übungen

1. Kommentieren Sie im Programm nach Listing 5.5 (`manyreturns.cpp`) die letzte `std::cout`-Anweisung aus. Was passiert, wenn Sie das Programm jetzt kompilieren? Was würde passieren, wenn Sie eine `return`-Anweisung nach dieser `std::cout`-Anweisung eingebaut hätten?
2. Wie könnten Sie Listing 5.2 (`localvar.cpp`) erweitern, um Temperaturen in Grad Fahrenheit für Grad Celsius zurückzugeben? Probieren Sie es aus!
3. Erweitern Sie Listing 5.2 (`localvar.cpp`), um Temperaturen entweder für ganzzahlige Werte oder für Gleitpunktwerte zu berechnen. Verwenden Sie hierfür überladene Funktionen.

5.10.3 Antworten zu den Kontrollfragen

1. Übergabe als Wert. Es wird nur der Wert (eine Kopie der Originalvariablen) übergeben. Dadurch kann die Funktion den Originalwert nicht unbeabsichtigt ändern.
2. Durch die Argumente, mit denen Sie die Funktion aufrufen. Der Compiler vergleicht sie mit den möglichen Definitionen und kann so die richtige Version ermitteln.
3. Funktionen erlauben es, bereits vorhandenen Code (entweder Ihren eigenen oder den von anderen) wiederzuverwenden und große Problemstellungen in kleinere Teile zu gliedern. Außerdem verringern sie den Wartungsaufwand und verkürzen die Entwicklungszeit.

4. Nur einen. Dieser kann das Ergebnis eines komplexen Ausdrucks sein, der Aufrufer erhält aber nur einen Ergebniswert zurück. Mit unterschiedlichen `return`-Anweisungen lassen sich unterschiedliche Werte oder Ergebnisse zurückgeben, trotzdem gibt eine Funktion immer nur einen einzigen Wert zurück.

KAPITEL 6

jetzt lerne ich

Programmsteuerung

In dieser Stunde lernen Sie,

- was Schleifen sind und wie man sie einsetzt,
- wie man verschiedene Schleifen konstruiert,
- wie man eine Alternativkonstruktion zu tief verschachtelten `if...else`-Anweisungen aufbaut.

6.1 Schleifenkonstruktionen

Die vorletzte Stunde hat eine erste Form für die Änderung des Programmflusses behandelt: `if` und `else`. Viele Probleme der Programmierung lassen sich durch wiederholtes Ausführen einer oder mehrerer Anweisungen lösen.

Iteration bedeutet die wiederholte Ausführung einer Aktion. Die Hauptmethode der Iteration ist die Schleife.

6.1.1 Ursprung der Schleifen – Konstruktionen mit goto

In den frühen Tagen der Informatik bestanden Schleifen aus einer Sprungmarke (Label), einigen Anweisungen und einem Sprung.

In C++ ist ein Label einfach ein Name gefolgt von einem Doppelpunkt. Das Label steht links neben einer zulässigen C++-Anweisung. Einen Sprung realisiert man mit der Anweisung `goto` und dem sich anschließenden Namen des Labels.

6.1.2 Warum man goto nicht verwendet

In C++ verzichtet man in der Regel aus gutem Grund auf `goto`-Konstruktionen. Mit `goto`-Anweisungen lässt sich ein Sprung zu einer beliebigen Stelle im Quellcode realisieren, rückwärts oder vorwärts. Die unüberlegte Verwendung von `goto`-Anweisungen führt zu unübersichtlichen, schlechten und schwer zu lesenden Programmen, die man als *Spagetti-Code* bezeichnet.

Der strukturierten Programmierung kommen die intelligenteren Schleifenbefehle `for`, `while` und `do...while` entgegen, mit denen man `goto`-Konstruktionen von vornherein vermeiden und verständlichere Programme schreiben kann. Man vermeidet `goto` fast aus den gleichen Gründen wie globale Variablen: zu wenig Kontrolle und zu schwierig zu debuggen.

6.2 while-Schleifen

Eine `while`-Schleife bewirkt die Wiederholung einer Folge von Anweisungen im Programm, solange die Startbedingung gleich `true` bleibt. Listing 6.1 zeigt ein Beispiel.

Listing 6.1: while.cpp – while-Schleifen

```
 1: // Listing 6.1
 2: // Schleifen mit while
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     int counter = 0;            // Bedingung initialisieren
 8:
 9:     while(counter < 5)          // Testen, ob Bedingung immer
noch true
10:     {
11:         counter++;              // Rumpf der Schleife
12:         std::cout << "counter: " << counter << "\n";
13:     }
14:
15:     std::cout << "Fertig. counter: " << counter << ".\n";
16:     return 0;
17: }
```

```
counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
Fertig. counter: 5.
```

Dieses einfache Programm demonstriert die Grundlagen der `while`-Schleife. Liefert der Test einer Bedingung den Wert `true`, wird der Rumpf der `while`-Schleife ausgeführt. Im Beispiel testet die Bedingung in Zeile 9, ob `counter` kleiner als `5` ist. Ergibt der Test `true`, führt das Programm den Rumpf der Schleife aus: Zeile 11 inkrementiert den Zähler, und Zeile 12 gibt den Wert aus. Wenn die Bedingungsanweisung in Zeile 9 den Wert `false` ergibt (wenn `counter` nicht mehr kleiner als `5` ist), wird der gesamte Rumpf der `while`-Schleife (in den Zeilen 10 bis 13) übersprungen. Die Programmausführung setzt dann sofort mit Zeile 15 fort.

6.2.1 Komplexere while-Anweisungen

Die in einer `while`-Schleife getestete Bedingung kann aus jedem zulässigen C++-Ausdruck bestehen. Dazu gehören auch Ausdrücke mit den logischen Operatoren `&&` (AND), `||` (OR) und `!` (NOT). Listing 6.2 zeigt eine etwas kompliziertere `while`-Anweisung.

Listing 6.2: complex-while.cpp – Komplexe while-Schleifen

```
 1: // Listing 6.2
 2: // Komplexe while-Anweisungen
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     unsigned short small;
 8:     unsigned long  large;
 9:     const unsigned short MAXSMALL=65535;
10:
11:     std::cout << "Bitte eine kleine Zahl eingeben: ";
12:     std::cin >> small;
13:     std::cout << "Bitte eine grosse Zahl eingeben: ";
14:     std::cin >> large;
15:
16:     std::cout << "Klein: " << small << "...";
17:
18:      // Bei jedem Schleifendurchlauf drei Bedingungen testen
19:     while (small < large && large > 0 && small < MAXSMALL)
20:     {
21:         if (small % 5000 == 0)  // Alle 5000 Zeilen einen Punkt
ausgeben
22:             std::cout << ".";
23:
24:         small++;
25:
26:         large-=2;
27:     }
28:
```

```
29:     std::cout << "\nKlein: " << small
30:               << " Gross: "  << large << std::endl;
31:     return 0;
32: }
```

```
Bitte eine kleine Zahl eingeben: 2
Bitte eine grosse Zahl eingeben: 100000
Klein: 2.........
Klein: 33335 Gross: 33334
```

Dieses Programm stellt ein Spiel dar. Man gibt zwei Zahlen ein, eine kleine (`small`) und eine große (`large`). Die kleinere Zahl wird um 1 nach oben gezählt, die größere in Schritten von 2 abwärts. Es ist nun zu erraten, wann sich die Zahlen treffen.

In den Zeilen 11 bis 14 erfolgt die Eingabe der Zahlen. Zeile 19 richtet eine `while`-Schleife ein, deren Durchläufe von drei Bedingungen abhängig sind:

- `small` ist nicht größer als `large`
- `large` ist nicht negativ
- `small` überschreitet nicht die Größe einer kleinen Integer-Zahl (`MAXSMALL`)

Zeile 21 berechnet den Wert in `small` modulo `5000`. Der Wert in `small` bleibt dabei unverändert. Der Ausdruck liefert das Ergebnis `0`, wenn `small` ein genaues Vielfaches von 5000 ist. In diesem Fall gibt das Programm als Fortschrittskontrolle einen Punkt auf dem Bildschirm aus. Zeile 24 inkrementiert den Wert von `small`, während Zeile 26 den Wert von `large` um 2 dekrementiert.

Wenn irgendeine der drei Bedingungen in der `while`-Schleife nicht erfüllt ist, endet die Schleife und die Ausführung des Programms setzt sich nach der schließenden Klammer der `while`-Schleife in Zeile 29 fort. Möglicherweise erscheint eine Warnung wie die folgende, weil null eine vorzeichenbehaftete Konstante ist:

```
"complexwhile.cpp": W8012 Vergleich von signed- und unsigned-Werten
in Funktion main() in Zeile 19
```

Um eine numerische Konstante explizit als vorzeichenlos zu kennzeichnen, hängen Sie ein `u` als Suffix an – im Beispiel `0u`.

6.2.2 Die Anweisungen continue und break

Manchmal soll das Programm an den Anfang einer `while`-Schleife zurückkehren, bevor die gesamte Gruppe von Anweisungen in der `while`-Schleife abgearbeitet ist. Die `continue`-Anweisung bewirkt einen Sprung zurück an den Beginn der Schleife.

Es kann auch sein, dass man die Schleife verlassen muss, bevor die Ende-Bedingung erfüllt ist. Die `break`-Anweisung führt unmittelbar zum Austritt aus der `while`-Schleife und die Programmausführung setzt sich nach der schließenden Klammer fort.

Listing 6.3 demonstriert die Verwendung dieser Anweisungen. Dieses Mal ist das Spiel etwas komplizierter. Der Benutzer wird eingeladen, eine kleine Zahl (`small`), eine große Zahl (`large`), eine Sprungzahl (`skip`) und eine Zielzahl (`target`) einzugeben. Das Programm inkrementiert die Zahl `small` um 1 und dekrementiert die Zahl `large` um 2. Das Dekrementieren wird übersprungen, wenn `small` ein Vielfaches der Zahl `skip` ist. Das Spiel endet, sobald `small` größer als `large` ist. Wenn die Zahl `large` genau die Zielzahl `target` trifft, erscheint eine Mitteilung und das Spiel stoppt.

Der Benutzer muss versuchen, eine Zielzahl für die Zahl `large` einzugeben, die das Spiel stoppt.

Listing 6.3: breaker.cpp – Die Anweisungen break und continue

```
 1: // Listing 6.3
 2: // Demonstriert die Anweisungen break und continue
 3: #include <iostream>
 4: using namespace std;// Diese Datei verwendet std::cout,
 5:                     // std::cin, std::endl usw.
 6:
 7: int main()
 8: {
 9:     unsigned short small;
10:     unsigned long  large;
11:     unsigned long  skip;
12:     unsigned long target;
13:     const unsigned short MAXSMALL=65535;
14:
15:     cout << "Bitte eine kleine Zahl eingeben: ";
16:     cin >> small;
17:     cout << "Bitte eine grosse Zahl eingeben: ";
18:     cin >> large;
19:     cout << "Bitte eine Sprungzahl eingeben: ";
20:     cin >> skip;
21:     cout << "Bitte eine Zielzahl eingeben: ";
22:     cin >> target;
23:
24:     cout << "\n";
25:
```

```
26:     // Drei Abbruchbedingungen für die Schleife einrichten
27:     while (small < large && large > 0 && small < MAXSMALL)
28:     {
29:         small++;
30:
31:         if (small % skip == 0) // Dekrementieren überspringen?
32:         {
33:             cout << "Ueberspringen " << small << endl;
34:             continue;
35:         }
36:
37:         if (large == target)   // target (Ziel) genau getroffen?
38:         {
39:             cout << "Ziel getroffen!";
40:             break;
41:         }
42:
43:         large-=2;
44:     }                          // Ende der while-Schleife
45:
46:     cout << "\nKlein: " << small << " Gross: " << large << endl;
47:     return 0;
48: }
```

```
Bitte eine kleine Zahl eingeben: 2
Bitte eine grosse Zahl eingeben: 20
Bitte eine Sprungzahl eingeben: 4
Bitte eine Zielzahl eingeben: 6
Ueberspringen 4
Ueberspringen 8
Klein: 10 Gross: 8
```

In diesem Spiel hat der Benutzer verloren. `small` wurde größer als `large`, bevor er die `target`-Zahl von 6 erreicht hat.

In Zeile 27 steht der Test der `while`-Bedingungen. Wenn `small` weiterhin kleiner als `large` ist, `large` größer als 0 ist und `small` noch nicht den Maximalwert für eine kleine `int`-Zahl überschritten hat, tritt die Programmausführung in den Rumpf der `while`-Schleife ein.

Die Anweisung in Zeile 31 berechnet den Rest der Ganzzahldivision von `small` und `skip`. Wenn `small` ein Vielfaches von `skip` ist, wird die `continue`-Anweisung erreicht und die Programmausführung springt an den Beginn der Schleife in Zeile 27. Damit übergeht das Programm den Test auf `target` und das Dekrementieren von `large`.

Zeile 37 testet `target` erneut gegen den Wert für `large`. Sind beide Werte gleich, hat der Benutzer gewonnen. Es erscheint eine Meldung und das Programm erreicht die `break`-Anweisung. Das bewirkt einen sofortigen Austritt aus der `while`-Schleife und die Programmausführung setzt sich nach Zeile 46 fort.

Sowohl `continue` als auch `break` sollte man mit Umsicht einsetzen. Programme, die mitten in einer Schleife plötzlich »die Richtung ändern«, sind schwer zu verstehen, und der großzügige Einsatz von `continue` und `break` macht sogar eine kleine `while`-Schleifenkonstruktion unverständlich.

Das Schlüsselwort using namespace

Das Codebeispiel in Listing 6.3 führt ein neues Schlüsselwort ein: `using namespace`. Es sagt dem Compiler, dass er bei einem Funktionsaufruf (wie zum Beispiel `cin` oder `cout`) das Standardpräfix `std::` annehmen soll. Damit verringert sich der Schreibaufwand beim Eintippen des Quelltextes.

Da viele Compiler diesen Befehl noch nicht unterstützen, verzichten die meisten Beispiele in diesem Buch darauf. Der Borland-Compiler unterstützt ihn.

6.2.3 while(1)-Schleifen

Bei der in einer `while`-Schleife getesteten Bedingung kann es sich um einen beliebigen gültigen C++-Ausdruck handeln. Solange die Bedingung `true` bleibt, setzt sich die Abarbeitung der `while`-Schleife fort. Man kann eine Endlosschleife erzeugen, indem man für die zu testende Bedingung die Zahl `1` verwendet. Da `1` immer `true` ist, bricht die Schleife nur dann ab, wenn eine `break`-Anweisung auszuführen ist. Listing 6.4 realisiert mithilfe dieser Konstruktion einen Zähler bis 10.

Listing 6.4: whileforever.cpp – while(1)-Schleifen

```
 1: // Listing 6.4
 2: // Demonstriert eine while(true)-Schleife
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     int counter = 0;
 8:
 9:     while (1)
10:     {
11:         counter ++;
12:         if (counter > 10)
13:             break;
```

```
14:     }
15:     std::cout << "counter: " << counter << "\n";
16:     return 0;
17: }
```

```
counter: 11
```

Zeile 9 richtet eine `while`-Schleife mit einer Bedingung ein, die niemals `false` liefern kann. Die Schleife inkrementiert die Zählervariable (`counter`) in Zeile 11 und testet dann in Zeile 12, ob `counter` den Wert 10 überschritten hat. Ist das nicht der Fall, führt die Schleife einen erneuten Durchlauf aus. Wenn `counter` größer als 10 ist, beendet die `break`-Anweisung in Zeile 13 die `while`-Schleife und die Programmausführung geht direkt zu Zeile 15, wo die Ausgabe der Ergebnisse stattfindet.

Dieses Programm funktioniert zwar, ist aber nicht sehr elegant formuliert – ein gutes Beispiel für das falsche Werkzeug. Das Gleiche kann man realisieren, wenn man den Test des Zählerwertes dorthin schreibt, wo er hingehört: in die `while`-Bedingung.

Endlosschleifen

Endlosschleifen wie `while(1)` können dazu führen, dass sich der Computer »aufhängt«, wenn die Ende-Bedingung niemals erreicht wird. Verwenden Sie diese Konstruktion mit Vorsicht und führen Sie gründliche Tests durch.

C++ bietet verschiedene Möglichkeiten, um die gleiche Aufgabe zu realisieren. Das eigentliche Kunststück ist es, das richtige Werkzeug für die jeweilige Aufgabe herauszusuchen.

Was Sie tun sollten

Arbeiten Sie mit `while`-Schleifen, um Iterationen auszuführen, solange eine Bedingung `true` ist.

Setzen Sie die Anweisungen `continue` und `break` mit Umsicht ein.

Stellen Sie sicher, dass es einen Austritt aus der Schleife gibt.

Was nicht

Schreiben Sie kein Semikolon unmittelbar nach der `while()`-Anweisung, weil es die Anweisung abschließt und die Schleife keinen Rumpf hat (sie führt keinerlei Aktionen aus).

6.3 do...while-Schleifen

Es kann sein, dass der Rumpf einer `while`-Schleife gar nicht zur Ausführung gelangt. Die `while`-Anweisung prüft die Bedingung vor allen Anweisungen. Liefert die Bedingung `false`, überspringt das Programm den gesamten Rumpf der Schleife. Listing 6.5 verdeutlicht das.

Listing 6.5: bodyskip.cpp – Den Rumpf der while-Schleife überspringen

```
 1: // Listing 6.5
 2: // Den Rumpf der while-Schleife überspringen, wenn
 3: // die Bedingung false ist.
 4: #include <iostream>
 5:
 6: int main()
 7: {
 8:     int counter;
 9:     std::cout << "Wie viele Hellos?: ";
10:     std::cin >> counter;
11:     while (counter > 0)
12:     {
13:         std::cout << "Hello!\n";
14:         counter--;
15:     }
16:     std::cout << "counter: " << counter;
17:     return 0;
18: }
```

```
Wie viele Hellos?: 2
Hello!
Hello!
counter: 0
```

Führen Sie dieses Programm ein zweites Mal aus und geben Sie `0` ein. Das sieht dann so aus:

```
Wie viele Hellos?: 0
counter: 0
```

Die Anweisung in Zeile 9 fordert den Benutzer zur Eingabe eines Startwertes auf, der in der Integer-Variablen `counter` gespeichert wird. Zeile 11 führt einen Test des Wertes in `counter` durch und dekrementiert ihn im Rumpf der `while`-Schleife. Beim ersten Programmdurchlauf hat der Benutzer für `counter` die Zahl 2 eingegeben, sodass die Schleife zweimal ausgeführt wurde. Beim zweiten Mal hat er allerdings eine `0` eingegeben. Der Test von `counter` in Zeile 11 liefert die Bedingung `false`, da `counter` nicht größer als `0` ist. Damit überspringt das Programm die gesamte `while`-Schleife und die Meldung »Hello!« erscheint überhaupt nicht.

Wie kann man nun sicherstellen, dass »Hello!« wenigstens einmal zu sehen ist? Die `while`-Schleife kann das nicht realisieren, da der Test der Bedingung vor jeglicher Ausgabe erfolgt. Man kann die Schleifenausführung erzwingen, indem man eine `if`-Anweisung unmittelbar vor dem Eintritt in die Schleife einbaut:

```
if (counter < 1)  // Einen Minimalwert erzwingen
counter = 1;
```

Das ist aber genau das, was der Programmierer als »Kludge« bezeichnet, eine hässliche und wenig elegante Lösung.

Die `do...while`-Schleife führt den Rumpf der Schleife aus, bevor der Test der Bedingung stattfindet. Damit ist gesichert, dass der Rumpf mindestens einmal abgearbeitet wird. Listing 6.6 zeigt das umgeschriebene Programm mit einer `do...while`-Schleife.

Listing 6.6: dowhile.cpp – Demonstration einer do...while-Schleife

```
 1: // Listing 6.6
 2: // Demonstriert eine do...while-Schleife
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     int counter;
 8:     std::cout << "Wie viele Hellos? ";
 9:     std::cin >> counter;
10:     do
11:     {
12:         std::cout << "Hello\n";
13:         counter--;
14:     } while (counter >0 );
15:     std::cout << "counter: " << counter << std::endl;
16:     return 0;
17: }
```

```
Wie viele Hellos? 2
Hello
Hello
counter: 0
```

Führen Sie dieses Programm ein zweites Mal aus und geben Sie `0` ein. Jetzt erhalten Sie:

```
Wie viele Hellos?: 0
Hello
counter: -1
```

Die Anweisung in Zeile 8 fordert den Benutzer zur Eingabe eines Startwertes auf, der in der Integer-Variablen `counter` gespeichert wird. Das Programm tritt in die `do...while`-Schleife ein, bevor der Test der Bedingung erfolgt. Damit ist die Ausführung der Schleife mindestens einmal garantiert. In Zeile 12 steht die Ausgabe der Meldung, Zeile 13 dekrementiert den Zähler und in Zeile 14 findet der Test der Bedingung statt. Wenn die Bedingung den Wert `true` ergibt, springt die Ausführung an den Anfang der Schleife in Zeile 12, andernfalls geht es direkt zu Zeile 15.

Die Anweisungen `continue` und `break` arbeiten in der `do...while`-Schleife genauso wie in der `while`-Schleife. Der einzige Unterschied zwischen einer `while`- und einer `do...while`-Schleife besteht im Zeitpunkt für den Test der Bedingung.

6.4 for-Schleifen

Bei der Programmierung von `while`-Schleifen stellt man häufig fest, dass man eine Startbedingung festlegt, eine Bedingung auf `true` testet und eine Variable bei jedem Schleifendurchlauf inkrementiert oder anderweitig verändert. Listing 6.7 zeigt dazu ein Beispiel.

Listing 6.7: whileagain.cpp – Untersuchung einer while-Schleife

```
 1: // Listing 6.7
 2: // Schleifendurchläufe mit while
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     int counter = 0;
 8:
 9:     while(counter < 5)
10:     {
11:         counter++;
```

```
12:         std::cout << "Schleife!  ";
13:     }
14:
15:     std::cout << "\nCounter: " << counter << ".\n";
16:     return 0;
17: }
```

```
Schleife!  Schleife!  Schleife!  Schleife!  Schleife!
Counter: 5.
```

Zeile 7 initialisiert die Variable `counter` für die Anfangsbedingung mit `0`. Der Test in Zeile 9 prüft, ob `counter` kleiner als `5` ist. In Zeile 11 wird `counter` inkrementiert. Zeile 12 gibt nur eine einfache Meldung aus – hier sind aber auch wichtigere Aufgaben denkbar, die bei jedem Inkrementieren von `counter` zu erledigen sind.

6.4.1 Initialisierung, Test und Inkrementierung

Eine `for`-Schleife fasst die drei Schritte Initialisierung, Test und Inkrementierung in einer Anweisung zusammen. Die Syntax der `for`-Anweisung besteht aus dem Schlüsselwort `for` gefolgt von einem Klammernpaar. Innerhalb dieser Klammern befinden sich drei durch Semikolons getrennte Anweisungen.

Die erste Anweisung ist die Initialisierung. Hier kann man jede zulässige C++-Anweisung angeben. Normalerweise verwendet man diese Anweisung aber, um eine Zählervariable zu erzeugen und zu initialisieren. Die zweite Anweisung realisiert den Test und kann ebenfalls jeder zulässige C++-Ausdruck sein. Diese Anweisung übernimmt die Rolle der Bedingung in der `while`-Schleife. Anweisung drei ist die Aktion. Normalerweise inkrementiert oder dekrementiert man einen Wert, obwohl auch alle zulässigen C++-Anweisungen möglich sind. Beachten Sie, dass zwar die erste und dritte Anweisung alle in C++ zulässigen Anweisungen sein können, dass aber für die zweite Anweisung ein Ausdruck erforderlich ist – eine C++-Anweisung, die einen Wert zurückgibt. Listing 6.8 demonstriert die `for`-Schleife.

Listing 6.8: forloop.cpp – Beispiel für eine for-Schleife

```
 1: // Listing 6.8
 2: // Schleifendurchläufe mit for
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     int counter;
 8:     for (counter = 0; counter < 5; counter++)
 9:         std::cout << "Schleife! ";
10:
```

```
11:     std::cout << "\nCounter: " << counter << ".\n";
12:     return 0;
13: }
```

```
Schleife!  Schleife!  Schleife!  Schleife!  Schleife!
Counter: 5.
```

Die for-Anweisung in Zeile 8 kombiniert die Initialisierung von counter, den Test, ob counter kleiner als 5 ist, und die Inkrementierung von counter in ein und derselben Zeile. Der Rumpf der for-Anweisung steht in Zeile 9. Natürlich könnte man hier genauso gut einen Block vorsehen.

Fehlerquellen

Ein häufiger Fehler ist es, innerhalb der for-Anweisung ein Komma anstelle eines Semikolons zu schreiben. Der Compiler gibt hier eine Fehlermeldung aus.

Zu den üblichen Fehlern gehört auch, ein Semikolon nach den Klammern der for-Anweisung zu setzen. Dadurch entsteht eine Schleife, die keine Aktionen im Rumpf ausführt. Manchmal ist das gewollt, sodass der Compiler in diesem Fall keinen Fehler meldet.

6.4.2 Erweiterte for-Schleifen

for-Anweisungen sind leistungsfähig und flexibel. Die drei unabhängigen Anweisungen (Initialisierung, Test und Aktion) führen von selbst zu einer Reihe von Varianten.

Eine for-Schleife arbeitet nach folgendem Schema:

1. Operationen in der Initialisierung ausführen.
2. Bedingung auswerten.
3. Wenn die Bedingung true ergibt, zuerst Schleife und dann Aktionsanweisung ausführen.

Nach jedem Schleifendurchlauf wiederholt die for-Schleife die Schritte 2 und 3.

Mehrfache Initialisierung und Inkrementierung

Es ist durchaus üblich, mehrere Variablen auf einmal zu initialisieren, einen zusammengesetzten logischen Ausdruck zu testen und mehrere Anweisungen auszuführen. Die Anweisungen für Initialisierung und Aktion lassen sich durch

mehrere C++-Anweisungen ersetzen, die jeweils durch Komma zu trennen sind. Listing 6.9 zeigt die Initialisierung und Inkrementierung von zwei Variablen.

Listing 6.9: formulti.cpp – Mehrere Anweisungen in for-Schleifen

```
 1: // Listing 6.9
 2: // Demonstriert mehrere Anweisungen in
 3: // for-Schleifen
 4: #include <iostream>
 5:
 6: int main()
 7: {
 8:     for (int i=0, j=0; i<3; i++, j++)
 9:         std::cout << "i: " << i << " j: " << j << std::endl;
10:     return 0;
11: }
```

```
i: 0  j: 0
i: 1  j: 1
i: 2  j: 2
```

Zeile 8 initialisiert die beiden Variablen `i` und `j` mit dem Wert `0`. Die Auswertung der Testbedingung (`i<3`) liefert `true`. Somit führt die `for`-Konstruktion die Anweisungen im Rumpf aus: hier die Ausgabe des Wertes. Schließlich wird die dritte Klausel in der `for`-Anweisung ausgeführt: Inkrementieren von `i` und `j`.

Nach Abarbeitung von Zeile 9 wertet die `for`-Konstruktion die Bedingung erneut aus. Liefert die Auswertung weiterhin `true`, wiederholt die `for`-Schleife die Aktionen (Inkrementieren von `i` und `j`) und führt den Rumpf der Schleife erneut aus. Das setzt sich so lange fort, bis der Test `false` ergibt. Die Aktionsanweisung gelangt dann nicht mehr zur Ausführung und der Programmablauf setzt sich nach der Schleife fort.

Leere Anweisungen in for-Schleifen

In einer `for`-Schleife können einige oder alle Anweisungen leer sein. Dazu markiert man mit einem Semikolon die Stelle, wo die Anweisung normalerweise steht. Um eine `for`-Schleife zu erzeugen, die genau wie eine `while`-Schleife arbeitet, lässt man die ersten und dritten Anweisungen weg. Dazu zeigt Listing 6.10 ein Beispiel.

Listing 6.10: fornull.cpp – Eine for-Schleife mit leeren Anweisungen

```
 1: // Listing 6.10
 2: // for-Schleife mit leeren Anweisungen
 3: #include <iostream>
 4:
 5: int main()
```

```
 6: {
 7:     int counter = 0;
 8:
 9:     for( ; counter < 5; )
10:     {
11:         counter++;
12:         std::cout << "Schleife!  ";
13:     }
14:
15:     std::cout << "\nCounter: " << counter << ".\n";
16:     return 0;
17: }
```

```
Schleife!  Schleife!  Schleife!  Schleife!  Schleife!
Counter: 5.
```

Ein Vergleich mit der weiter vorn in Listing 6.7 gezeigten `while`-Schleife lässt eine nahezu identische Konstruktion erkennen. Die Initialisierung der Zählervariablen erfolgt in Zeile 7. Die `for`-Anweisung in Zeile 9 initialisiert keinerlei Werte, enthält aber einen Test für `counter < 5`. Weiterhin fehlt eine Inkrement-Anweisung, sodass sich diese Schleife wie die folgende Konstruktion verhält:

```
while (counter < 5)
```

Auch hier bietet C++ verschiedene Möglichkeiten, dasselbe zu verwirklichen. Kein erfahrener C++-Programmierer würde eine `for`-Schleife auf diese Weise verwenden. Das Beispiel verdeutlicht aber die Flexibilität der `for`-Anweisung. In der Tat ist es mit `break` und `continue` möglich, eine `for`-Schleife ohne eine der drei Anweisungen zu realisieren. Listing 6.11 zeigt dazu ein Beispiel.

Listing 6.11: forempty.cpp – Eine leere for-Schleifenanweisung

```
 1: // Listing 6.11 zeigt eine
 2: // leere for-Schleifenanweisung
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     int counter=0;          // Initialisierung
 8:     int max;
 9:     std::cout << "Wie viele Hellos?";
10:     std::cin >> max;
11:     for (;;)                // Endlosschleife mit for
12:     {
13:         if (counter < max)  // Test
```

```
14:         {
15:             std::cout << "Hello!\n";
16:             counter++;      // Inkrementieren
17:         }
18:         else
19:             break;
20:     }
21:     return 0;
22: }
```

```
Wie viele Hellos?  3
Hello!
Hello!
Hello!
```

Damit hat man die `for`-Schleife bis zu ihrem absoluten Limit ausgereizt. Initialisierung, Test und Aktion wurden gänzlich aus der `for`-Anweisung herausgenommen. Die Initialisierung findet man in Zeile 7, bevor die `for`-Schleife überhaupt beginnt. Der Test erfolgt in einer separaten `if`-Anweisung in Zeile 13. Verläuft er erfolgreich, wird die Aktion – Inkrementieren von `counter` – in Zeile 16 ausgeführt. Wenn der Test scheitert, verlässt das Programm die Schleife mit der Anweisung in Zeile 19.

Wenn dieses Programm auch etwas absurd erscheint, kann es durchaus sein, dass man genau eine Schleife im Stil von `for(;;)` oder `while(1)` braucht. Ein Beispiel für einen sinnvolleren Einsatz derartiger Schleifen bei `switch`-Anweisungen folgt später in diesem Kapitel.

6.4.3 Leere for-Schleifen

Der Kopf einer `for`-Anweisung bietet sehr viel Spielraum, sodass man manchmal auf einen Rumpf gänzlich verzichten kann. In diesem Fall muss man eine leere Anweisung (`;`) als Rumpf der Schleife vorsehen. Das Semikolon darf auf derselben Zeile wie der Kopf stehen, obwohl man es an dieser Stelle leicht übersehen kann. Listing 6.12 zeigt dazu ein Beispiel.

Listing 6.12: fornullbody.cpp – Darstellung einer leeren Anweisung als Rumpf einer for-Schleife

```
1: // Listing 6.12
2: // Zeigt eine leere Anweisung als Rumpf
3: // einer for-Schleife
4: #include <iostream>
5:
6: int main()
7: {
8:     for (int i = 0; i<5; std::cout << "i: " << i++ << std::endl)
```

```
 9:         ;
10:     return 0;
11: }
```

```
i: 0
i: 1
i: 2
i: 3
i: 4
```

Die for-Schleife in Zeile 8 enthält drei Anweisungen: Die Initialisierungsanweisung richtet den Zähler `i` ein und initialisiert ihn auf `0`. Die Bedingungsanweisung testet auf `i<5` und die Aktionsanweisung gibt den Wert in `i` aus und inkrementiert ihn.

Im Rumpf der for-Schleife selbst bleibt nichts weiter zu tun, sodass man hier eine leere Anweisung (`;`) schreibt. Es sei darauf hingewiesen, dass diese Konstruktion nicht die beste Lösung darstellt. Die Aktionsanweisung ist zu umfangreich. Die folgende Version ist besser geeignet:

```
8:         for (int i = 0; i<5; i++)
9:              std::cout << "i: " << i << endl;
```

Beide Versionen bewirken exakt dasselbe, das zweite Beispiel ist aber leichter zu verstehen. Wenn Sie eine Variable innerhalb einer for-Schleife erzeugen, dürfen Sie nicht vergessen, dass die Variable lokal zu dieser Schleife ist und nur während der Schleifenausführung existiert.

6.4.4 Verschachtelte Schleifen

Befindet sich eine Schleife im Rumpf einer anderen Schleife, spricht man vom *Verschachteln* von Schleifen. Die innere Schleife wird bei jedem Durchlauf der äußeren Schleife vollständig abgearbeitet. Listing 6.13 zeigt ein Beispiel, das Markierungen in einer Matrix mithilfe verschachtelter for-Schleifen schreibt.

Listing 6.13: fornested.cpp – Verschachtelte for-Schleifen

```
 1: // Listing 6.13
 2: // Zeigt verschachtelte for-Schleifen
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     int rows, columns;
 8:     char theChar;
 9:     std::cout << "Wie viele Zeilen? ";
10:     std::cin >> rows;
```

```
11:     std::cout << "Wie viele Spalten? ";
12:     std::cin >> columns;
13:     std::cout << "Welches Zeichen? ";
14:     std::cin >> theChar;
15:     for (int i = 0; i<rows; i++)
16:     {
17:         for (int j = 0; j < columns; j++)
18:             std::cout << theChar;
19:         std::cout << "\n";
20:     }
21:     return 0;
22: }
```

```
Wie viele Zeilen? 4
Wie viele Spalten? 12
Welches Zeichen? x
xxxxxxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxx
```

Der Benutzer wird aufgefordert, die Anzahl der Zeilen (`rows`) und Spalten (`columns`) sowie ein auszugebendes Zeichen einzugeben. Die erste Schleife in Zeile 15 initialisiert einen Zähler `i` mit `0`. Dann beginnt die Ausführung der äußeren Schleife.

In Zeile 17 richtet die erste Zeile im Rumpf der äußeren `for`-Schleife eine weitere `for`-Schleife ein. Hier wird ein zweiter Zähler (`j`) ebenfalls mit `0` initialisiert und der Rumpf der inneren `for`-Schleife ausgeführt. In Zeile 18 erfolgt die Ausgabe des gewählten Zeichens und die Steuerung kehrt zum Kopf der inneren `for`-Schleife zurück. Beachten Sie, dass die innere `for`-Schleife nur aus einer Anweisung besteht (der Ausgabe des Zeichens). Ergibt der Test der Bedingung `j<columns` das Ergebnis `true`, wird `j` inkrementiert und das nächste Zeichen ausgegeben. Das setzt sich fort, bis `j` gleich der Anzahl der Spalten (`columns`) ist.

Sobald der Test der inneren Schleife den Wert `false` liefert, in diesem Beispiel nach Ausgabe von zwölf `x`-Zeichen, springt die Ausführung direkt zu Zeile 19 und gibt eine neue Zeile aus. Die äußere `for`-Schleife kehrt nun zu ihrem Kopf zurück, wo die Bedingung `i<rows` getestet wird. Ergibt dieser Test `true`, wird `i` inkrementiert und der Rumpf der Schleife ausgeführt.

Im zweiten Durchlauf der äußeren `for`-Schleife beginnt die innere `for`-Schleife von neuem. Damit erhält `j` erneut den Anfangswert `0` (!) und wieder wird die gesamte innere Schleife ausgeführt.

Dem Programm liegt der Gedanke zugrunde, dass in einer verschachtelten Schleife die innere Schleife für jeden Durchlauf der äußeren Schleife ausgeführt wird. Die Anzahl der ausgegebenen Zeichen pro Zeile entspricht damit dem Wert in `columns`.

6.5 switch-Anweisungen

Wenn man `if`-Anweisungen und `else...if`-Anweisungen zu tief verschachtelt, kann man leicht den Überblick verlieren. C++ bietet für derartige Konstruktionen eine Alternative. Im Gegensatz zur `if`-Anweisung, die lediglich einen Wert auswertet, lassen sich mit `switch`-Anweisungen Verzweigungen in Abhängigkeit von mehreren unterschiedlichen Werten aufbauen. Die allgemeine Form der `switch`-Anweisung lautet:

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

`Ausdruck` ist jeder gültige C++-Ausdruck. *`Anweisung`* steht für beliebige C++-Anweisungen oder Anweisungsblöcke. `switch` wertet *`Ausdruck`* aus und vergleicht das Ergebnis mit allen Werten hinter `case`. Beachten Sie, dass die Auswertung nur auf Gleichheit erfolgt. Man kann weder relationale noch boolesche Operatoren verwenden.

Wenn einer der `case`-Werte mit dem Ausdruck übereinstimmt, springt die Programmausführung zu diesen Anweisungen und arbeitet sie bis zum Ende des `switch`-Blocks oder bis zur nächsten `break`-Anweisung ab. Lässt sich keine Übereinstimmung ermitteln, verzweigt die Ausführung zur optionalen `default`-Anweisung. Ist kein `default`-Zweig vorhanden und gibt es keinen übereinstimmenden Wert, kommt es überhaupt nicht zur Abarbeitung von Anweisungen in der `switch`-Konstruktion – die Anweisung ist damit beendet.

Fehlersuche

Es empfiehlt sich, in `switch`-Anweisungen immer einen `default`-Zweig vorzusehen. Auch wenn man diesen Zweig eigentlich nicht benötigt, kann man hier zumindest auf angeblich unmögliche `case`-Bedingungen testen und eine Fehlermeldung ausgeben. Damit lässt sich die Fehlersuche oft erheblich beschleunigen.

Im `default`-Zweig ist ein `break` zwar nicht erforderlich, viele Programmierer schreiben es aber, um die ganze Konstruktion einheitlich zu gestalten.

Wichtig ist insbesondere die `break`-Anweisung am Ende eines `case`-Zweiges. Wenn diese fehlt, geht das Programm zur Ausführung des nächsten `case`-Zweiges über – die Programmausführung »fällt durch«. Manchmal ist das zwar gewollt, gewöhnlich handelt es sich aber um einen Fehler. Falls Sie die Programmausführung tatsächlich mit dem nächsten `case`-Zweig fortsetzen möchten, sollten Sie mit einem Kommentar darauf hinweisen, dass die `break`-Anweisung nicht vergessen wurde.

Listing 6.14 zeigt ein Beispiel für die `switch`-Anweisung.

Listing 6.14: switcher.cpp – Einsatz der switch-Anweisung

```
// Listing 6.14
// Zeigt die switch-Anweisung
#include <iostream>

int main()
{
    unsigned short int number;
    std::cout << "Bitte eine Zahl zwischen 1 und 5 eingeben: ";
    std::cin >> number;
    switch (number)
    {
    case 0:
        std::cout << "Leider zu klein!";
        break;
    case 5:
        std::cout << "Gut!\n";           // Weiter mit nächstem case
    case 4:
        std::cout << "Sehr gut!\n";      // Weiter mit nächstem case
    case 3:
        std::cout << "Ausgezeichnet!\n"; // Weiter mit nächstem case
    case 2:
        std::cout << "Meisterhaft!\n";   // Weiter mit nächstem case
    case 1:
        std::cout << "Unglaublich!\n";
        break;
```

```
26:     default:
27:         std::cout << "Zu gross!\n";
28:         break;
29:     }
30:     std::cout << "\n\n";
31:     return 0;
32: }
```

```
Bitte eine Zahl zwischen 1 und 5 eingeben:  3
Ausgezeichnet!
Meisterhaft!
Unglaublich!

Bitte eine Zahl zwischen 1 und 5 eingeben: 8
Zu gross!
```

Das Programm fragt zunächst eine Zahl vom Benutzer ab und wertet sie dann in der `switch`-Anweisung aus. Ist die Zahl gleich `0`, stimmt das mit der `case`-Anweisung in Zeile 12 überein. Es erscheint die Meldung `Leider zu klein!` und die `break`-Anweisung beendet die `switch`-Struktur. Wenn der Wert gleich `5` ist, springt die Programmausführung zu Zeile 16, wo eine Meldung ausgegeben wird. Dann fällt die Programmausführung zu Zeile 18 durch und gibt eine weitere Meldung aus. Das geht so lange, bis die `break`-Anweisung in Zeile 25 erreicht ist.

Im Endeffekt erscheinen bei Zahlen zwischen `2` und `5` mehrere Meldungen auf dem Bildschirm. Wenn die Zahl nicht im Bereich zwischen `0` und `5` liegt, gilt sie im Programm als zu groß. Diesen Fall behandelt die `default`-Anweisung in Zeile 26.

6.6 Fragen und Antworten

F: Wie wählt man zwischen `if...else` und `switch` aus?

A: Wenn mehr als eine oder zwei `else`-Klauseln vorhanden sind und alle denselben Wert testen, bietet sich eine `switch`-Anweisung an. Die `switch`-Anweisung lässt sich nicht einsetzen, wenn Beziehungen (beispielsweise `a > b`) zu vergleichen sind.

F: Wie wählt man zwischen `while` und `do...while` aus?

A: Wenn der Rumpf der Schleife zumindest einmal auszuführen ist, verwendet man eine `do...while`-Schleife. Andernfalls kann man mit einer `while`-Schleife arbeiten.

F: Wie wählt man zwischen `while` und `for` aus?

A: Wenn man eine Zählvariable initialisiert, diese Variable testet und sie bei jedem Schleifendurchlauf inkrementiert, kommt eine `for`-Schleife infrage. Ist die Variable bereits initialisiert und wird nicht bei jedem Schleifendurchlauf inkrementiert, kann eine `while`-Schleife besser geeignet sein.

F: Ist es besser, `while(1)` oder `for(;;)` zu verwenden?

A: Hier gibt es keinen wesentlichen Unterschied.

6.7 Workshop

Nachdem Sie in dieser Stunde einige komplexe Steuerungsstrukturen kennen gelernt haben, sollten Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu diesem Thema zu festigen.

6.7.1 Kontrollfragen

1. Welchen Datentyp verwendet man in einer `for`-Schleife?
2. Wie unterscheiden sich `break` und `continue`?
3. Listing 6.13 zeigt verschachtelte `for`-Schleifen. Lassen sich auch `while`- oder `do...while`-Schleifen verschachteln?
4. Was bewirkt die `break`-Anweisung in einer `switch`-Konstruktion?

6.7.2 Übungen

1. Verwenden Sie in `forloop.cpp` einen Zähler vom Datentyp `float` und geben Sie den Wert innerhalb der Schleife aus. Versuchen Sie, den Zähler um einen Wert wie zum Beispiel `0.1` statt `1` zu inkrementieren.
2. Modifizieren Sie `fornested.cpp`, um `while`-Schleifen anstelle von `for`-Schleifen zu verwenden. Ersetzen Sie eine `for`-Schleife und starten Sie das Programm. Ersetzen Sie dann die andere `for`-Schleife. Das zeigt, dass man auch verschiedene Arten von Schleifen verschachteln kann.
3. Bauen Sie in `switcher.cpp` eine Schleife in einen der `case`-Zweige ein. Lässt sich das Programm kompilieren und ausführen?

6.7.3 Antworten zu den Kontrollfragen

1. Die meisten Programmierer beschränken sich in `for`-Anweisungen auf ganze Zahlen. Das ist aber keine Beschränkung der Sprache; man kann genauso gut mit Gleitpunktzahlen, Zeichenfolgenwerten oder anderen Datentypen arbeiten.
2. Die `break`-Anweisung beendet die Schleife und das Programm fährt mit der nächsten Anweisung nach dem Schleifenkörper fort. Bei der `continue`-Anweisung überspringt das Programm den Rest des Schleifenkörpers, beendet aber die Schleife nicht (außer wenn die Abbruchbedingung erfüllt ist).
3. Selbstverständlich! C++ ist in sich einheitlich. Man kann `for`-Schleifen in `while`-Schleifen innerhalb von `do...while`-Schleifen usw. verschachteln – alles klar?
4. In einer `switch`-Anweisung bewirkt `break`, dass das Programm mit der nächsten Anweisung außerhalb des Körpers der `switch`-Anweisung fortfährt. Ohne `break` werden die Anweisungen beginnend bei der ersten erfüllten `case`-Bedingung bis zum Ende des `switch`-Körpers ausgeführt.

KAPITEL 7

jetzt lerne ich

Klassen – die Grundlagen

In dieser Stunde lernen Sie,

- was Typen sind,
- was Klassen und Objekte sind,
- wie man eine neue Klasse definiert und Objekte dieser Klasse erzeugt.

7.1 Was ist ein Typ?

Ein *Typ* ist eine Kategorie. Den Menschen zeichnet unter anderem die Fähigkeit aus zu kategorisieren. Wir sehen nicht Hunderte von Formen der Savanne. Wir sehen Tiere und Bäume. Und wir sehen nicht einfach Tiere; wir sehen Gazellen, Elefanten, Wasserbüffel usw. Wir klassifizieren, ordnen, gruppieren und teilen ein. Kurz gesagt denken wir in *Typen* von Dingen.

Eine Orange ist ein Typ von Zitrusfrucht. Eine Zitrusfrucht ist ein Typ von Frucht. Eine Frucht ist ein Typ von Pflanze. Eine Pflanze ist ein Typ von Lebewesen.

Zu den alltäglichen Typen gehören Auto, Haus, Person und Figur. In C++ stellt ein Typ ein Objekt mit einer Größe, einem Zustand und einem Satz von Fähigkeiten dar.

Ein C++-Programmierer kann jeden benötigten Typ erzeugen und jeder dieser neuen Typen kann über die gesamte Funktionalität und Leistungsfähigkeit der vordefinierten Typen von C++ wie zum Beispiel `int`, `long` und `double` verfügen.

Programme schreibt man in der Regel, um praxisgerechte Probleme zu lösen, wie etwa die Erfassung von Mitarbeiterdaten oder die Simulation der Abläufe eines Heizungssystems. Obwohl man komplexe Probleme auch mit Programmen lösen kann, die nur mit ganzen Zahlen und Zeichen arbeiten, ist es weitaus einfacher, große, komplexe Probleme in den Griff zu bekommen, wenn man Darstellungen der betreffenden Objekte erzeugen kann. Mit anderen Worten lässt sich die Simulation der Abläufe eines Heizungssystems leichter umsetzen, wenn man variable Typen für die Darstellung von Räumen, Wärmesensoren, Thermostaten und Dampfkesseln erzeugen kann. Je näher diese Variablen der Realität entsprechen, desto einfacher ist es, das Programm zu schreiben.

7.2 Neue Typen erzeugen

Bis jetzt haben Sie eine Reihe von vordefinierten C++-Typen kennen gelernt, einschließlich `int` (Ganzzahlen) und `char` (Zeichen). Der Typ einer Variablen sagt eine ganze Menge über die Variable aus. Wenn man zum Beispiel `Height` und `Width` als ganze Zahlen des Typs `unsigned short` deklariert, ist bekannt, dass jede Variable eine Zahl zwischen 0 und 65535 aufnehmen kann (vorausgesetzt, dass `unsigned short` eine Länge von 2 Byte hat).

Neben der Größe sagt der Typ etwas über die Fähigkeiten der Variablen aus. Beispielsweise kann man Ganzzahlen vom Typ `short` addieren. Somit ergibt sich allein aus der Deklaration von `Height` und `Width` als `unsigned short int`, dass man `Height` und `Width` addieren und diese Zahl einer anderen Zahl zuweisen kann.

Der Typ dieser Variablen sagt etwas aus über

- ihre Größe im Speicher,
- die speicherbaren Informationen,
- die auf diesen Variablen ausführbaren Aktionen.

In C++ können Sie auch Ihre eigenen Typen definieren, um das zu lösende Problem zu modellieren. Der Mechanismus für das Deklarieren eines neuen Typs ist das Erstellen einer Klasse. Eine Klasse ist eine Definition eines neuen Typs.

7.3 Klassen und Elemente

Einen neuen Typ erzeugt man durch die Deklaration einer Klasse. Eine *Klasse* ist eine Sammlung von Variablen – häufig mit unterschiedlichen Typen – kombiniert mit einem Satz relevanter Funktionen.

Man kann sich ein Auto als Sammlung von Rädern, Türen, Sitzen, Fenstern usw. vorstellen. Eine andere Möglichkeit ist es, das Auto als Sammlung seiner möglichen Aktionen aufzufassen: es fährt, beschleunigt, bremst, stoppt, parkt usw.

Kapselung ist die Zusammenfassung aller Informationen, Fähigkeiten und Zuständigkeiten einer Einheit zu einem einzelnen Objekt.

Die Kapselung aller Informationen über ein Auto in einer Klasse bietet für den Programmierer eine Reihe von Vorteilen. Alles ist an einer Stelle zusammengefasst, wodurch man die Daten leicht im Zugriff hat, kopieren und manipulieren kann.

Klienten einer Klasse sind andere Klassen oder Funktionen, die von der jeweiligen Klasse Gebrauch machen. Kapselung erlaubt den Klienten der Klasse, diese zu nutzen, ohne sich um die Funktionsweise zu kümmern oder diese zu kennen. Man kann ein Auto fahren, ohne dass man weiß, wie der Motor funktioniert, und die Klienten einer Klasse können die Klasse verwenden, ohne zu wissen, wie die Klasse die jeweilige Aufgabe realisiert.

Eine Klasse kann aus jeder Kombination von Variablentypen und auch anderen Klassentypen bestehen. Die Variablen in der Klasse bezeichnet man als *Elementvariablen* oder *Datenelemente*. Eine Klasse `Auto` verfügt beispielsweise über Variablen zur Darstellung von Sitzen, Radiotypen, Reifen usw.

Elementvariablen oder *Datenelemente* sind die Variablen in einer Klasse und gehören zu einer Klasse genau wie die Räder und der Motor Teile eines Autos sind.

Normalerweise manipulieren die Funktionen in der Klasse die Elementvariablen. Man bezeichnet diese Funktionen als *Elementfunktionen* oder *Methoden* der Klasse. Methoden der Klasse `Auto` können zum Beispiel `Starten()` und `Bremsen()` sein. Eine Klasse `Katze` könnte über Datenelemente verfügen, die das Alter und Gewicht repräsentieren, als Methoden sind `Schlafen()`, `Miauen()` und `MausFangen()` vorstellbar.

Elementfunktionen – oder Methoden – sind die Funktionen in einer Klasse. Elementfunktionen gehören ebenso zur Klasse wie Elementvariablen und sie bestimmen, was die Objekte der Klasse realisieren können.

7.3.1 Klassen deklarieren

Um eine Klasse zu deklarieren, verwendet man das Schlüsselwort `class` gefolgt von einer öffnenden geschweiften Klammer und einer Liste der Datenelemente und Methoden dieser Klasse. Den Abschluss der Deklaration bildet eine schließende geschweifte Klammer und ein Semikolon. Die Deklaration einer Klasse namens `Cat` (Katze) sieht wie folgt aus:

```
class Cat
{
   unsigned int  itsAge;
   unsigned int  itsWeight;
   Meow();
};
```

Die Deklaration dieser Klasse reserviert noch keinen Speicher für `Cat`. Es ist nur eine Anweisung an den Compiler, was die Klasse `Cat` darstellt, welche Daten sie enthält (`itsAge` und `itsWeight`) und was sie tun kann (`Meow()`). Außerdem wird dem Compiler mitgeteilt, wie groß `Cat` ist – das heißt, wie viel Platz der Compiler für jedes erzeugte `Cat`-Objekt reservieren muss. In diesem Beispiel ist `Cat` lediglich 8 Byte groß (vorausgesetzt, dass Integer mit 4 Byte dargestellt wird): `itsAge` mit 4 Byte und `itsWeight` mit weiteren 4. Die Methode `Meow()` belegt keinen Platz, da für Elementfunktionen (Methoden) kein Speicher reserviert wird.

7.3.2 Ein Wort zu Namenskonventionen

Als Programmierer muss man alle Elementvariablen, Elementfunktionen und Klassen benennen. Wie Sie in Stunde 3 gelernt haben, sollten diese Namen leicht verständlich und aussagekräftig sein. `Cat`, `Rectangle` und `Employee` sind geeignete Klassennamen. `Meow()`, `ChaseMice()` und `StopEngine()` sind passende Funktionsnamen, da man daraus die Aufgabe der Funktionen ablesen kann. Viele Programmierer benennen die Elementvariablen mit dem Präfix `its` (zu deutsch: ihr, bezieht sich auf Klasse) wie in `itsAge`, `itsWeight` und `itsSpeed` (ihrAlter, ihrGewicht, ihreGeschwindigkeit). Damit lassen sich Elementvariablen leichter von anderen Variablen unterscheiden.

Da C++ die Groß-/Kleinschreibung beachtet, sollten alle Klassennamen demselben Muster folgen. Auf diese Weise braucht man nie zu prüfen, wie man den Klassennamen schreiben muss – war es `Rectangle`, `rectangle` oder `RECTANGLE`? Einige Programmierer setzen vor den Klassennamen einen bestimmten Buchstaben (beispielsweise `cCat` oder `cPerson`), während andere den Namen durchgängig groß oder klein schreiben. Der Autor beginnt alle Klassennamen mit einem Großbuchstaben wie in `Cat` und `Person`.

Analog dazu beginnen viele Programmierer alle Funktionen mit Großbuchstaben und alle Variablen mit Kleinbuchstaben. Zusammengesetzte Wörter werden durch einen Unterstrich (wie in `Chase_Mice`) oder durch Großschreibung der einzelnen Wörter (wie in `ChaseMice` oder `DrawCircle`) auseinander gehalten.

Wichtig ist vor allem, dass Sie sich einen Stil zu eigen machen und ihn in jedem Programm konsequent anwenden. Mit der Zeit vervollkommnet sich dieser Stil und umfasst nicht nur Namenskonventionen, sondern auch Einzüge, Ausrichtung von Klammern und Kommentare.

7.3.3 Objekte definieren

Ein Objekt des neuen Typs definiert man fast genauso wie eine Integer-Variable:

```
unsigned int GrossWeight;    // unsigned Integer-Variable definieren
Cat Frisky;                  // ein Cat-Objekt definieren
```

Dieser Code definiert eine Variable namens `GrossWeight` vom Typ `unsigned int`. Die zweite Zeile zeigt die Definition von `Frisky` als Objekt, dessen Klasse (oder Typ) `Cat` ist.

7.3.4 Klassen und Objekte

Man pflegt niemals die Definition einer Katze, man pflegt die Katze an sich. Man unterscheidet also zwischen der Vorstellung von einer Katze und der realen Katze, die gerade durchs Wohnzimmer schleicht. Auf die gleiche Weise unterscheidet C++ zwischen der Klasse `Cat` als Vorstellung von einer Katze und jedem individuellen `Cat`-Objekt. Demzufolge ist `Frisky` ein Objekt vom Typ `Cat`, genauso wie `GrossWeight` eine Variable vom Typ `unsigned int` ist.

Ein Objekt ist einfach eine selbstständige Instanz einer Klasse. Wenn Sie ein Objekt erzeugen, »instanzieren« Sie es von der Klasse.

7.4 Auf Klassenelemente zugreifen

Nachdem man ein `Cat`-Objekt definiert hat (zum Beispiel `Frisky`), kann man mit dem Punktoperator (`.`) auf die Elemente dieses Objekts zugreifen. Zum Beispiel weist man den Wert `50` an die Elementvariable `itsWeight` des Objekts `Frisky` wie folgt zu:

```
Frisky.itsWeight = 50;
```

In gleicher Weise ruft man die Funktion `Meow()` auf:

```
Frisky.Meow();
```

7.4.1 An Objekte, nicht an Klassen zuweisen

In C++ weist man die Werte nicht den Typen, sondern den Variablen zu. Beispielsweise schreibt man niemals

```
int = 5;              // falsch
```

Der Compiler mokiert das als Fehler, da man den Wert `5` nicht an einen Integer-Typ zuweisen kann. Stattdessen muss man eine Integer-Variable erzeugen und dieser Variablen den Wert `5` zuweisen:

```
int  x;                 // x als int definieren
x = 5;                  // den Wert von x auf 5 setzen
```

Das ist eine Kurzform für die Anweisung »weise 5 der Variablen x zu, die vom Typ int ist«. In diesem Sinne schreibt man auch nicht

```
Cat.age = 5;
```

Man muss ein Cat-Objekt definieren und diesem Objekt den Wert 5 zuweisen:

```
Cat Frisky;             // analog zum Beispiel int x;
Frisky.age = 5;         // analog zum Beispiel x = 5;
```

7.5 Die Schlüsselwörter private und public

Bei der Deklaration einer Klasse verwendet man weitere Schlüsselwörter. Zwei der wichtigsten sind public (öffentlich) und private.

Alle Elemente einer Klasse – Daten und Methoden – sind per Vorgabe privat. Auf private Elemente kann man nur innerhalb der Methoden der Klasse selbst zugreifen. Öffentliche Elemente sind über jedes Objekt der Klasse zugänglich. Der Unterschied zwischen beiden ist wichtig und verwirrend zugleich. Um es klarer herauszustellen, sehen wir uns ein früheres Beispiel dieses Kapitels an:

```
class Cat
{
   unsigned int  itsAge;
   unsigned int  itsWeight;
   Meow();
};
```

In dieser Deklaration sind itsAge, itsWeight und Meow() privat, da alle Elemente einer Klasse per Vorgabe privat sind. Solange man also nichts anderes festlegt, bleiben sie privat.

Wenn man allerdings schreibt

```
Cat  Boots;
Boots.itsAge=5;    // Fehler! Kann nicht auf private Daten zugreifen!
```

zeigt der Compiler einen Fehler an. Praktisch hat man dem Compiler gesagt: »Ich werde auf itsAge, itsWeight und Meow() nur innerhalb der Elementfunktionen der Cat-Klasse zugreifen.« Im Beispiel findet jedoch ein Zugriff von außerhalb einer Cat-Methode statt. Nur weil Boots ein Objekt der Klasse Cat ist, heißt das nicht, dass man auf die privaten Teile von Boots zugreifen kann.

Um auf die Datenelemente von Cat zugreifen zu können, deklariert man einen Abschnitt der Klasse Cat als public:

```
class Cat
{
public:
   unsigned int  itsAge;
   unsigned int  itsWeight;
   Meow();
};
```

Nunmehr sind itsAge, itsWeight und Meow() öffentlich. Der Compiler hat nichts mehr an Boots.itsAge=5 auszusetzen.

Als allgemeine Entwurfsregel gilt, dass man die Datenelemente einer Klasse privat halten sollte. Daher muss man öffentliche Elementfunktionen, die so genannten *Zugriffsmethoden*, erzeugen, um private Elementvariablen zu setzen und zu lesen. Über Zugriffsmethoden greifen andere Teile des Programms auf die Werte von privaten Elementvariablen zu, um diese festzulegen oder abzurufen.

Zugriffsmethoden gestatten es, die Einzelheiten der Speicherung von der Verwendung der Daten zu trennen. Damit kann man die Speicherung der Daten verändern, ohne dass man die Funktionen neu schreiben muss, die mit den Daten arbeiten.

Mit den Schlüsselwörtern für die Zugriffssteuerung deklariert man Abschnitte der Klasse als öffentlich oder privat. Jedes Schlüsselwort ändert die Zugriffssteuerung von diesem Punkt an bis zum Ende der Klasse oder bis zum nächsten dementsprechenden Schlüsselwort. Klassendeklarationen enden mit einer schließenden geschweiften Klammer und einem Semikolon.

Sehen Sie sich folgendes Beispiel an:

```
class Cat
{
public:
    unsigned int Age;
    unsigned int Weight;
    void Meow();
};

Cat  Frisky;
Frisky.Age = 8;
Frisky.Weight = 18;
Frisky.Meow();
```

Und hier noch ein umfangreicheres Beispiel:

```
class Car
{
public:                              // die nächsten fünf Elemente sind
                                     // öffentlich

    void Start();
    void Accelerate();
    void Brake();
    void SetYear(int year);
    int GetYear();

private:                             // der Rest ist privat

    int Year;
    Char Model [255];
};                                   // Ende der Klassendeklaration

Car OldFaithful;                     // eine Instanz von Car erzeugen
int bought;                          // eine lokale Variable vom Typ
int
OldFaithful.SetYear(84) ;            // 84 an Year zuweisen
bought = OldFaithful.GetYear();      // bought auf 84 setzen
OldFaithful.Start();                 // die Methode Start aufrufen
```

Hält man die Elementdaten privat, schränkt man den Zugriff ein und steuert, wie die Werte geändert werden können.

7.6 Klassenmethoden implementieren

Jede Klassenmethode (oder Elementfunktion), die man deklariert, muss man auch definieren.

Eine Methodendefinition beginnt mit dem Namen der Klasse, gefolgt von zwei Doppelpunkten, dem Namen der Methode und ihren Parametern. Listing 7.1 zeigt die vollständige Deklaration einer einfachen `Cat`-Klasse mit der Implementierung einer Zugriffsmethode und einer allgemeinen Elementfunktion.

Listing 7.1: simple-class.cpp – Methoden einer einfachen Klasse implementieren

```
1: // Zeigt die Deklaration einer Klasse und
2: // die Definition von Klassenmethoden
3:
4: #include <iostream>          // für std::cout
5:
6: class Cat                    // Beginn der Klassendeklaration
7: {
```

```
 8: public:                     // Beginn des öffentlichen Abschnitts
 9:     int GetAge();           // Zugriffsfunktion
10:     void SetAge (int age);  // Zugriffsfunktion
11:     void Meow();            // Allgemeine Funktion
12: private:                    // Beginn des privaten Abschnitts
13:     int itsAge;             // Elementvariable
14: };
15:
16: // Die öffentliche Zugriffsfunktion GetAge
17: // gibt den Wert des Datenelements itsAge zurück.
18: int Cat::GetAge()
19: {
20:     return itsAge;
21: }
22:
23: // Definition von SetAge als öffentliche
24: // Zugriffsfunktion
25: // Setzt Datenelement itsAge
26: void Cat::SetAge(int age)
27: {
28:    // Elementvariable itsAge auf den als
29:    // Parameter age übergebenen Wert setzen.
30:     itsAge = age;
31: }
32:
33: // Definition der Methode Meow
34: // Rückgabe: void
35: // Parameter: Keine
36: // Aktion: Gibt "Miau" auf dem Bildschirm aus
37: void Cat::Meow()
38: {
39:     std::cout << "Miau.\n";
40: }
41:
42: // Eine Katze erzeugen, Alter festlegen, Miauen lassen,
43: // Alter mitteilen, dann erneut miauen.
44: int main()
45: {
46:     Cat Frisky;
47:     Frisky.SetAge(5);
48:     Frisky.Meow();
49:     std::cout << "Frisky ist eine Katze, die " ;
50:     std::cout << Frisky.GetAge() << " Jahre alt ist.\n";
51:     Frisky.Meow();
52:     return 0;
53: }
```

```
Miau.
Frisky ist eine Katze, die 5 Jahre alt ist.
Miau.
```

Die Zeilen 6 bis 14 enthalten die Definition der Klasse `Cat`. Das Schlüsselwort `public` in Zeile 8 teilt dem Compiler mit, dass die nachfolgenden Elemente öffentlich sind. Zeile 9 deklariert die öffentliche Zugriffsmethode `GetAge()`, die den Zugriff auf die in Zeile 13 deklarierte private Elementvariable `itsAge` realisiert. In Zeile 10 steht die öffentliche Zugriffsmethode `SetAge()`. Diese Methode übernimmt eine ganze Zahl als Argument und setzt `itsAge` auf den Wert dieses Arguments.

Zeile 12 leitet den privaten Abschnitt ein, der nur die Deklaration in Zeile 13 für die private Elementvariable `itsAge` enthält. Die Klassendeklaration endet mit einer schließenden geschweiften Klammer und dem Semikolon in Zeile 14.

Die Zeilen 18 bis 21 enthalten die Definition der Elementfunktion `GetAge()`. Diese Methode übernimmt keine Parameter, gibt aber eine ganze Zahl zurück. Beachten Sie, dass Klassenmethoden den Klassennamen enthalten, gefolgt von zwei Doppelpunkten und dem Namen der Methode (siehe Zeile 18). Diese Syntax weist den Compiler an, dass die hier definierte Methode `GetAge()` diejenige ist, die in der Klasse `Cat` deklariert wurde. Mit Ausnahme der Kopfzeile wird die Funktion `GetAge()` wie jede andere Funktion erzeugt.

Die Funktion `GetAge()` besteht nur aus einer Zeile und gibt den Wert in `itsAge` zurück. Beachten Sie, dass die Funktion `main()` nicht auf `itsAge` zugreifen kann, da `itsAge` zur Klasse `Cat` privat ist. Die Funktion `main()` hat Zugriff auf die öffentliche Methode `GetAge()`. Da `GetAge()` eine Elementfunktion der Klasse `Cat` ist, hat sie uneingeschränkten Zugriff auf die Variable `itsAge`. Dieser Zugriff erlaubt `GetAge()`, den Wert von `itsAge` an `main()` zurückzugeben.

In Zeile 26 beginnt die Definition der Elementfunktion `SetAge()`. Diese Funktion übernimmt einen ganzzahligen Parameter und setzt in Zeile 30 den Wert von `itsAge` auf den Wert dieses Parameters. Da es sich um eine Elementfunktion der Klasse `Cat` handelt, hat `SetAge()` direkten Zugriff auf die Elementvariable `itsAge`.

In Zeile 37 beginnt die Definition – oder Implementierung – der Methode `Meow()` der Klasse `Cat`. Es handelt sich um eine einzeilige Funktion, die das Wort `Miau` gefolgt vom Zeichen für Neue Zeile auf dem Bildschirm ausgibt. (Wie Sie wissen, bewirkt das Zeichen `\n` eine Zeilenschaltung bei der Ausgabe auf dem Bildschirm.)

In Zeile 44 beginnt das eigentliche Programm mit der bekannten Funktion `main()`. Die Funktion `main()` übernimmt hier keine Argumente und gibt `void` zurück. In Zeile 46 deklariert `main()` ein `Cat`-Objekt namens `Frisky`. In Zeile 47 wird der Wert `5` an die Elementvariable `itsAge` mithilfe der Zugriffsmethode `SetAge()` zugewiesen. Beachten Sie, dass der Aufruf dieser Methode mit dem Klassennamen (`Frisky`) gefolgt vom Punktoperator (`.`) und dem Methodennamen (`SetAge`) erfolgt. In der gleichen Weise kann man jede andere Methode in einer Klasse aufrufen.

Zeile 48 ruft die Elementfunktion `Meow()` auf. Die Zeilen 49 und 50 geben eine Meldung mittels der Zugriffsmethode `GetAge()` aus. In Zeile 51 steht ein weiterer Aufruf von `Meow()`.

7.7 Konstruktoren und Destruktoren – Objekte erstellen und löschen

Bei der Definition einer Integer-Variablen gibt es zwei Verfahren. Man kann die Variable definieren und ihr später im Programm einen Wert zuweisen. Dazu ein Beispiel:

```
int Weight;           // eine Variable definieren
...                   // hier steht anderer Code
Weight = 7;           // der Variablen einen Wert zuweisen
```

Man kann die Ganzzahl auch definieren und sie sofort initialisieren:

```
int Weight = 7;       // definieren und mit 7 initialisieren
```

Die Initialisierung kombiniert die Definition der Variablen mit einer anfänglichen Zuweisung. Nichts hindert Sie daran, den Wert später zu ändern. Die Initialisierung stellt sicher, dass eine Variable niemals ohne einen sinnvollen Wert vorkommt.

Wie initialisiert man nun die Datenelemente einer Klasse? Klassen haben eine spezielle Elementfunktion, einen so genannten *Konstruktor*. Das Programm ruft den Konstruktor auf, wenn es ein Objekt der betreffenden Klasse instanziert. Die Aufgabe des Konstruktors besteht darin, eine gültige Instanz der Klasse zu erzeugen. Dazu gehört oftmals auch das Initialisieren ihrer Datenelemente. Der Konstruktor ist eine Klassenmethode mit demselben Namen wie die Klasse selbst, jedoch ohne Rückgabewert. Optional lassen sich an den Konstruktor Parameter übergeben – wie bei jeder anderen Methode der Klasse.

Zur Deklaration eines Konstruktors gehört in der Regel auch die Deklaration eines *Destruktors*. Genau wie Konstruktoren Objekte der Klasse erzeugen und initialisieren, führen Destruktoren Aufräumungsarbeiten nach Abbau des

Objekts aus und geben reservierten Speicher frei. Ein Destruktor hat immer den Namen der Klasse, wobei eine Tilde (~) vorangestellt ist. Destruktoren übernehmen keine Argumente und haben keinen Rückgabewert. Der Destruktor der Klasse `Cat` ist demnach wie folgt deklariert:

```
~Cat();
```

7.7.1 Standardkonstruktoren

Wenn man

```
Cat Frisky(5);
```

schreibt, ruft man den Konstruktor für `Cat` auf, der einen Parameter (in diesem Fall den Wert `5`) übernimmt. Schreibt man dagegen

```
Cat Frisky;
```

kann man die Klammern weglassen. Der Compiler ruft damit den *Standardkonstruktor* auf. Der Standardkonstruktor ist ein Konstruktor ohne Parameter.

7.7.2 Vom Compiler bereitgestellte Konstruktoren

Wenn man überhaupt keine Konstruktoren deklariert, erzeugt der Compiler automatisch einen Standardkonstruktor. (Wie eben gesagt, handelt es sich beim Standardkonstruktor um den Konstruktor, der keine Parameter übernimmt.)

Der vom Compiler bereitgestellte Standardkonstruktor führt keine Aktion aus. Es ist quasi so, als hätte man einen Konstruktor deklariert, der keine Parameter übernimmt und dessen Rumpf leer ist.

Dazu sind zwei wichtige Punkte zu beachten:

- Der Standardkonstruktor ist ein beliebiger Konstruktor ohne Parameter. Es spielt keine Rolle, ob man ihn selbst deklariert oder vom Compiler übernimmt.
- Wenn man *irgendeinen* Konstruktor deklariert (mit oder ohne Parameter), stellt der Compiler *keinen* Standardkonstruktor bereit. Wenn man in diesem Fall dennoch einen Standardkonstruktor wünscht, muss man ihn selbst deklarieren.

Wenn man keinen Destruktor angibt, stellt der Compiler ebenfalls einen entsprechenden Destruktor bereit, dessen Rumpf leer ist und der keine Aktionen ausführt.

Aus Formgründen sollte man einen Destruktor deklarieren, wenn man einen Konstruktor deklariert, auch wenn der Destruktor keine weiteren Aufgaben

wahrnimmt. Obwohl der Standarddestruktor durchaus korrekt arbeitet, stört es nicht, einen eigenen Destruktor zu deklarieren. Der Code wird dadurch verständlicher.

In Listing 7.2 ist die Klasse `Cat` umgeschrieben und verwendet nun einen Konstruktor, um das `Cat`-Objekt zu initialisieren, wobei das Alter (`age`) auf den bereitgestellten Anfangswert gesetzt wird. Außerdem demonstriert das Listing, wo der Aufruf des Destruktors stattfindet.

Listing 7.2: constructive.cpp – Konstruktoren und Destruktoren

```
// Zeigt die Deklaration eines Konstruktors und
// eines Destruktors für die Klasse Cat
#include <iostream>      // für std::cout
using std::cout;         // in dieser Datei immer std::cout
verwenden

class Cat                // Beginn der Klassendeklaration
{
public:                  // Beginn des öffentlichen Abschnitts
    Cat(int initialAge); // Konstruktor
    ~Cat();              // Destruktor
    int GetAge();        // Zugriffsfunktion
    void SetAge(int age);// Zugriffsfunktion
    void Meow();
private:                 // Beginn des privaten Abschnitts
    int itsAge;          // Elementvariable
};

// Konstruktor von Cat
Cat::Cat(int initialAge)
{
    itsAge = initialAge;
}

// Destruktor, führt keine Aktionen aus
Cat::~Cat()
{

}

// Die öffentliche Zugriffsfunktion GetAge
// gibt den Wert des Datenelements itsAge zurück.
int Cat::GetAge()
{
    return itsAge;
}

// Definition von SetAge als öffentliche
// Zugriffsfunktion
```

```
39: void Cat::SetAge(int age)
40: {
41:     // Elementvariable itsAge auf den als
42:     // Parameter age übergebenen Wert setzen.
43:     itsAge = age;
44: }
45:
46: // Definition der Methode Meow
47: // Rückgabe: void
48: // Parameter: Keine
49: // Aktion: Gibt "Miau" auf dem Bildschirm aus
50: void Cat::Meow()
51: {
52:     cout << "Miau.\n";
53: }
54:
55: // Eine Katze erzeugen, Alter festlegen, Miauen lassen,
56: // Alter mitteilen, dann erneut miauen.
57: int main()
58: {
59:     Cat Frisky(5);
60:     Frisky.Meow();
61:     cout << "Frisky ist eine Katze, die " ;
62:     cout << Frisky.GetAge() << " Jahre alt ist.\n";
63:     Frisky.Meow();
64:     Frisky.SetAge(7);
65:     cout << "Jetzt ist Frisky " ;
66:     cout << Frisky.GetAge() << " Jahre alt.\n";
67:     return 0;
68: }
```

```
Miau.
Frisky ist eine Katze, die 5 Jahre alt ist.
Miau.
Jetzt ist Frisky 7 Jahre alt.
```

Listing 7.2 ist ähnlich zu Listing 7.1, außer dass Zeile 9 einen Konstruktor hinzufügt, der eine Ganzzahl übernimmt. Zeile 10 deklariert den Destruktor. Destruktoren übernehmen niemals Parameter und weder Konstruktoren noch Destruktoren geben Werte zurück – nicht einmal `void`.

Die Zeilen 19 bis 22 zeigen die Implementierung des Konstruktors, die der Implementierung der Zugriffsmethode `SetAge()` ähnlich ist. Es gibt keinen Rückgabewert.

Die Zeilen 25 bis 28 zeigen die Implementierung des Destruktors `~Cat()`. Diese Funktion führt nichts aus, man muss aber die Definition der Funktion einbinden, wenn man sie in der Klassendeklaration definiert.

Zeile 59 enthält die Definition des `Cat`-Objekts `Frisky`. An den Konstruktor von `Frisky` wird der Wert `5` übergeben. Man muss hier nicht `SetAge()` aufrufen, da `Frisky` mit dem Wert `5` in seiner Elementvariablen `itsAge` erzeugt wurde. Zeile 62 zeigt diesen Wert zur Bestätigung an. Auf Zeile 64 wird der Variablen `itsAge` von `Frisky` der neue Wert `7` zugewiesen. Zeile 66 gibt den neuen Wert aus.

Mehr zum Schlüsselwort using

Dieses Beispiel verwendet das Schlüsselwort `using`. Dieses Format (ohne den Modifizierer `namespace` wie in Lektion 6) sagt dem Compiler, dass sich jeder Aufruf von `cout` auf die Standardbibliothek bezieht. Wenn Sie andere Standardfunktionen (beispielsweise `cin`) verwenden, müssten Sie sie als `std::cin` kodieren oder wie in Zeile 4 spezifizieren.

7.8 Fragen und Antworten

F: Wie groß ist ein Klassenobjekt?

A: Die Größe eines Klassenobjekts im Speicher ergibt sich aus der Summe aller Größen seiner Elementvariablen. Klassenmethoden belegen keinen Platz in dem für das Objekt reservierten Hauptspeicher.

Manche Compiler richten die Variablen im Speicher so aus, dass 2 Byte-Variablen tatsächlich etwas mehr Speicher verbrauchen als 2 Bytes. Es empfiehlt sich, hierzu in der Dokumentation des Compilers nachzusehen, aber momentan gibt es keinen Grund, sich mit diesen Dingen im Detail zu beschäftigen.

F: Warum sollten Datenelemente nach Möglichkeit privat bleiben?

A: Indem man die Datenelemente privat macht, kann der Benutzer der Klasse die Daten verwenden, ohne sich um die Art und Weise ihrer Speicherung oder Verarbeitung kümmern zu müssen. Wenn zum Beispiel die Klasse `Cat` eine Methode `GetAge()` enthält, können die Benutzer der Klasse `Cat` das Alter der Katze abrufen, ohne zu wissen oder sich darum zu kümmern, ob die Katze ihr Alter in einer Elementvariablen speichert oder es im Vorübergehen berechnet. Öffentliche Daten verhalten sich wie globale Daten – jeglicher Code, der das Objekt verwendet, kann auf die Daten zugreifen; hat sich etwas geändert, lässt sich nur schwer herausfinden, warum oder wo das geschehen ist.

7.9 Workshop

Nachdem Sie einen ersten Einblick in Klassen und Objekte erhalten haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihre Kenntnisse zu festigen.

7.9.1 Kontrollfragen

1. Worin liegt der Unterschied zwischen einer Klasse und einem Objekt?
2. Worin liegt der Unterschied zwischen `public` und `private` innerhalb eines Objekts.
3. Was bewirkt ein Konstruktor?
4. Was bewirkt ein Destruktor?

7.9.2 Übungen

1. Modifizieren Sie `simpleclass.cpp`, um eine zweite Katze (namens `Charly`) zu erstellen. Können Sie Charly zum Miauen bringen?
2. Was passiert in `simpleclass.cpp`, wenn Sie versuchen, `itsAge` aus `main` heraus zu ändern? Können Sie einen Ausdruck wie `itsAge++` oder `Frisky.itsAge++` kodieren? Was sagt Ihnen das über öffentliche und private Elementvariablen?
3. Machen Sie `itsAge` in `simpleclass.cpp` zu einer öffentlichen Variablen und probieren Sie Übung 2 erneut aus. Da sich nun die Daten in `Frisky` auch aus `main` heraus ändern lassen, ist es schwieriger zu bestimmen, wo etwas schief gegangen ist oder warum die Variable ihren derzeitigen Wert erhalten hat.

7.9.3 Antworten zu den Kontrollfragen

1. Eine Klasse ist die Beschreibung oder Definition; sie belegt keinen Speicher. Ein Objekt ist die Implementierung (oder: Instanzierung) dieser Klasse.
2. Auf private Daten und Methoden (Funktionen) kann man nur aus der Klasse selbst zugreifen. Öffentliche Daten und Methoden (Funktionen) sind auch von außerhalb der Klasse zugänglich. Im Allgemeinen sollte man alle Klassendatenelemente privat halten und die Methoden öffentlich zugänglich machen (damit sie sich aufrufen lassen).

3. Ein Konstruktor erzeugt ein Objekt basierend auf der Klassendefinition. Bei einem einfachen Objekt können Sie es dem Compiler überlassen, einen Standardkonstruktor zu erstellen, während bei einem komplexen Objekt möglicherweise ein eigener Konstruktor erforderlich ist.
4. Ein Destruktor bereinigt die Datenelemente innerhalb Ihres Objekts, wenn Sie es nicht mehr benötigen. Bei einem einfachen Objekt können Sie es dem Compiler überlassen, einen Standarddestruktor zu erstellen, während Sie bei einem komplexen Objekt möglicherweise einen speziellen Destruktor schreiben müssen (darauf gehen wir in Kürze ein).

KAPITEL 8

jetzt lerne ich

Mehr über Klassen

In dieser Stunde lernen Sie,

- was konstante Elementfunktionen sind,
- wie man die Schnittstelle der Klasse von ihrer Implementierung trennt,
- wie man Klassen verwaltet und wie der Compiler dabei helfen kann, Fehler zu suchen und zu vermeiden.

8.1 Konstante Elementfunktionen

Wenn man eine Elementfunktion (sprich: Methode) einer Klasse als `const` deklariert, legt man fest, dass die Methode nicht den Wert irgendeines Elements der Klasse ändert. Um eine Klassenmethode als konstant zu deklarieren, schreibt man das Schlüsselwort `const` nach den Klammern und vor das Semikolon. Die Deklaration der *konstanten Elementfunktion* `SomeFunction()` übernimmt keine Argumente und gibt `void` zurück. Sie sieht folgendermaßen aus:

```
void SomeFunction() const;
```

Zugriffsfunktionen werden häufig als konstante Funktionen mithilfe des Modifizierers `const` deklariert. Die Klasse `Cat` hat zwei Zugriffsfunktionen:

```
void SetAge(int anAge);
int GetAge();
```

SetAge() kann nicht const sein, da diese Funktion die Elementvariable itsAge ändert. Dagegen kann und sollte man GetAge() als const deklarieren, da diese Funktion die Klasse überhaupt nicht verändert. Sie gibt einfach den aktuellen Wert der Elementvariablen itsAge zurück. Die Deklaration dieser Funktionen schreibt man daher wie folgt:

```
void SetAge(int anAge);
int GetAge() const;
```

Wenn man eine Funktion als konstant deklariert und dann die Implementierung dieser Funktion das Objekt ändert (durch Änderung des Wertes eines ihrer Elemente), zeigt der Compiler einen entsprechenden Fehler an. Hat man zum Beispiel die Methode GetAge() so geschrieben, dass sie die Anzahl der Anfragen nach dem Alter von Cat protokolliert, würde man einen Compiler-Fehler erhalten. Durch den Aufruf dieser Methode verändert man nämlich das Cat-Objekt.

Unveränderlichkeit

Verwenden Sie const wann immer möglich. Deklarieren Sie Elementfunktionen als const, wenn sie das Objekt nicht ändern sollen. Damit erleichtert Ihnen der Compiler die Fehlersuche. Das geht schneller und ist weniger aufwändig, als wenn man es selbst machen müsste.

Es gehört zum guten Programmierstil, so viele Methoden wie möglich als const zu deklarieren. Damit kann bereits der Compiler Fehler abfangen und die Fehler zeigen sich nicht erst bei laufendem Programm. Eine const-Methode verwendet man aus den gleichen Gründen wie eine const-Variable.

8.2 Schnittstelle und Implementierung

Wie Sie bereits wissen, sind Klienten die Teile des Programms, die Objekte einer bestimmten Klasse erzeugen und verwenden. Man kann sich die Schnittstelle zu dieser Klasse – die Klassendeklaration – als Vertrag mit diesen Klienten vorstellen. Der Vertrag sagt aus, welche Daten in der Klasse verfügbar sind und wie sich die Klasse verhält.

In der Klassendeklaration von Cat erzeugen Sie zum Beispiel einen Vertrag, dass jedes Cat-Objekt über eine Elementvariable itsAge verfügt, die man mit dem Konstruktor der Klasse initialisieren, durch ihre Zugriffsfunktion SetAge() zuweisen und durch ihre Zugriffsfunktion GetAge() lesen kann. Außerdem sagen Sie zu, dass jedes Cat-Objekt weiß, wie zu Miauen ist (Funktion Meow()).

Wenn man `GetAge()` zu einer konstanten Funktion macht (wie es empfohlen wird), verspricht der Vertrag auch, dass `GetAge()`das `Cat`-Objekt nicht ändert, auf dem die Funktion aufgerufen wird.

Warum mit dem Compiler Fehler abfangen?

In der rauen Wirklichkeit der Programmierung sieht es so aus, dass kein Mensch absolut fehlerfreien Code schreiben kann. Was die Profis von den Amateuren unterscheidet ist nicht, dass der Code von Profis perfekt ist, sondern dass hier die Fehler *vor* der Auslieferung des Produkts und nicht danach gefunden werden.

Compiler-Fehler – das heißt, während der Kompilierung gefundene Fehler – sind weitaus angenehmer als Laufzeitfehler – das heißt, während der Ausführung des Programms auftretende Fehler.

Das hängt damit zusammen, dass sich Compiler-Fehler zuverlässiger einkreisen lassen. Es kann durchaus sein, dass ein Programm viele Male läuft, ohne alle möglichen Wege im Code zu durchlaufen. Somit kann ein Laufzeitfehler eine ganze Weile im Verborgenen bleiben. Compiler-Fehler werden während der Kompilierung gefunden, sodass man sie leicht identifizieren und beseitigen kann. Ziel der qualitätsgerechten Programmierung ist es sicherzustellen, dass der Code keine Laufzeitfehler hervorbringt. Deshalb hat es sich bewährt, bereits im frühen Stadium der Programmentwicklung Fehler mit dem Compiler abzufangen.

Code kann hundertprozentig zulässig sein und trotzdem nicht das tun, was der Programmierer beabsichtigt hat. Aus diesem Grund ist immer ein Team für die Qualitätssicherung erforderlich.

8.3 Klassendeklarationen und Methodendefinitionen platzieren

Jede deklarierte Funktion muss eine Definition haben. Die Definition einer Funktion bezeichnet man auch als *Implementierung*. Wie jede andere Funktion, hat auch die Definition einer Klassenmethode einen Funktionskopf und einen Funktionsrumpf.

Die Definition muss in einer Datei stehen, die der Compiler finden kann. Die meisten C++-Compiler erwarten, dass diese Datei auf `.c` oder `.cpp` endet. In diesem Buch verwenden wir die Erweiterung `.cpp`, da wir die Beispiele mit dem C++BuilderX-Compiler kompilieren. Bei einem anderen Compiler sollten Sie im Handbuch nachsehen, welche Erweiterung er bevorzugt.

Quelldateierweiterungen

Viele Compiler nehmen an, dass es sich bei Dateien mit der Erweiterung `.c` um C-Programme handelt und dass C++-Programme mit `.cpp` enden. Man kann zwar eine beliebige Erweiterung verwenden, mit `.cpp` schafft man aber klare Verhältnisse.

Diese `.cpp`-Dateien müssen Sie in Ihr Projekt oder in die Make-Datei aufnehmen. Die hierzu notwendigen Schritte hängen vom konkreten Compiler ab. In einer integrierten Entwicklungsumgebung ist in der Regel ein Befehl wie »Dem Projekt hinzufügen« (beispielsweise in C++BuilderX DATEIEN HINZUFÜGEN im Menü PROJEKT) vorhanden. In das Projekt sind alle `.cpp`-Dateien aufzunehmen, damit sich das Projekt zur ausführbaren Datei kompilieren und linken lässt.

Obwohl man durchaus die Deklaration in der Quellcodedatei unterbringen kann, gehört dieses Vorgehen nicht zum guten Programmierstil. Die von den meisten Programmierern anerkannte Konvention ist die Unterbringung der Deklaration in einer so genannten *Header-Datei*, die gewöhnlich den gleichen Namen hat, aber auf `.h`, `.hp` oder `.hpp` endet. Im Buch werden Header-Dateien mit der Erweiterung `.hpp` verwendet. Auch hierzu sollten Sie im Compiler-Handbuch nachschlagen.

Schreibt man zum Beispiel die Deklaration der Klasse `Cat` in eine Datei namens `cat.hpp`, wird die Definition der Klassenmethoden in der Datei `cat.cpp` gespeichert. Die Header-Datei bindet man dann in die `.cpp`-Datei ein, indem man den folgenden Code an den Beginn von `cat.cpp` schreibt:

```
#include Cat.hpp
```

Damit weist man den Compiler an, `cat.hpp` in die Datei einzulesen, genauso als hätte man ihren Inhalt an dieser Stelle eingetippt. Warum macht man sich die Mühe, diese Angaben in separate Dateien zu schreiben, wenn man sie ohnehin wieder zusammenführt? In der Regel kümmern sich die Klienten einer Klasse nicht um die Details der Implementierung. Es genügt demnach, die Header-Datei zu lesen, um alle erforderlichen Angaben beisammen zu haben.

Klassendeklaration

Die Deklaration einer Klasse sagt dem Compiler, um welche Klasse es sich handelt, welche Daten sie aufnimmt und welche Funktionen sie ausführt. Die Deklaration der Klasse bezeichnet man als *Schnittstelle* (Interface), da der Benutzer hieraus die Einzelheiten zur Interaktion mit der Klasse entnehmen kann. Die Schnittstelle speichert man in der Regel in einer `.hpp`-Datei, einer so genannten *Header-Datei*.

Die Funktionsdefinition sagt dem Compiler, wie die Funktion arbeitet. Man bezeichnet die Funktionsdefinition als *Implementierung* der Klassenmethode und legt sie in einer `.cpp`-Datei ab. Die Implementierungsdetails der Klasse sind nur für den Autor der Klasse von Belang. Klienten der Klasse – das heißt, die Teile des Programms, die die Klasse verwenden – brauchen nicht zu wissen (und sich nicht darum zu kümmern), wie die Funktionen implementiert sind.

Dieser Ansatz bietet vor allem den Vorteil, dass man Klassen gemeinsam nutzen kann, sodass andere Programmierer die von Ihnen entworfenen Objekte instanzieren können. Ebenso können Sie Klassen von anderen Programmierern übernehmen (beispielsweise von Team-Mitgliedern oder kommerziellen Anbietern).

8.4 Inline-Implementierung

So wie man den Compiler auffordern kann, eine normale Funktion als Inline-Funktion einzubinden, kann man auch Klassenmethoden als inline implementieren. Das Schlüsselwort `inline` steht vor dem Rückgabewert. Zum Beispiel sieht die Inline-Implementierung der Funktion `GetWeight()` folgendermaßen aus:

```
inline int Cat::GetWeight()
{
    return itsWeight;     // das Datenelement Weight zurückgeben
}
```

Man kann auch die Definition einer Funktion in die Deklaration der Klasse schreiben, was die Funktion automatisch zu einer Inline-Funktion macht. Zum Beispiel:

```
class Cat
{
public:
    int GetWeight() { return itsWeight; }    // inline
    void SetWeight(int aWeight);
};
```

Beachten Sie die Syntax der Definition von `GetWeight()`. Der Rumpf der Inline-Funktion beginnt unmittelbar nach der Deklaration der Klassenmethode. Nach den Klammern steht kein Semikolon. Wie bei jeder Funktion beginnt die Definition mit einer öffnenden geschweiften Klammer und endet mit einer schließenden geschweiften Klammer. Wie üblich spielen Whitespaces keine Rolle. Man hätte die Deklaration auch wie folgt schreiben können:

```
class Cat
{
public:
    int GetWeight()
    {
        return itsWeight;
    }                           // inline
    void SetWeight(int aWeight);
};
```

Die Listings 8.1 und 8.2 bringen eine Neuauflage der Klasse `Cat`. Jetzt aber werden die Funktionen in `cat.hpp` deklariert und in `cat.cpp` implementiert. Listing 8.1 realisiert außerdem sowohl die Zugriffsfunktionen als auch die Funktion `Meow()` als Inline-Funktionen.

Listing 8.1: Die Klassendeklaration von Cat in cat.hpp

```
 1: #include <iostream>
 2:
 3: class Cat
 4: {
 5: public:
 6:     Cat (int initialAge);
 7:     ~Cat();
 8:     int GetAge() { return itsAge;}             // inline!
 9:     void SetAge (int age) { itsAge = age;}     // inline!
10:     void Meow() { std::cout << "Miau.\n";}     // inline!
11: private:
12:     int itsAge;
13: };
```

Listing 8.2: Die Implementierung von Cat in cat.cpp

```
 1: // Zeigt Inline-Funktionen
 2: // und das Einbinden von Header-Dateien
 3: #include "cat.hpp"  // Nicht vergessen, die Header-Dateien
einzubinden!
 4:
 5: Cat::Cat(int initialAge)   // Konstruktor
 6: {
 7:     itsAge = initialAge;
 8: }
 9:
10: Cat::~Cat()               // Destruktor, keine Aktionen
11: {
12: }
13:
14: // Eine Katze erzeugen, Alter festlegen, Miauen lassen,
15: // Alter mitteilen, dann erneut miauen.
16: int main()
17: {
18:     Cat Frisky(5);
```

```
19:     Frisky.Meow();
20:     std::cout << "Frisky ist eine Katze, die " ;
21:     std::cout << Frisky.GetAge() << " Jahre als ist.\n";
22:     Frisky.Meow();
23:     Frisky.SetAge(7);
24:     std::cout << "Jetzt ist Frisky " ;
25:     std::cout << Frisky.GetAge() << " Jahre alt.\n";
26:     return 0;
27: }
```

```
Miau.
Frisky ist eine Katze, die 5 Jahre alt ist.
Miau.
Jetzt ist Frisky 7 Jahre alt.
```

Zeile 8 in Listing 8.1 deklariert die Funktion `GetAge()` und stellt ihre Inline-Implementierung bereit. Die Zeilen 9 und 10 enthalten weitere Inline-Funktionen, wobei aber die Funktionalität dieser Funktionen unverändert aus den vorherigen »Outline«-Implementierungen übernommen wurde.

Die Anweisung `#include "cat.hpp"` in Zeile 3 von Listing 8.2 bindet das Listing von `cat.hpp` ein. Damit wird auch die für `cout` erforderliche Header-Datei `iostream` eingebunden.

8.5 Klassen als Datenelemente einer Klasse

Es ist durchaus üblich, eine komplexe Klasse aufzubauen, indem man einfachere Klassen deklariert und sie in die Deklaration der komplexeren Klasse einbindet. Beispielsweise kann man eine Klasse `Motor`, eine Klasse `Wheel`, eine Klasse `Transmission` usw. deklarieren und dann zur Klasse `Car` kombinieren. Damit deklariert man eine »Hat-ein«-Beziehung: Ein Auto hat einen Motor, es hat Räder, es hat eine Kraftübertragung.

Sehen wir uns ein zweites Beispiel an. Ein Rechteck setzt sich aus Linien zusammen. Eine Linie ist durch zwei Punkte definiert. Ein Punkt ist durch eine X- und eine Y-Koordinate festgelegt. Listing 8.3 zeigt die vollständige Deklaration einer Klasse `Rectangle`, wie sie in `rectangle.hpp` stehen könnte. Da ein Rechteck mit vier Linien definiert ist, die vier Punkte verbinden, und sich jeder Punkt auf eine Koordinate in einem Diagramm bezieht, wird zuerst die Klasse `Point` deklariert, um die X/Y-Koordinaten jedes Punktes aufzunehmen. Listing 8.4 zeigt eine vollständige Deklaration beider Klassen.

Listing 8.3: rect.hpp – Deklaration einer vollständigen Klasse (Header-Datei)

```
// Beginn von rect.hpp
#include <iostream>

class Point     // nimmt X/Y-Koordinaten auf
{
    // Kein Konstruktor, Standardkonstruktor verwenden
public:
    void SetX(int x) { itsX = x; }
    void SetY(int y) { itsY = y; }
    int GetX()const { return itsX;}
    int GetY()const { return itsY;}
private:
    int itsX;
    int itsY;
};  // Ende der Klassendeklaration von Point


class  Rectangle
{
public:
    Rectangle (int top, int left, int bottom, int right);
    ~Rectangle () {}

    int GetTop() const { return itsTop; }
    int GetLeft() const { return itsLeft; }
    int GetBottom() const { return itsBottom; }
    int GetRight() const { return itsRight; }

    Point  GetUpperLeft() const { return itsUpperLeft; }
    Point  GetLowerLeft() const { return itsLowerLeft; }
    Point  GetUpperRight() const { return itsUpperRight; }
    Point  GetLowerRight() const { return itsLowerRight; }

    void SetUpperLeft(Point Location);
    void SetLowerLeft(Point Location);
    void SetUpperRight(Point Location);
    void SetLowerRight(Point Location);

    void SetTop(int top);
    void SetLeft (int left);
    void SetBottom (int bottom);
    void SetRight (int right);

    int GetArea() const;

private:
    Point  itsUpperLeft;
    Point  itsUpperRight;
```

```
49:     Point  itsLowerLeft;
50:     Point  itsLowerRight;
51:     int    itsTop;
52:     int    itsLeft;
53:     int    itsBottom;
54:     int    itsRight;
55: };
56: // Ende von Rect.hpp
```

```
 1: // Beginn von rect.cpp
 2: #include "rect.hpp"
 3:
 4: Rectangle::Rectangle(int top, int left, int bottom, int right)
 5: {
 6:     itsTop = top;
 7:     itsLeft = left;
 8:     itsBottom = bottom;
 9:     itsRight = right;
10:
11:     itsUpperLeft.SetX(left);
12:     itsUpperLeft.SetY(top);
13:
14:     itsUpperRight.SetX(right);
15:     itsUpperRight.SetY(top);
16:
17:     itsLowerLeft.SetX(left);
18:     itsLowerLeft.SetY(bottom);
19:
20:     itsLowerRight.SetX(right);
21:     itsLowerRight.SetY(bottom);
22: }
23:
24: void Rectangle::SetUpperLeft(Point Location)
25: {
26:     itsUpperLeft = Location;
27:     itsUpperRight.SetY(Location.GetY());
28:     itsLowerLeft.SetX(Location.GetX());
29:     itsTop = Location.GetY();
30:     itsLeft = Location.GetX();
31: }
32:
33: void Rectangle::SetLowerLeft(Point Location)
34: {
35:     itsLowerLeft = Location;
36:     itsLowerRight.SetY(Location.GetY());
37:     itsUpperLeft.SetX(Location.GetX());
38:     itsBottom = Location.GetY();
39:     itsLeft = Location.GetX();
40: }
```

Listing 8.4: rect.cpp – Deklaration einer vollständigen Klasse

```
41:
42: void Rectangle::SetLowerRight(Point Location)
43: {
44:     itsLowerRight = Location;
45:     itsLowerLeft.SetY(Location.GetY());
46:     itsUpperRight.SetX(Location.GetX());
47:     itsBottom = Location.GetY();
48:     itsRight = Location.GetX();
49: }
50:
51: void Rectangle::SetUpperRight(Point Location)
52: {
53:     itsUpperRight = Location;
54:     itsUpperLeft.SetY(Location.GetY());
55:     itsLowerRight.SetX(Location.GetX());
56:     itsTop = Location.GetY();
57:     itsRight = Location.GetX();
58: }
59:
60: void Rectangle::SetTop(int top)
61: {
62:     itsTop = top;
63:     itsUpperLeft.SetY(top);
64:     itsUpperRight.SetY(top);
65: }
66:
67: void Rectangle::SetLeft(int left)
68: {
69:     itsLeft = left;
70:     itsUpperLeft.SetX(left);
71:     itsLowerLeft.SetX(left);
72: }
73:
74: void Rectangle::SetBottom(int bottom)
75: {
76:     itsBottom = bottom;
77:     itsLowerLeft.SetY(bottom);
78:     itsLowerRight.SetY(bottom);
79: }
80:
81: void Rectangle::SetRight(int right)
82: {
83:     itsRight = right;
84:     itsUpperRight.SetX(right);
85:     itsLowerRight.SetX(right);
86: }
87:
```

```
 88: int Rectangle::GetArea() const
 89: {
 90:     int Width = itsRight-itsLeft;
 91:     int Height = itsTop - itsBottom;
 92:     return (Width * Height);
 93: }
 94:
 95: // Rechteckflaeche berechnen. Dazu Ecken bestimmen,
 96: // Breite und Höhe ermitteln, dann multiplizieren.
 97: int main()
 98: {
 99:     // Eine lokale Rectangle-Variable initialisieren
100:     Rectangle MyRectangle (100, 20, 50, 80 );
101:
102:     int Area = MyRectangle.GetArea();
103:
104:     std::cout << "Flaeche: " << Area << "\n";
105:     std::cout << "Obere linke X-Koordinate: ";
106:     std::cout << MyRectangle.GetUpperLeft().GetX();
107:     return 0;
108: }
```

```
Flaeche: 3000
Obere linke X-Koordinate: 20
```

Die Zeilen 4 bis 15 in Listing 8.3 deklarieren die Klasse `Point`, die eine bestimmte X/Y-Koordinate in einem Diagramm speichert. In der hier gezeigten Form greift das Programm kaum auf `Point` zurück. Allerdings ist `Point` auch für andere Zeichenmethoden vorgesehen.

Die Klassendeklaration von `Point` deklariert in den Zeilen 13 und 14 zwei Elementvariablen – `itsX` und `itsY`. Diese Variablen nehmen die Werte der Koordinaten auf. Bei Vergrößerung der X-Koordinate verschiebt sich die Zeichnung im Diagramm nach rechts. Mit wachsender Y-Koordinate wandert die Zeichnung im Diagramm nach oben. (Es gibt auch Programme, die bei Vergrößerung der Y-Koordinate die Zeichnung im Fenster nach unten verschieben.)

Die Klasse `Point` verwendet Inline-Zugriffsfunktionen, um die in den Zeilen 8 bis 11 deklarierten Punkte `X` und `Y` zu lesen und zu setzen. Die Klasse `Point` übernimmt den Standardkonstruktor und den Standarddestruktor vom Compiler. Daher muss man die Koordinaten der Punkte explizit festlegen.

In Zeile 18 beginnt die Deklaration der Klasse `Rectangle`. Ein `Rectangle`-Objekt besteht aus vier Punkten, die die Ecken des Rechtecks darstellen.

Der Konstruktor für die Klasse `Rectangle` (Zeile 21) übernimmt vier Ganzzahlen, die als `top` (oben), `left` (links), `bottom` (unten) und `right` (rechts) bezeichnet sind. Die Klasse `Rectangle` kopiert die an den Konstruktor übergebenen Parameter in vier Elementvariablen und richtet dann vier `Point`-Objekte ein.

Außer den üblichen Zugriffsfunktionen verfügt `Rectangle` über eine Funktion namens `GetArea()`, die in Zeile 44 deklariert ist. Die Klasse speichert die Fläche nicht in einer Variablen, sondern lässt sie berechnen. Diese Aufgabe realisiert die Funktion `GetArea()` in den Zeilen 90 bis 92 von Listing 8.4. Die Funktion ermittelt zunächst Breite und Höhe des Rechtecks und multipliziert dann beide Werte.

Um die X-Koordinate der oberen linken Ecke des Rechtecks zu ermitteln, muss man auf den Punkt `UpperLeft` zugreifen und von diesem Punkt den Wert für `X` abrufen. `GetUpperLeft()` ist eine Methode von `Rectangle` und kann somit direkt auf die privaten Daten von `Rectangle` (einschließlich `itsUpperLeft`) zugreifen. Da `itsUpperLeft` ein `Point`-Objekt und der Wert `itsX` von `Point` als privat deklariert ist, stehen `GetUpperLeft()` diese Daten nicht direkt zur Verfügung. Stattdessen muss man die öffentliche Zugriffsfunktion `GetX()` aufrufen, um diesen Wert zu erhalten.

In Zeile 97 von Listing 8.4 beginnt der Rumpf des eigentlichen Programms. Bis Zeile 100 ist praktisch noch nichts passiert. Es hat nicht einmal eine Reservierung von Speicher stattgefunden. Man hat dem Compiler bislang lediglich mitgeteilt, wie ein `Point`-Objekt und ein `Rectangle`-Objekt zu erzeugen sind, falls man diese Objekte irgendwann benötigt.

Zeile 100 definiert ein `Rectangle`-Objekt mit Übergabe der Werte für `top`, `left`, `bottom` und `right`.

Zeile 101 deklariert die lokale Variable `Area` vom Typ `int`. Diese Variable nimmt die Fläche des erzeugten `Rectangle`-Objekts auf. `Area` wird mit dem von der Funktion `GetArea()` des Objekts `Rectangle` zurückgegebenen Wert initialisiert.

Ein Klient von `Rectangle` kann ein `Rectangle`-Objekt erzeugen und dessen Fläche berechnen, ohne die Implementierung von `GetArea()` zu kennen.

Allein aus der Header-Datei mit der Deklaration der Klasse `Rectangle` kann der Programmierer ersehen, dass `GetArea()` einen `int`-Wert zurückgibt. Wie `GetArea()` diese magische Berechnung ausführt, ist für den Benutzer der Klasse `Rectangle` nicht von Belang. Im Prinzip könnte der Autor von `Rectangle` die Funktion `GetArea()` ändern, ohne die Programme zu beeinflussen, die auf die Klasse `Rectangle` zugreifen.

8.6 Fragen und Antworten

F: Wenn ich mit einer konstanten Funktion eine Klasse ändere und dadurch einen Compiler-Fehler hervorrufe, warum sollte ich dann nicht einfach das Schlüsselwort `const` weglassen und Compiler-Fehlern aus dem Wege gehen?

A: Wenn Ihre Elementfunktion aufgrund der Logik die Klasse nicht ändern sollte, stellt die Verwendung des Schlüsselwortes `const` eine gute Möglichkeit dar, den Compiler beim Aufspüren verzwickter Fehler hinzuzuziehen. Wenn zum Beispiel `GetAge()` keinen Grund hat, die Klasse `Cat` zu ändern, aber Ihre Implementierung die Zeile

```
if (itsAge = 100) cout << "Hey! Sie sind 100 Jahre alt\n";
```

enthält, führt die Deklaration von `GetAge()` als `const` zu einem Compiler-Fehler. Hier war beabsichtigt, einen Test durchzuführen, ob `itsAge` gleich 100 ist. Stattdessen haben Sie versehentlich den Wert `100` an `itsAge` zugewiesen. Diese Zuweisung ändert die Klasse. Da Sie aber explizit keine Änderungen zugelassen haben, kann der Compiler den Fehler ermitteln.

Derartige Fehler lassen sich nur schwer aufspüren, wenn man den Code lediglich durchsucht – das Auge sieht oft nur das, was es sehen will. Vor allem aber scheint das Programm korrekt zu laufen, obwohl `itsAge` nun auf eine falsche Zahl gesetzt wurde. Früher oder später treten dann Probleme auf.

F: Gibt es überhaupt einen Grund, in einem C++-Programm eine Struktur zu verwenden?

A: Viele C++-Programmierer reservieren das Schlüsselwort `struct` für Klassen ohne Funktionen. Das ist ein Rückfall in die alten C-Strukturen, die keine Funktionen haben konnten. Offen gesagt, sind Strukturen nur verwirrend und gehören nicht zum guten Programmierstil. Die heutige Struktur ohne Methoden braucht morgen vielleicht eine Methode. Dann ist man gezwungen, entweder den Typ in `class` zu ändern oder die Regel zu durchbrechen und schließlich eine Struktur mit Methoden zu erzeugen.

8.7 Workshop

Nachdem Sie in dieser Stunde mehr über Klassen gelernt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu festigen.

8.7.1 Kontrollfragen

1. Warum bringt man Klassendefinitionen in separaten Dateien unter?
2. Welchen Zweck erfüllt der Konstruktor in `cat.cpp`?
3. Was passiert in `rect.cpp` und `rect.hpp`, wenn die Klasse `Point` nicht definiert ist?
4. Woher weiß der Compiler, wo Ihre Include-Dateien zu finden sind?

8.7.2 Übungen

1. Spalten Sie die Klasse `Point` aus `rect.hpp` in eine eigene Datei ab und binden Sie die neue Datei in `rect.hpp` ein. Wie wirkt sich das auf die Kompilierung von `rect.cpp` aus? Ändern sich die Ergebnisse?
2. Machen Sie `GetAge()` in `cat.hpp` zu einer `const`-Methode. Muss noch etwas anderes geändert werden?
3. Ändern Sie `cat.cpp`, um ein zweites `cat`-Objekt (namens »Charly«) zu erstellen. Können Sie Charly dazu bringen, zu miauen und älter zu werden?

8.7.3 Antworten zu den Kontrollfragen

1. Damit können Sie sie auch anderen Programmierern verfügbar machen (und von diesen Objekte übernehmen).
2. Er initialisiert die Variable `itsAge` auf den übergebenen Wert, wenn das Objekt instanziert wird. Dadurch ersparen Sie sich den Schritt, ein Objekt zu deklarieren und dann `SetAge()` aufzurufen, um das Alter festzulegen.
3. Unter der Annahme, dass `Point` nirgendwo sonst definiert ist, erhalten Sie einen Compiler-Fehler für eine nicht definierte Referenz. Falls die Klasse an einer anderen Stelle definiert ist, müssen Sie zusätzliche Dateien einbinden (beispielsweise `point.hpp`). Es ist durchaus üblich, dass sich Klassen auf andere Klassen stützen und darauf aufbauen.
4. Ist der Include-Dateiname in doppelte Anführungszeichen (`"`*`Dateiname`*`"`) eingeschlossen, sucht der Compiler im selben Verzeichnis wie die aktuelle Quelldatei. Man kann den Compiler auch anweisen, in anderen Verzeichnissen zu suchen. Gibt man einen Pfad in doppelten Anführungszeichen an, sucht der Compiler nur in diesem Verzeichnis. Wenn der Include-Dateiname in spitze Klammern eingeschlossen ist (`<`*`Dateiname`*`>`), sucht der Compiler im Include-Verzeichnis des Systems.

KAPITEL 9

jetzt lerne ich

Zeiger

In dieser Stunde lernen Sie,

- was Zeiger sind,
- wie man Zeiger deklariert und verwendet,
- was ein Heap ist und wie man den Speicher manipuliert.

9.1 Zeiger und ihre Verwendung

Mit Zeigern (Pointern) lässt sich der Computer-Speicher direkt manipulieren. Damit gehören Zeiger einerseits zu den leistungsfähigsten Werkzeugen in der Hand des C++-Programmierers, andererseits aber auch zu den schwierigsten Aspekten von C++. Wenn man sich aber etwas mit den Grundlagen der Zeiger beschäftigt, dürfte der Umgang damit nicht weiter schwer fallen.

Dieses Kapitel erläutert Schritt für Schritt wie Zeiger funktionieren. Beachten Sie dabei aber bitte, dass der eigentliche Aha-Effekt erst mit fortschreitender Programmiertätigkeit (auf das Buch bezogen: mit späteren Kapiteln) zutage tritt.

Ein *Zeiger* ist eine Variable, die eine Speicheradresse aufnimmt.

Stopp. Lesen Sie das bitte noch einmal. Ein Zeiger ist eine Variable. Sie wissen bereits, was eine Variable ist: ein Objekt, das einen Wert aufnehmen kann. Eine Integer-Variable nimmt eine ganze Zahl auf. Eine Zeichenvariable

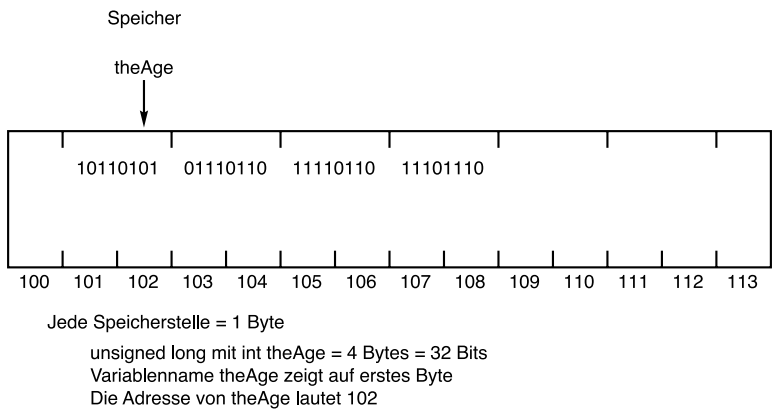
speichert einen Buchstaben. Ein Zeiger ist eine Variable, die eine Speicheradresse aufnimmt.

Was ist aber nun eine Speicheradresse? Um das voll und ganz zu verstehen, muss man ein wenig über den Speicher des Computers wissen. Keine Angst, es ist nicht schwierig.

Der Hauptspeicher des Computers nimmt alle Daten und Programme auf. Per Konvention teilt man den Hauptspeicher in fortlaufend nummerierte Speicherzellen ein. Diese Nummern sind die so genannten Speicheradressen, unter denen die Speicherstellen ansprechbar sind.

Jede Variable eines bestimmten Typs befindet sich an einer eindeutig adressierbaren Speicherstelle. Abbildung 9.1 zeigt schematisch die Speicherung der Integer-Variablen `theAge` vom Typ `unsigned long`.

Abbildung 9.1: Schematische Darstellung von theAge im Hauptspeicher



Zeichnung Seite 148

Die Adressierung des Speichers unterscheidet sich bei den verschiedenen Computer-Typen. Normalerweise braucht der Programmierer die konkrete Adresse einer bestimmten Variablen gar nicht zu kennen, da sich der Compiler um die Einzelheiten kümmert. Wenn man allerdings diese Information haben möchte, kann man sie mithilfe des Adressoperators (`&`) ermitteln, wie es Listing 9.1 zeigt.

Listing 9.1: address-demo.cpp – Adressen von Variablen

```
1: // Listing 9.1 Demonstriert Adressoperator und
2: // Adressen von lokalen Variablen
3: #include <iostream>
4:
5: int main()
6: {
```

```
 7:     unsigned short shortVar=5;
 8:     unsigned long  longVar=65535;
 9:     long sVar = -65535;
10:
11:     std::cout << "shortVar:\t" << shortVar;
12:     std::cout << "\tAdresse von shortVar:\t" << &shortVar << "\n";
13:     std::cout << "longVar:\t"  << longVar;
14:     std::cout << "\tAdresse von longVar:\t"  << &longVar  << "\n";
15:     std::cout << "sVar:\t\t"   << sVar;
16:     std::cout << "\tAdresse von sVar:\t"     << &sVar     << "\n";
17:
18:     return 0;
19: }
```

Sonderzeichen in C++

In Listing 9.1 bewirkt die Zeichenfolge `\t`, dass ein Tabulatorzeichen in die Ausgabe eingefügt wird. Damit lässt sich eine tabellarische Ausgabe in Spalten realisieren (zwar einfach, aber bei weitem nicht perfekt).

Die Zeichenfolge `\n` verschiebt den Cursor auf die nächste Zeile (entspricht der Funktion `std::endl`).

Mit `\\` können Sie einen Backslash ausgeben (aus einer Zeichenfolge heraus).

`\"` gibt doppelte Anführungszeichen aus (aus einer Zeichenfolge heraus).

`\'` liefert ein einfaches Anführungszeichen (aus einer Zeichenfolge heraus oder als einzelnes Zeichen).

Hier dient der Backslash als so genanntes *Escape-Zeichen*, das die Bedeutung des nächsten Zeichens von einem normalen Druckzeichen zu einem Sonderzeichen ändert (`\n` gibt eine Zeilenschaltung aus und nicht den Buchstaben `n`).

```
shortVar: 5             Adresse von shortVar: 1245066
longVar:  65535         Adresse von longVar:  1245060
sVar:     -65535        Adresse von sVar:     1245056
```

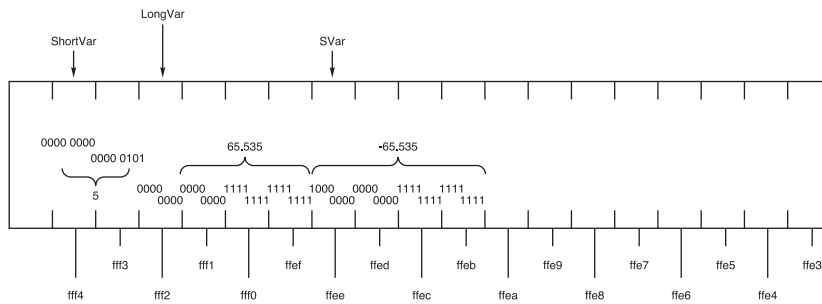
Die Ausgabe kann bei Ihrem Computer ein etwas anderes Aussehen haben, je nachdem, was noch im Speicher steht und wie viel Speicher verfügbar ist. Beispielsweise ist auch Folgendes möglich:

```
shortVar: 5             Adresse von shortVar: 0x8fc9:fff4
longVar:  65535         Adresse von longVar:  0x8fc9:fff2
sVar:     -65535        Adresse von sVar:     0x8fc9:ffee
```

Das Programm deklariert und initialisiert drei Variablen: eine vom Typ `short` in Zeile 7, eine `unsigned long` in Zeile 8 und eine `long` in Zeile 9. Die Werte und Adressen dieser Variablen gibt das Programm in den Zeilen 11 bis 16 mithilfe des Adressoperators (`&`) aus.

Der Wert von `shortVar` lautet (wie erwartet) 5, und die Adresse bei Ausführung des Programms mit dem Borland-Compiler auf dem x86-Computer des Autors ist `1245066`. Diese Adresse hängt vom jeweiligen Computer ab und kann bei einem neuen Programmstart etwas anders aussehen. Was sich allerdings nicht ändert, sind die Differenzen zwischen den einzelnen Adressen, denn der Compiler legt die (relativen) Positionen der Variablen im Programm anhand der durch die Datentypen bestimmten Größen fest. Auf älteren Systemen können `short` Integer-Zahlen 2 Bytes belegen, auf neueren Systemen sind es meistens 4 Bytes. Abbildung 9.2 zeigt ein Beispiel, wie das Programm die Variablen im Speicher ablegt.

Abbildung 9.2: Speicherung von Variablen



Es ist eigentlich gar nicht notwendig, den tatsächlichen numerischen Wert der Adresse jeder Variablen zu kennen. Wichtig ist einzig, dass jede Variable eine Adresse *hat* und die entsprechende Anzahl von Bytes im Speicher reserviert sind.

Woher weiß nun der Compiler, wie viel Speicher eine bestimmte Variable benötigt? Man teilt das dem Compiler durch die Deklaration des Variablentyps mit.

Wenn man also eine Variable vom Typ `unsigned long` deklariert, weiß der Compiler, dass 4 Bytes im Speicher zu reservieren sind, da jede Variable dieses Typs 4 Bytes benötigt. Der Compiler übernimmt auch die Zuweisung der tatsächlichen Adresse.

Wird ein Zeiger reserviert, weist der Compiler ausreichend Speicher zu, damit die Zeigervariable eine Adresse für die jeweilige Hardware und Betriebssystemumgebung aufnehmen kann. Die Größe eines Zeigers kann – muss aber nicht – die gleiche Größe wie eine Ganzzahl haben. Achten Sie einfach darauf, dass Sie in Ihrem Programm keine Annahmen über die Größe von Zeigern treffen.

9.1.1 Die Adresse in einem Zeiger speichern

Jede Variable hat eine Adresse. Selbst ohne Kenntnis der genauen Adresse einer bestimmten Variablen kann man diese Adresse in einem Zeiger speichern.

Nehmen wir zum Beispiel die Variable `howOld` als Integer an. Um einen Zeiger namens `pAge` zur Aufnahme ihrer Adresse zu deklarieren, schreibt man

```
int *pAge = NULL
```

Damit deklariert man `pAge` als Zeiger auf `int`. Das heißt, `pAge` wird für die Aufnahme der Adresse einer `int`-Zahl deklariert.

Beachten Sie, dass `pAge` eine Variable wie jede andere Variable ist. Wenn man eine Integer-Variable (vom Typ `int`) deklariert, wird sie für die Aufnahme einer ganzen Zahl eingerichtet. Ein Zeiger ist einfach ein spezieller Typ einer Variablen, die man für die Aufnahme der Adresse eines bestimmten Objekts im Speicher einrichtet. In diesem Beispiel nimmt `pAge` die Adresse einer Integer-Variablen auf.

Man deklariert den Typ der Variablen, auf die der Zeiger verweisen soll. Damit sagt man dem Compiler, wie der Speicher an der Stelle zu behandeln ist, auf die der Zeiger verweist. Der Zeiger selbst enthält eine Adresse.

Die obige Anweisung initialisiert `pAge` mit `NULL`. Ein Zeiger mit dem Wert `NULL` heißt *Null-Zeiger*. Alle Zeiger sollte man bei ihrer Erzeugung initialisieren. Wenn man nicht weiß, was man dem Zeiger zuweisen soll, wählt man einfach den Wert `NULL`. Ein nicht initialisierter Zeiger ist ein so genannter *wilder Zeiger*. Wie der Name vermuten lässt, sind derartige Zeiger gefährlich.

Zeiger kann man auch mit 0 initialisieren:

```
int *pAge = 0
```

Das Ergebnis ist das gleiche wie bei der Initialisierung mit `NULL`, wobei aber aus technischer Sicht die `0` eine Ganzzahlkonstante und `NULL` eine Konstante für die Adresse 0 ist.

Initialisieren Sie Zeiger!

Praktizieren Sie sichere Programmierung: Initialisieren Sie Zeiger!

Wenn man den Zeiger mit `0` oder `NULL` initialisiert, muss man explizit die Adresse von `howOld` an `pAge` zuweisen. Das lässt sich beispielsweise wie folgt realisieren:

```
unsigned short int howOld = 50;    // eine Integer-Variable erzeugen
unsigned short int * pAge = 0;     // einen Zeiger erzeugen
pAge = &howOld;                    // die Adresse von howOld in pAge
speichern
```

Die erste Zeile erzeugt die Variable `howOld` mit dem Typ `unsigned short int` und initialisiert sie mit dem Wert `50`. Die zweite Zeile deklariert `pAge` als Zeiger auf den Typ `unsigned short int` und weist ihr den Anfangswert `0` zu. Das Sternchen (`*`) nach dem Variablentyp und vor dem Variablennamen kennzeichnet `pAge` als Zeiger.

Die dritte und letzte Zeile weist die Adresse von `howOld` an die Zeigervariable `pAge` zu. Die Zuweisung einer Adresse kennzeichnet der Adressoperator (`&`). Ohne diesen Operator hätte man den *Wert* von `howOld` zugewiesen. Auch wenn es sich dabei um eine gültige Adresse handeln sollte, hat diese wahrscheinlich nichts mit der beabsichtigten Adresse gemein.

Zeiger und Nicht-Zeiger

Ein häufig gemachter Fehler ist es, einen Nicht-Zeiger an eine Zeigervariable zuzuweisen. Der Compiler sollte Sie davor warnen.

Nunmehr enthält `pAge` als Wert die Adresse von `howOld`. Die Variable `howOld` hat ihrerseits den Wert `50`. Das Ganze kann man auch mit einem Schritt weniger erreichen:

```
unsigned short int howOld = 50;       // eine Variable erzeugen
unsigned short int * pAge = &howOld;  // einen Zeiger auf howOld
erzeugen
```

`pAge` ist ein Zeiger, der nun die Adresse der Variablen `howOld` enthält. Mittels `pAge` kann man tatsächlich den Wert von `howOld` bestimmen, der im Beispiel `50` lautet. Den Zugriff auf `howOld` über den Zeiger `pAge` nennt man *Umleitung* (Indirektion), da man indirekt auf `howOld` mithilfe von `pAge` zugreift. Später in diesem Kapitel erfahren Sie, wie man mit der Indirektion auf den Wert einer Variablen zugreift.

Indirektion bedeutet den Zugriff auf den Wert an der im Zeiger gespeicherten Adresse. Der Zeiger stellt einen indirekten Weg bereit, um den an dieser Adresse abgelegten Wert zu erhalten.

9.1.2 Zeigernamen

Zeiger können jeden Namen erhalten, der auch für andere Variablen gültig ist. Dieses Buch folgt den Konventionen zur Benennung von Zeigern mit einem vorangestellten `p` wie in `pAge` und `pNumber` (`p` steht für pointer – Zeiger).

9.1.3 Der Indirektionsoperator

Den Indirektionsoperator (*) bezeichnet man auch als *Dereferenzierungsoperator*. Wenn ein Zeiger dereferenziert wird, ruft man den Wert an der im Zeiger gespeicherten Adresse ab.

Normale Variablen erlauben direkten Zugriff auf ihre eigenen Werte. Wenn man eine neue Variable vom Typ `unsigned short int` namens `yourAge` erzeugt und den Wert in `howOld` dieser neuen Variablen zuweisen möchte, schreibt man:

```
unsigned short int howOld = 50;
unsigned short int yourAge;
yourAge = howOld;
```

Ein Zeiger bietet indirekten Zugriff auf den Wert der Variablen, deren Adresse er speichert. Der Wert in `howOld` lässt sich der neuen Variablen `yourAge` mithilfe des Zeigers `pAge` folgendermaßen zuweisen:

```
unsigned short int howOld = 50;      // die Variable howOld erzeugen
unsigned short int * pAge = &howOld; // pAge zeigt auf die Adresse
von howOld
unsigned short int yourAge;          // eine andere Variable erzeugen
yourAge = *pAge;                     // Wert bei pAge (50) an yourAge
zuweisen
```

Der Indirektionsoperator (*) vor der Variablen `pAge` bedeutet »der an Adresse ... gespeicherte Wert«. Die Zuweisung ist wie folgt zu lesen: »Nimm den Wert, der an der in `pAge` gespeicherten Adresse steht, und weise ihn `yourAge` zu.« Man kann das auch wie folgt ausdrücken: »Beeinflusse nicht den Zeiger, sondern das Element, das an der im Zeiger gespeicherten Adresse steht.«

Indirektionsoperationen

Der Indirektionsoperator (*) kommt bei Zeigern in zwei unterschiedlichen Versionen vor: Deklaration und Dereferenz. Bei der Deklaration eines Zeigers gibt das Sternchen an, dass es sich um einen Zeiger und nicht um eine normale Variable handelt. Dazu ein Beispiel:

```
unsigned short * pAge = NULL; // einen Zeiger auf unsigned short
erzeugen
```

Wenn der Zeiger dereferenziert wird, gibt der Indirektionsoperator an, dass auf den Wert an der im Zeiger abgelegten Speicheradresse zuzugreifen ist und nicht auf die Adresse selbst:

```
*pAge = 5; // Der Speicherzelle, deren Adresse in pAge steht, den
Wert 5 zuweisen
```

Beachten Sie auch, dass C++ das gleiche Zeichen (*) als Multiplikationsoperator verwendet. Der Compiler erkennt aus dem Kontext, welcher Operator gemeint ist.

9.1.4 Zeiger, Adressen und Variablen

Um Probleme mit Zeigern zu vermeiden, muss man

- Zeiger,
- die vom Zeiger gespeicherte Adresse und
- den Wert an der im Zeiger enthaltenen Adresse

sicher auseinanderhalten können.

Sehen Sie sich das folgende Codefragment an:

```
int theVariable = 5;
int * pPointer = &theVariable ;
```

Die Variable `theVariable` ist als Integer-Variable deklariert und mit dem Wert 5 initialisiert. `pPointer` ist als Zeiger auf einen Integer deklariert und wird mit der Adresse von `theVariable` initialisiert. `pPointer` ist der Zeiger. Die von `pPointer` gespeicherte Adresse ist die Adresse von `theVariable`. Der Wert an der von `pPointer` gespeicherten Adresse lautet `5`. Abbildung 9.3 zeigt eine schematische Darstellung von `theVariable` und `pPointer`.

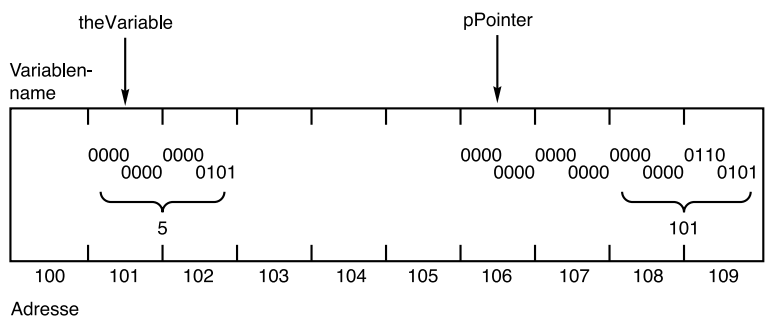


Abbildung 9.3: Schematische Darstellung des Speichers

9.1.5 Daten mithilfe von Zeigern manipulieren

Nachdem man einem Zeiger die Adresse einer Variablen zugewiesen hat, kann man mit diesem Zeiger auf die Daten in der Variablen zugreifen. Listing 9.2 demonstriert, wie der Zugriff auf eine lokale Variable mittels eines Zeigers vonstatten geht und wie der Zeiger die Werte in dieser Variablen manipuliert.

Listing 9.2: pointeruse.cpp – Daten mithilfe von Zeigern manipulieren

```
// Listing 9.2 Verwendung von Zeigern
#include <iostream>
using std::cout; // Diese Datei verwendet std::cout

int main()
{
    int myAge;            // eine Variable
    int * pAge = NULL;    // ein Zeiger

    myAge = 5;
    pAge = &myAge;     // Adresse von myAge an pAge zuweisen
    cout << "myAge: " << myAge << "\n";
    cout << "*pAge: " << *pAge << "\n\n";

    cout << "*pAge = 7\n";
    *pAge = 7;         // myAge auf 7 setzen
    cout << "*pAge: " << *pAge << "\n";
    cout << "myAge: " << myAge << "\n\n";

    cout << "myAge = 9\n";
    myAge = 9;
    cout << "myAge: " << myAge << "\n";
    cout << "*pAge: " << *pAge << "\n";

    return 0;
}
```

```
myAge: 5
*pAge: 5

*pAge = 7
*pAge: 7
myAge: 7

myAge = 9
myAge: 9
*pAge: 9
```

Dieses Programm deklariert zwei Variablen: `myAge` vom Typ `int` und `pAge` als Zeiger auf eine Variable vom Typ `int`. Der Zeiger nimmt die Adresse von `myAge` auf. Zeile 10 weist der Variablen `myAge` den Wert `5` zu. Die Ausgabe in Zeile 12 dient der Kontrolle der Zuweisung.

Zeile 11 weist `pAge` die Adresse von `myAge` zu. Zeile 13 dereferenziert `pAge` und gibt das erhaltene Ergebnis aus. Es zeigt, dass der Wert an der in `pAge` gespeicherten Adresse der in `myAge` abgelegte Wert `5` ist. Zeile 16 weist den Wert `7` der Variablen zu, die sich an der in `pAge` gespeicherten Adresse befindet. Damit setzt man `myAge` auf `7`. Eine Bestätigung liefern die Ausgabeanweisungen in den Zeilen 17 und 18.

Zeile 21 weist der Variablen `myAge` den Wert `9` zu. Diesen Wert erhält man direkt in Zeile 22 und indirekt – durch Dereferenzierung von `pAge` – in Zeile 23.

9.1.6 Adressen untersuchen

Zeiger erlauben die Manipulation von Adressen, ohne dass man deren eigentlichen Wert kennt. Nach der heutigen Stunde wissen Sie genau, dass der Wert eines Zeigers die Adresse der Variablen darstellt, wenn man deren Adresse an den Zeiger zuweist. Überprüfen wir das gleich einmal an einem Beispiel. Listing 9.3 zeigt ein entsprechendes Programm.

Listing 9.3: pointerstore.cpp – Untersuchen, was in einem Zeiger gespeichert ist

```
 1:  // Listing 9.3 Was in einem Zeiger gespeichert ist
 2: #include <iostream>
 3:
 4: int main()
 5: {
 6:     unsigned short int myAge = 5, yourAge = 10;
 7:     unsigned short int * pAge = &myAge;  // ein Zeiger
 8:
 9:     std::cout << "myAge:\t" << myAge;
10:     std::cout << "\t\tyourAge:\t" << yourAge << "\n";
11:     std::cout << "&myAge:\t" << &myAge;
12:     std::cout << "\t&yourAge:\t" << &yourAge <<"\n";
13:
14:     std::cout << "pAge:\t" << pAge << "\n";
15:     std::cout << "*pAge:\t" << *pAge << "\n\n";
16:
17:     pAge = &yourAge;       // den Zeiger erneut zuweisen
18:
19:     std::cout << "myAge:\t" << myAge;
20:     std::cout << "\t\tyourAge:\t" << yourAge << "\n";
21:     std::cout << "&myAge:\t" << &myAge;
22:     std::cout << "\t&yourAge:\t" << &yourAge <<"\n";
23:
```

```
24:     std::cout << "pAge:\t" << pAge << "\n";
25:     std::cout << "*pAge:\t" << *pAge << "\n\n";
26:
27:     std::cout << "&pAge:\t" << &pAge << "\n";
28:     return 0;
29: }
```

```
myAge:      5             yourAge:  10
&myAge:     1245066       &yourAge: 1245064
pAge:       1245066
*pAge:      5
myAge:      5             yourAge:  10
&myAge:     1245066       &yourAge: 1245064
pAge:       1245064
*pAge:      10
&pAge:      1245060
```

Die Ausgabe kann bei Ihrem Computer ein etwas anderes Aussehen haben, je nachdem, was noch im Speicher steht und wie viel Speicher verfügbar ist. Beispielsweise ist auch Folgendes möglich:

```
myAge:      5             yourAge:  10
&myAge:     0x355C        &yourAge: 0x355E
pAge:       0x355C
*pAge:      5
myAge:      5             yourAge:  10
&myAge:     0x355C        &yourAge: 0x355E
pAge:       0x355E
*pAge:      10
&pAge:      0x355A
```

Zeile 6 deklariert die Variablen `myAge` und `yourAge` vom Typ `unsigned short int`. Zeile 7 deklariert `pAge` als Zeiger auf eine ganze Zahl vom Typ `unsigned short` und initialisiert den Zeiger mit der Adresse der Variablen `myAge`.

Die Zeilen 9 bis 12 geben die Werte und Adressen von `myAge` und `yourAge` aus. Zeile 14 zeigt den Inhalt von `pAge` an, wobei es sich um die Adresse von `myAge` handelt. Zeile 15 dereferenziert `pAge` und zeigt das Ergebnis an: den Wert an der in `pAge` verzeichneten Adresse, d.h. den Wert in `myAge`, also `5`.

Das Wesen der Zeiger dürfte nun klar sein. Zeile 14 zeigt, dass `pAge` die Adresse von `myAge` speichert, und Zeile 15 zeigt, wie man den in `myAge` gespeicherten Wert durch Dereferenzierung des Zeigers `pAge` ermittelt. Bevor Sie im Stoff fortfahren, sollten Ihnen diese Dinge in Fleisch und Blut übergegangen sein. Studieren Sie den Code, und sehen Sie sich die Ausgaben an.

In Zeile 17 findet eine Neuzuweisung von `pAge` statt, sodass diese Variable (Zeiger) nun auf die Adresse von `yourAge` weist. Die Werte und Adressen werden auch hier wieder ausgegeben. Anhand der Ausgabe kann man ablesen, dass `pAge` nun die Adresse der Variablen `yourAge` enthält und dass die Dereferenzierung den Wert in `yourAge` holt.

Zeile 25 gibt die Adresse von `pAge` selbst aus. Wie jede Variable hat auch `pAge` eine Adresse, und diese Adresse lässt sich in einem Zeiger speichern. (Auf die Zuweisung der Adresse eines Zeigers an einen anderen Zeiger gehen wir in Kürze ein.)

Was Sie tun sollten

Verwenden Sie den Indirektionsoperator (*), um auf die Daten an einer in einem Zeiger enthaltenen Adresse zuzugreifen.

Initialisieren Sie alle Zeiger entweder mit einer gültigen Adresse oder mit `NULL` (`0`).

Denken Sie an den Unterschied zwischen der Adresse in einem Zeiger und dem Wert an dieser Adresse.

Versuchen Sie nicht, eine Adresse in einer Nicht-Zeiger-Variablen (beispielsweise einer Integer-Variablen) zu speichern.

9.2 Warum man Zeiger verwendet

Bisher haben Sie in einzelnen Schritten die Zuweisung der Adresse einer Variablen an einen Zeiger kennen gelernt. In der Praxis wird das allerdings in dieser Form kaum vorkommen. Warum plagt man sich überhaupt mit einem Zeiger herum, wenn man bereits eine Variable mit Zugriff auf deren Wert hat? Diese Art der Zeigermanipulation einer automatischen Variablen sollte hier nur die Arbeitsweise von Zeigern demonstrieren.

Nachdem Sie nun mit der Syntax der Zeiger vertraut sind, können Sie sich den echten Einsatzfällen zuwenden. Vorwiegend kommen dabei drei Aufgaben in Betracht:

- Verwalten von Daten auf dem Heap
- Zugriff auf Datenelemente und Funktionen einer Klasse
- Übergabe von Variablen als Referenz an Funktionen

Das restliche Kapitel konzentriert sich auf die Verwaltung von Daten auf dem Heap und den Zugriff auf Datenelemente und Funktionen einer Klasse. In Stunde 11 lernen Sie, wie man Variablen als Referenz übergibt.

9.3 Stack und Heap

Der Programmierer hat es im Allgemeinen mit fünf Speicherbereichen zu tun:

- Bereich der globalen Namen
- Heap
- Register
- Codebereich
- Stack

Lokale Variablen befinden sich zusammen mit den Funktionsparametern auf dem Stack. Der Code steht natürlich im Codebereich und globale Variablen stehen im globalen Namensbereich. Die Register dienen internen Verwaltungsaufgaben. Dazu gehören beispielsweise Stack-Operationen und die Steuerung des Programmablaufs. Der verbleibende Speicher geht an den so genannten *Heap* (auch Freispeicher genannt).

Leider sind lokale Variablen nicht dauerhaft verfügbar: Bei der Rückkehr aus einer Funktion werden die Variablen verworfen. Globale Variablen lösen dieses Problem zum Preis des uneingeschränkten Zugriffs durch das gesamte Programm hindurch. Dabei entsteht Code, der schwer zu verstehen und zu warten ist. Bringt man die Daten auf den *Heap*, umgeht man beide Nachteile.

Man kann sich den Heap als massiven Speicherbereich vorstellen, in dem Tausende von fortlaufend nummerierten Fächern auf Ihre Daten warten. Genau wie beim Stack lassen sich diese Fächer aber nicht benennen. Man muss die Adresse des zu reservierenden Fachs abrufen und diese Adresse dann in einem Zeiger festhalten.

Als Analogie dazu folgendes Beispiel: Ein Freund gibt Ihnen die gebührenfreie Rufnummer für Acme Mail Order. Sie gehen nach Hause und programmieren Ihr Telefon mit dieser Nummer. Dann schmeißen Sie den Zettel weg, auf dem Sie sich die Nummer notiert hatten. Wenn Sie die Kurzwahltaste drücken, läutet das Telefon irgendwo und Acme Mail Order antwortet. An die Nummer können Sie sich nicht erinnern und Sie wissen auch nicht, wo sich das andere Telefon befindet. Aber die Kurzwahltaste gibt Ihnen den Zugriff zu Acme Mail Order. Vergleichbar mit Acme Mail Order sind die Daten auf Ihrem Heap. Sie wissen nicht, wo sie abgelegt sind, aber Sie wissen, wie man sie holen kann. Man greift darauf mit der Adresse zu – im obigen Beispiel mit der Telefonnummer. Diese Nummer müssen Sie aber nicht kennen, Sie brauchen sie nur

in einem Zeiger abzulegen – d.h. der Kurzwahltaste zuzuordnen. Der Zeiger gibt Ihnen den Zugriff auf Ihre Daten, ohne dass Sie sich um die Details kümmern müssen.

Wenn das Programm aus einer Funktion zurückkehrt, räumt es automatisch den Stack auf. Alle lokalen Variablen verlieren ihren Gültigkeitsbereich und verschwinden vom Stack. Den Heap räumt ein Programm erst am Ende auf. Es liegt in Ihrer Verantwortung, reservierten Speicher freizugeben, wenn Sie ihn nicht mehr benötigen.

Der Vorteil des Heap liegt darin, dass der reservierte Speicher verfügbar bleibt, bis man ihn explizit freigibt. Wenn man Speicher auf dem Heap reserviert, während das Programm eine Funktion abarbeitet, bleibt der Speicher nach Rückkehr aus der Funktion weiterhin zugänglich.

Gegenüber globalen Variablen bringt ein derartiger Zugriff auf den Speicher den Vorteil, dass nur Funktionen mit Zugriff auf den Zeiger auch auf die Daten zugreifen können. Damit hat man eine gut kontrollierte Schnittstelle zu den Daten und vermeidet die Probleme, dass fremde Funktionen die Daten in unerwarteter Weise verändern.

Damit das Ganze funktioniert, muss man einen Zeiger auf einen Bereich im Heap erzeugen und den Zeiger an Funktionen übergeben können. Die folgenden Abschnitte erläutern diese Vorgehensweise.

9.3.1 Das Schlüsselwort new

In C++ weist man Speicher im Heap mit dem Schlüsselwort `new` zu. An `new` schließt sich der Typ des betreffenden Objekts an, damit der Compiler die erforderliche Speichergröße kennt. Folglich reserviert `new unsigned short int` 2 Bytes auf dem Heap und `new long` 4 Bytes.

Der Rückgabewert von `new` ist eine Speicheradresse. Diese muss man einem Zeiger zuweisen. Um einen `unsigned short` auf dem Heap zu erzeugen, schreibt man z.B.:

```
unsigned short int * pPointer;
pPointer = new unsigned short int;
```

Hierbei kann man gleichzeitig auch den Zeiger initialisieren:

```
unsigned short int * pPointer = new unsigned short int;
```

In beiden Fällen zeigt nun `pPointer` auf einen `unsigned short int` im Heap. Man kann diesen Zeiger wie jeden anderen Zeiger auf eine Variable verwenden und einen Wert in diesem Bereich des Speichers ablegen. Dazu ist beispielsweise folgende Anweisung zu formulieren:

```
*pPointer = 72;
```

Das bedeutet: »Lege 72 am Wert in `pPointer` ab« oder »Weise den Wert 72 an den Bereich im Heap zu, auf den `pPointer` zeigt«.

Wenn `new` keinen Speicher auf dem Heap erzeugen kann – Speicher ist immerhin eine begrenzte Ressource –, wird eine Ausnahme (Exception) ausgelöst.

Auf Ausnahmen und Objekte zur Fehlerbehandlung geht Kapitel 24 ausführlich ein.

Ältere Compiler geben den Null-Zeiger zurück. In diesen Fällen muss man bei jeder Anforderung von neuem Speicher auf einen Null-Zeiger prüfen. Dagegen kann man sich bei allen modernen Compilern darauf verlassen, dass eine Ausnahme ausgelöst wird.

9.3.2 Das Schlüsselwort delete

Wenn man den Speicherbereich nicht mehr benötigt, ruft man `delete` auf dem Zeiger auf. `delete` gibt den Speicher an den Heap zurück. Denken Sie daran, dass der Zeiger selbst – im Gegensatz zum Speicher, auf den er zeigt – eine lokale Variable ist. Wenn die Funktion, in der er deklariert ist, zurückkehrt, verliert der Zeiger seinen Gültigkeitsbereich und ist nicht mehr zugänglich. Der mit dem Operator `new` reservierte Speicher wird allerdings nicht automatisch freigegeben. Auf den Speicher kann man nicht mehr zugreifen – es entsteht eine so genannte *Speicherlücke*. Diese Lücke lässt sich bis zum Ende des Programms nicht mehr beseitigen. Es stellt sich so dar, als ob der Computer leck wäre und Speicher verliert.

Mit dem Schlüsselwort `delete` wird Speicher an den Heap wie im folgenden Beispiel zurückgegeben:

```
delete pPointer;
```

Wenn man den Zeiger löscht, gibt man praktisch den Speicher frei, dessen Adresse im Zeiger abgelegt ist. Man sagt also »gib an den Heap den Speicher zurück, auf den der Zeiger verweist«. Der Zeiger bleibt weiterhin ein Zeiger und kann erneut zugewiesen werden. Listing 9.4 zeigt, wie man eine Variable auf dem Heap reserviert, diese Variable verwendet und sie dann löscht.

Wenn man `delete` auf einem Zeiger aufruft, wird der Speicher, auf den der Zeiger verweist, freigegeben. Ruft man `delete` erneut auf diesem Zeiger auf, stürzt das Programm ab! Einen Zeiger sollte man beim Löschen auf `0` (Null) setzen. Der Aufruf von `delete` auf einem Null-Zeiger ist garantiert sicher. Dazu folgendes Beispiel:

```
Animal *pDog = new Animal;
delete pDog;     // gibt den Speicher frei
pDog = 0;        // setzt den Zeiger auf Null
//...
delete pDog;     // harmlos
```

Lassen Sie sich vom obigen Code nicht verwirren. Im nächsten Kapitel behandeln wir die Reservierung von Objekten auf dem Heap. Das funktioniert auch mit atomaren Datentypen wie zum Beispiel `int`:

```
int *pNumber = new int;
delete pNumber;  // gibt den Speicher frei
pNumber = 0;     // setzt den Zeiger auf Null
//...
delete pNumber;  // harmlos
```

Listing 9.4: pointer-heap.cpp – Zuweisen und Löschen eines Zeigers

```
 1: // Listing 9.4
 2: // Zuweisen und Löschen eines Zeigers
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     int localVariable = 5;
 8:     int * pLocal= &localVariable;
 9:     int * pHeap = new int;
10:     if (pHeap == NULL)
11:     {
12:         std::cout << "Fehler! Kein Speicher für pHeap!!";
13:         return 1;
14:     }
15:     *pHeap = 7;
16:     std::cout << "localVariable: " << localVariable << "\n";
17:     std::cout << "*pLocal: " << *pLocal << "\n";
18:     std::cout << "*pHeap: " << *pHeap << "\n";
19:     delete pHeap;
20:     pHeap = new int;
21:     if (pHeap == NULL)
22:     {
23:         std::cout << "Fehler! Kein Speicher für pHeap!!";
24:         return 1;
25:     }
```

```
26:     *pHeap = 9;
27:     std::cout << "*pHeap: " << *pHeap << "\n";
28:     delete pHeap;
29:     return 0;
30: }
```

```
localVariable: 5
*pLocal: 5
*pHeap: 7
*pHeap: 9
```

Zeile 7 deklariert und initialisiert eine lokale Variable. Zeile 8 deklariert und initialisiert einen Zeiger mit der Adresse der lokalen Variablen. Zeile 9 deklariert einen weiteren Zeiger, initialisiert ihn aber mit dem Ergebnis aus dem Aufruf von `new int`. Damit reserviert man Speicher im Heap für eine ganze Zahl (`int`).

Zeile 15 weist den Wert `7` an die neu reservierte Speicherstelle zu. Zeile 16 gibt den Wert der lokalen Variablen aus, und Zeile 17 zeigt den Wert an, auf den `pLocal` verweist. Wie erwartet sind die Ergebnisse gleich. Zeile 18 gibt den Wert aus, auf den `pHeap` zeigt. Der in Zeile 14 zugewiesene Wert ist also tatsächlich zugänglich.

Der Aufruf von `delete` in Zeile 19 gibt den in Zeile 9 reservierten Speicher an den Heap zurück. Diese Aktion gibt den Speicher frei und hebt die Zuordnung des Zeigers zu diesem Speicherbereich auf. `pHeap` steht nun wieder als Zeiger für einen anderen Speicherbereich zur Verfügung. In den Zeilen 20 bis 26 findet eine erneute Zuweisung statt und Zeile 27 bringt das Ergebnis auf den Bildschirm. Zeile 28 gibt den Speicher an den Heap zurück.

Obwohl Zeile 28 redundant ist (durch das Ende des Programms wird der Speicher ohnehin zurückgegeben), empfiehlt es sich immer, diesen Speicher explizit freizugeben. Wenn man das Programm ändert oder erweitert, ist es vorteilhaft, dass dieser Schritt bereits berücksichtigt wurde.

9.3.3 Speicherlücken

Unbeabsichtigte Speicherlücken entstehen auch, wenn man einen Zeiger neu zuweist, ohne vorher den vom Zeiger referenzierten Speicher freizugeben. Sehen Sie sich das folgende Codefragment an:

```
1:   unsigned short int * pPointer = new unsigned short int;
2:   *pPointer = 72;
3:   pPointer = new unsigned short int;
4:   *pPointer = 84;
```

Zeile 1 erzeugt `pPointer` und weist ihm die Adresse eines Bereichs auf dem Heap zu. Zeile 2 speichert den Wert `72` in diesem Speicherbereich. Zeile 3 nimmt eine erneute Zuweisung von `pPointer` auf einen anderen Speicherbereich vor. Zeile 4 platziert den Wert `84` in diesem Bereich. Der ursprüngliche Bereich, in dem sich jetzt der Wert `72` befindet, ist nun nicht mehr zugänglich, da der Zeiger auf diesen Speicherbereich neu zugewiesen wurde. Es gibt keine Möglichkeit, auf diesen Originalspeicherbereich zuzugreifen, und man kann auch diesen Speicher nicht freigeben, bevor das Programm endet.

Der Code sollte wie folgt formuliert werden:

```
1:   unsigned short int * pPointer = new unsigned short int;
2:   *pPointer = 72;
3:   delete pPointer;
4:   pPointer = new unsigned short int;
5:   *pPointer = 84;
```

In Zeile 3 löscht man den Speicher, auf den `pPointer` ursprünglich gezeigt hat, und gibt damit diesen Bereich frei.

Zur Lücke bekennen

Für jeden Aufruf von `new` sollte auch ein korrespondierender Aufruf von `delete` vorhanden sein. An jedem Punkt im Programm muss klar sein, welcher Zeiger einen Speicherbereich besitzt. Außerdem ist sicherzustellen, dass nicht mehr benötigter Speicher umgehend an den Heap zurückgegeben wird.

Die Funktionen malloc() und free()

In manchen älteren Programmen finden Sie Aufrufe der Funktionen `malloc()`und `free()`. Diese zum Sprachumfang von C gehörenden Funktionen reservieren Speicher auf dem Heap und geben ihn fast in der gleichen Weise wie `new` und `delete` frei. Die wesentlichen Unterschiede zwischen den Versionen von C und C++ bestehen darin, dass man mit `malloc` explizit angeben muss, wie viel Speicher zu reservieren ist. Weitere Informationen zu diesen Funktionen finden Sie in der Hilfe oder im Handbuch Ihres Compilers. In diesem Buch verwenden wir die neueren Funktionen von C++.

9.4 Fragen und Antworten

F: Warum sind Zeiger so wichtig?

A: Heute haben Sie erfahren, wie Zeiger die Adresse von Objekten im Heap aufnehmen und wie man sie bei der Übergabe von Argumenten als Referenz verwendet. In Kapitel 13 lernen Sie, wie man Zeiger bei polymorphen Klassen einsetzt.

F: Warum soll ich mir die Mühe machen, Objekte auf dem Heap zu deklarieren?

A: Objekte auf dem Heap bleiben auch nach dem Rücksprung aus einer Funktion bestehen. Darüber hinaus kann man zur Laufzeit entscheiden, wie viele Objekte man benötigt, und muss die Objekte nicht im Voraus deklarieren. Auf dieses Thema geht Stunde 10 ein.

9.5 Workshop

Nachdem Sie sich mit Zeigern beschäftigt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zum Compiler zu festigen.

9.5.1 Kontrollfragen

1. Worin besteht der Unterschied zwischen `0` und `NULL`, wenn man einen Zeiger initialisiert?
2. Wie viel Speicher belegt ein Zeiger auf eine Ganzzahl? Wie viel Speicher belegt ein Speicher auf eine Gleitpunktzahl?
3. Was versteht man unter einer Speicherlücke?
4. Wie gebe ich den mit `new` reservierten Speicher wieder frei?

9.5.2 Übungen

1. Modifizieren Sie `pointeruse.cpp` (Listing 9.2), um `pAge` mit `0` anstelle von `NULL` zu initialisieren. Ändern sich die Ergebnisse?
2. Modifizieren Sie `pointerstore.cpp` (Listing 9.3), um `yourAge` und `*pAge` zu multiplizieren und das Ergebnis in einer neuen Variablen zu speichern. Geben Sie diese Variable aus. Machen Sie sich klar, wie der Compiler den Unterschied zwischen den Sternchensymbolen (*) für Multiplikation und Dereferenzierung von `pAge` erkennen kann.

3. Modifizieren Sie `pointerstore.cpp` (Listing 9.3) weiter, um den Inhalt von `myAge` oder `yourAge` mithilfe des dereferenzierten Zeigers `*pAge` zu ändern.

9.5.3 Antworten zu den Kontrollfragen

1. Beide initialisieren die Adresse mit Null. Während aber `NULL` von vornherein als Zeiger erkennbar ist, deutet `0` eher auf eine Ganzzahl hin. Dennoch lassen sich beide Formen gleichberechtigt verwenden und es ist eine Frage des persönlichen Stils, welche Form man wählt.
2. In den meisten Systemen ist der Speicherbedarf für beide Zeiger gleich. Als Programmierer brauchen Sie sich aber gar nicht um die Größe eines Zeigers zu kümmern – er ist ausreichend groß, um eine Adresse auf dem jeweiligen Computer aufzunehmen!
3. Eine Speicherlücke entsteht, wenn man Platz auf dem Heap reserviert und ihn am Ende nicht freigibt. Das Programm reserviert weiterhin nach Bedarf neuen Speicher, es ist aber immer weniger Speicher verfügbar.
4. Verwenden Sie das Schlüsselwort `delete`. Es gehört zum guten Programmierstil, den Speicher mit `delete` unverzüglich freizugeben, sobald man eine Variable auf dem Heap nicht mehr benötigt.

KAPITEL 10

jetzt lerne ich

Zeiger – weiterführende Themen

In dieser Stunde lernen Sie,

- wie man Objekte auf dem Heap erzeugt,
- wie man Zeiger effektiv einsetzt,
- wie man Speicherprobleme bei Verwendung von Zeigern vermeidet.

10.1 Objekte auf dem Heap erzeugen

Mit Zeigern (Pointern) lässt sich der Computer-Speicher direkt manipulieren. Damit gehören Zeiger zu den leistungsfähigsten Werkzeugen in der Hand des C++-Programmierers.

So wie man einen Zeiger auf eine Ganzzahl erzeugen kann, lässt sich auch ein Zeiger auf ein beliebiges Objekt erzeugen. Wenn man ein Objekt vom Typ `Cat` deklariert hat, kann man einen Zeiger auf diese Klasse deklarieren und ein `Cat`-Objekt auf dem Heap instanzieren, wie es auch auf dem Stack möglich ist. Die Syntax entspricht der für Integer-Objekte:

```
Cat *pCat = new Cat;
```

Damit wird der Standardkonstruktor aufgerufen – der Konstruktor, der keine Parameter übernimmt. Der Aufruf des Konstruktors findet immer statt, wenn man ein Objekt – auf dem Stack oder auf dem Heap – erzeugt.

10.2 Objekte löschen

Beim Aufruf von `delete` auf einem Zeiger zu einem Objekt auf dem Heap wird der Destruktor dieses Objekts aufgerufen, bevor die Freigabe des Speichers erfolgt. Damit kann die Klasse Aufräumungsarbeiten erledigen, genau wie bei Objekten, die auf dem Stack abgebaut werden. Listing 10.1 demonstriert das Erzeugen und Löschen von Objekten auf dem Heap.

Listing 10.1: heapcreate.cpp – Objekte auf dem Heap erzeugen und löschen

```
// Listing 10.1
// Objekte auf dem Heap erzeugen
#include <iostream>

class SimpleCat
{
public:
    SimpleCat();
    ~SimpleCat();
private:
    int itsAge;
};

SimpleCat::SimpleCat()
{
    std::cout << "Konstruktor aufgerufen." << std::endl;
    itsAge = 1;
}

SimpleCat::~SimpleCat()
{
    std::cout << "Destruktor aufgerufen." << std::endl;
}

int main()
{
    std::cout << "SimpleCat Frisky..." << std::endl;
    SimpleCat Frisky;

    std::cout << "SimpleCat *pRags = new SimpleCat..." <<
std::endl;
    SimpleCat * pRags = new SimpleCat;

    std::cout << "Loeschen pRags..." << std::endl;
    delete pRags;

    std::cout << "Beenden, Frisky geht..." << std::endl;
    return 0;
}
```

```
SimpleCat Frisky...
Konstruktor aufgerufen.
SimpleCat * pRags = new SimpleCat...
Konstruktor aufgerufen.
Loeschen pRags...
Destruktor aufgerufen.
Beenden, Frisky geht...
Destruktor aufgerufen.
```

Die Zeilen 5 bis 12 deklarieren die abgespeckte Klasse `SimpleCat`. Zeile 28 erzeugt das Objekt `Frisky` auf dem Stack und bewirkt damit den Aufruf des Konstruktors. Zeile 31 erzeugt auf dem Heap ein `SimpleCat`-Objekt, auf das `pRags` zeigt. Der Konstruktor wird erneut aufgerufen. Zeile 34 ruft `delete` auf `pRags` auf und löst damit den Aufruf des Destruktors aus. Am Ende der Funktion verliert `Frisky` den Gültigkeitsbereich und der Destruktor wird aufgerufen.

10.3 Mit Zeigern auf Datenelemente zugreifen

Auf Datenelemente und Funktionen greift man bei lokal erzeugten `Cat`-Objekten mit dem Punktoperator (`.`) zu. Um das `Cat`-Objekt im Heap anzusprechen, muss man den Zeiger dereferenzieren und den Punktoperator auf dem Objekt aufrufen, auf das der Zeiger verweist. Für einen Zugriff auf die Elementfunktion `GetAge()` schreibt man daher

```
(*pRags).GetAge();
```

Mit den Klammern stellt man sicher, dass `pRags` vor dem Zugriff auf `GetAge()` dereferenziert wird.

Da diese Methode umständlich ist, bietet C++ einen Kurzoperator für den indirekten Zugriff: den *Elementverweis-Operator* (`->`), der aus einem Bindestrich und einem unmittelbar folgenden Größer-Symbol besteht. C++ behandelt diese Zeichenfolge als ein einzelnes Symbol. Listing 10.2 demonstriert den Zugriff auf Elementvariablen und Funktionen von Objekten, die auf dem Heap erzeugt wurden.

Listing 10.2: heapaccess.cpp – Zugriff auf Datenelemente von Objekten auf dem Heap

```
 1: // Listing 10.2
 2: // Zugriff auf Datenelemente von Objekten auf dem Heap
 3: #include <iostream>
 4: using std::endl;
 5: class SimpleCat
 6: {
 7: public:
 8:     SimpleCat() {itsAge = 2; }
 9:     ~SimpleCat() {}
10:     int GetAge() const { return itsAge; }
11:     void SetAge(int age) { itsAge = age; }
12: private:
13:     int itsAge;
14: };
15:
16: int main()
17: {
18:     SimpleCat * Frisky = new SimpleCat;
19:     std::cout << "Frisky ist " << Frisky->GetAge()
20:               << " Jahre alt." << endl;
21:
22:     Frisky->SetAge(5);
23:     std::cout << "Frisky ist " << Frisky->GetAge()
24:               << " Jahre alt." << endl;
25:
26:     delete Frisky;
27:     return 0;
28: }
```

```
Frisky ist 2 Jahre alt.
Frisky ist 5 Jahre alt.
```

Zeile 18 instanziert ein `SimpleCat`-Objekt auf dem Heap. Der Standardkonstruktor setzt das Alter auf `2`. Zeile 19 ruft die Methode `GetAge()` auf. Da es sich hierbei um einen Zeiger handelt, wird der Elementverweis-Operator (`->`) für den Zugriff auf Datenelemente und Funktionen verwendet. In Zeile 22 steht der Aufruf von `SetAge()` und in Zeile 23 findet ein erneuter Zugriff auf `GetAge()` statt.

10.4 Datenelemente auf dem Heap

Datenelemente einer Klasse lassen sich als Zeiger auf ein Objekt auf dem Heap deklarieren. Der Speicher kann im Konstruktor der Klasse oder einer ihrer Methoden reserviert und in ihrem Destruktor gelöscht werden, wie es Listing 10.3 verdeutlicht.

Listing 10.3: datamemberptr.cpp – Zeiger als Datenelemente

```
 1: // Listing 10.3
 2: // Zeiger als Datenelemente
 3: #include <iostream>
 4:
 5: class SimpleCat
 6: {
 7: public:
 8:     SimpleCat();
 9:     ~SimpleCat();
10:     int GetAge() const { return *itsAge; }
11:     void SetAge(int age) { *itsAge = age; }
12:
13:     int GetWeight() const { return *itsWeight; }
14:     void setWeight (int weight) { *itsWeight = weight; }
15:
16: private:
17:     int * itsAge;
18:     int * itsWeight;
19: };
20:
21: SimpleCat::SimpleCat()
22: {
23:     itsAge = new int(2);
24:     itsWeight = new int(5);
25: }
26:
27: SimpleCat::~SimpleCat()
28: {
29:     delete itsAge;
30:     delete itsWeight;
31: }
32:
33: int main()
34: {
35:     SimpleCat *Frisky = new SimpleCat;
36:     std::cout << "Frisky ist " << Frisky->GetAge()
37:               << " Jahre alt.\n";
38:
39:     Frisky->SetAge(5);
40:     std::cout << "Frisky ist " << Frisky->GetAge()
41:               << " Jahre alt.\n";
42:
43:     delete Frisky;
44:     return 0;
45: }
```

```
Frisky ist 2 Jahre alt.
Frisky ist 5 Jahre alt.
```

Die Klasse `SimpleCat` hat zwei Elementvariablen – beide sind Zeiger auf Integer. Der Konstruktor (Zeilen 21 bis 25) initialisiert die Zeiger auf einen Speicherbereich im Heap und auf die Standardwerte.

Der Destruktor (Zeilen 27 bis 31) räumt den reservierten Speicher auf. Da es sich um den Destruktor handelt, gibt es keinen Grund, diesen Zeigern `null` zuzuweisen, da sie nicht mehr zugänglich sind. Hier kann man ausnahmsweise die Regel brechen, dass gelöschte Zeiger den Wert `null` erhalten sollten, obwohl die Einhaltung dieser Regel auch nicht stören würde.

Die aufrufende Funktion – in diesem Fall `main()` – ist nicht davon unterrichtet, dass `itsAge` und `itsWeight` Zeiger auf Speicher im Heap sind. `main()` ruft weiterhin `GetAge()` und `SetAge()` auf und die Einzelheiten der Speicherverwaltung werden – wie es sich gehört – in der Implementierung der Klasse versteckt.

Beim Löschen von `Frisky` in Zeile 41 wird der Destruktor aufgerufen. Der Destruktor löscht alle zugehörigen Elementzeiger. Wenn diese ihrerseits auf Objekte anderer benutzerdefinierter Klassen zeigen, werden deren Destruktoren ebenfalls aufgerufen.

Das ist ein gutes Beispiel, warum man einen eigenen Destruktor schreibt und sich nicht auf den Standardkonstruktor des Compilers verlässt. Normalerweise finden die Löschoperationen in den Zeilen 29 und 30 nicht statt; wir mussten sie explizit kodieren. Ohne diese Anweisungen wird das Objekt in Zeile 43 gelöscht (einschließlich der Zeiger auf den Heap) – ohne dass die Einträge auf dem Heap selbst gelöscht werden. Ohne Destruktor würde also eine Speicherlücke entstehen.

10.5 Der Zeiger this

Jede Elementfunktion einer Klasse verfügt über einen versteckten Parameter: den Zeiger `this`, der auf das eigene Objekt zeigt. Deswegen ist bei jedem Aufruf von `GetAge()` oder `SetAge()` der Zeiger `this` für das Objekt als versteckter Parameter mit von der Partie.

Die Aufgabe des Zeigers `this` besteht darin, auf das jeweilige Objekt zu verweisen, dessen Methode aufgerufen wurde. Normalerweise ist das nicht erforderlich. Man ruft einfach Methoden auf und setzt Elementvariablen. Gelegentlich benötigt man aber den Zugriff auf das Objekt selbst (etwa um einen Zeiger auf das aktuelle Objekt zurückzugeben). Hier erweist sich der Zeiger `this` als hilfreich.

Normalerweise braucht man den Zeiger `this` nicht, um auf Elementvariablen eines Objekts über die Methoden dieses Objekts zuzugreifen. Allerdings kann man ohne weiteres den Zeiger `this` explizit aufrufen. Diese Vorgehensweise zeigt Listing 10.4, das die Existenz und Arbeitsweise des Zeigers `this` demonstriert.

Listing 10.4: usingthis.cpp – Den Zeiger this verwenden

```
// Listing 10.4
// Verwendung des Zeigers this
#include <iostream>
using namespace std;
class Rectangle
{
   public:
      Rectangle();
      ~Rectangle();
      void SetLength(int length) { this->itsLength = length; }
      int GetLength() const { return this->itsLength; }
      void SetWidth(int width) { itsWidth = width; }
      int GetWidth() const { return itsWidth; }

   private:
      int itsLength;
      int itsWidth;
};

Rectangle::Rectangle()
{
   itsWidth = 5;
   itsLength = 10;
}

Rectangle::~Rectangle()
{}

int main()
{
   Rectangle theRect;
   cout << "theRect ist " << theRect.GetLength()
        << " Meter lang." << endl;
   cout << "theRect ist " << theRect.GetWidth()
        << " Meter breit." << endl;

   theRect.SetLength(20);
   theRect.SetWidth(10);
   cout << "theRect ist " << theRect.GetLength()
        << " Meter lang." << endl;
   cout << "theRect ist " << theRect.GetWidth()
        << " Meter breit." << endl;
```

```
43:
44:    return 0;
45: }
```

```
theRect ist 10 Meter lang.
theRect ist 5 Meter breit.
theRect ist 20 Meter lang.
theRect ist 10 Meter breit.
```

Die Zugriffsfunktionen `SetLength()` und `GetLength()` verwenden explizit den Zeiger `this`, um auf die Elementvariablen des Objekts `Rectangle` zuzugreifen. Dagegen arbeiten die Zugriffsfunktionen `SetWidth()` und `GetWidth()` ohne diesen Zeiger. Im Verhalten ist kein Unterschied festzustellen, obwohl das Verfahren ohne den Zeiger `this` leichter zu verstehen ist.

Wofür ist der Zeiger this gut?

Wäre das alles, wofür der Zeiger `this` gut ist, hätte man kaum Anlass, sich mit diesem Zeiger zu beschäftigen. Gerade aber weil `this` ein Zeiger ist, kann er die Speicheradresse eines Objekts aufnehmen. In dieser Eigenschaft ist der Zeiger ein leistungsfähiges Werkzeug.

Die praktische Verwendung des Zeigers `this` lernen Sie später in diesem Buch kennen, wenn es um das Überladen von Operatoren geht. Momentan genügt es zu wissen, dass es den Zeiger `this` gibt und dass er ein Zeiger auf das Objekt selbst ist.

Um das Erzeugen oder Löschen des Zeigers `this` braucht man sich nicht zu kümmern. Das erledigt der Compiler.

10.6 Wilde Zeiger

Zu den schwer zu lokalisierenden Fehlerquellen gehören *wilde Zeiger*. Derartige Zeiger entstehen, wenn man `delete` auf einem Zeiger aufruft – dabei den Speicher freigibt, auf den der Zeiger verweist – und später versucht, diesen Zeiger erneut ohne Neuzuweisung zu verwenden.

Es verhält sich genauso, als ob die Firma Acme Mail Order umgezogen wäre und man weiterhin dieselbe Kurzwahltaste auf dem Telefon betätigt. Vielleicht passiert nichts Furchtbares – ein Telefon klingelt in einem verwaisten Warenhaus. Die Telefonnummer könnte aber auch jemandem zugeteilt worden sein, der gerade aus der Nachtschicht gekommen ist – und nun haben Sie ihn aufgeweckt!

Man sollte also keinen Zeiger verwenden, nachdem man darauf `delete` aufgerufen hat. Der Zeiger verweist immer noch auf den alten Speicherbereich, aber der Compiler kann dort ohne weiteres andere Daten unterbringen. Die Verwendung des Zeigers kann zum Absturz des Programms führen. Der schlimmere Fall ist, dass das Programm noch unbekümmert läuft und erst einige Minuten später abstürzt – eine so genannte *Zeitbombe*. Setzen Sie daher den Zeiger sicherheitshalber auf `NULL` (0), nachdem Sie ihn gelöscht haben. Damit ist der Zeiger »entwaffnet«.

10.7 Konstante Zeiger

Bei Zeigern kann man das Schlüsselwort `const` vor oder nach dem Typ (oder an beiden Stellen) verwenden. Die folgenden Beispiele zeigen gültige Deklarationen:

```
const int * pOne;
int * const pTwo;
const int * const pThree;
```

Diese Anweisungen bedeuten nicht das Gleiche. `pOne` ist ein *Zeiger auf eine konstante* Ganzzahl. Der Wert, auf den er zeigt, lässt sich nicht über diesen Zeiger ändern. Man kann also nicht schreiben

```
*pOne = 5
```

Versucht man es, reagiert der Compiler mit einem Fehler.

`pTwo` ist ein *konstanter Zeiger* auf eine Ganzzahl. Die Ganzzahl kann man ändern, aber `pTwo` kann nicht auf etwas anderes zeigen. Ein konstanter Zeiger lässt sich nicht erneut zuweisen. Eine Anweisung wie

```
pTwo = &x
```

verbietet sich also.

`pThree` ist ein *konstanter Zeiger auf eine konstante Ganzzahl*. Hier lässt sich weder der Wert ändern, auf den der Zeiger verweist, noch kann man `pThree` durch eine erneute Zuweisung auf etwas anderes zeigen lassen.

Ziehen Sie eine gedachte Linie unmittelbar rechts vom Sternchen. Wenn das Schlüsselwort `const` links der Linie vorkommt, bedeutet das ein konstantes Objekt. Taucht das Schlüsselwort `const` rechts der Linie auf, ist der Zeiger selbst konstant.

```
const int * p1;  // der int-Typ, auf den verwiesen wird, ist konstant
int * const p2;  // p2 ist konstant und kann nicht auf irgendetwas
anderes zeigen
```

10.7.1 Konstante Zeiger und Elementfunktionen

In Stunde 7 haben Sie gelernt, dass man das Schlüsselwort `const` auf eine Elementfunktion anwenden kann. Bei einer als `const` deklarierten Funktion bringt der Compiler einen Fehler, wenn man Daten im Objekt aus dieser Funktion heraus verändern möchte.

Wenn man einen Zeiger auf ein konstantes Objekt deklariert, lassen sich mit diesem Zeiger einzig und allein konstante Methoden aufrufen. Listing 10.5 verdeutlicht diesen Sachverhalt.

Listing 10.5: constptr.cpp – Zeiger auf const-Objekte

```
// Listing 10.5
// Zeiger bei konstanten Methoden
#include <iostream>

class Rectangle
{
public:
    Rectangle();
    ~Rectangle();
    void SetLength(int length) { itsLength = length; }
    int GetLength() const { return itsLength; }

    void SetWidth(int width) { itsWidth = width; }
    int GetWidth() const { return itsWidth; }

private:
    int itsLength;
    int itsWidth;
};

Rectangle::Rectangle():
itsWidth(5),
itsLength(10)
{}

Rectangle::~Rectangle()
{}

int main()
{
    Rectangle* pRect =  new Rectangle;
    const Rectangle * pConstRect = new Rectangle;
    Rectangle * const pConstPtr = new Rectangle;

    std::cout << "pRect Breite:      "
              << pRect->GetWidth() << " Meter" << std::endl;
```

```
37:     std::cout << "pConstRect Breite: "
38:               << pConstRect->GetWidth() << " Meter" << std::endl;
39:     std::cout << "pConstPtr Breite:  "
40:               << pConstPtr->GetWidth() << " Meter" << std::endl;
41:
42:     pRect->SetWidth(10);
43:     // pConstRect->SetWidth(10);
44:     pConstPtr->SetWidth(10);
45:
46:     std::cout << "pRect Breite:      "
47:               << pRect->GetWidth() << " Meter" << std::endl;
48:     std::cout << "pConstRect Breite: "
49:               << pConstRect->GetWidth() << " Meter" << std::endl;
50:     std::cout << "pConstPtr Breite:   "
51:               << pConstPtr->GetWidth() << " Meter" << std::endl;
52:     return 0;
53: }
```

```
pRect Breite:      5 Meter
pConstRect Breite: 5 Meter
pConstPtr Breite:  5 Meter
pRect Breite:      10 Meter
pConstRect Breite: 5 Meter
pConstPtr Breite:  10 Meter
```

In den Zeilen 5 bis 19 steht die Deklaration der Klasse `Rectangle`. Zeile 14 deklariert die Elementmethode `GetWidth` als `const`. Zeile 31 deklariert einen Zeiger auf `Rectangle`. Zeile 32 deklariert `pConstRect` als Zeiger auf ein konstantes `Rectangle`-Objekt und Zeile 33 `pConstPtr` als konstanten Zeiger auf `Rectangle`.

Die Zeilen 35 bis 40 geben die Werte der Breiten aus.

Zeile 42 setzt die Breite des Rechtecks über `pRect` auf den Wert `10`. In Zeile 43 käme eigentlich `pConstRect` zum Einsatz. Die Deklaration dieses Zeigers bezieht sich aber auf ein konstantes `Rectangle`-Objekt. Deshalb ist der Aufruf einer nicht als `const` deklarierten Elementfunktion unzulässig und die Zeile wurde auskommentiert. Zeile 33 deklariert `pConstPtr` als konstanten Zeiger auf ein Rechteck. Der Zeiger ist also konstant und kann nicht auf irgendetwas anderes zeigen, aber das Rechteck ist nicht konstant.

10.7.2 Konstante this-Zeiger

Wenn man ein Objekt als `const` deklariert, hat man praktisch den Zeiger `this` als Zeiger auf ein konstantes Objekt deklariert. Einen `const this`-Zeiger kann man nur auf konstante Elementfunktionen anwenden.

Konstante Objekte und konstante Zeiger behandeln wir noch einmal in der nächsten Stunde, wenn es um Referenzen auf konstante Objekte geht.

10.8 Fragen und Antworten

F: Warum soll ich ein Objekt als `const` deklarieren, obwohl das die Verwendungsmöglichkeiten einschränkt?

A: Als Programmierer möchte man den Compiler auch zur Fehlersuche heranziehen. So sind beispielsweise solche Fehler schwer zu finden, die darauf beruhen, dass eine aufgerufene Funktion ein Objekt ändert, diese Änderung aber mit der aufrufenden Funktion nicht im ursächlichen Zusammenhang steht. Durch die Deklaration eines Objekts als `const` verhindert man derartige Änderungen.

F: Warum soll ich mir die Mühe machen, Objekte auf dem Heap zu deklarieren?

A: Objekte auf dem Heap bleiben auch nach Rückkehr einer Funktion bestehen. Außerdem lassen sich mit Objekten, die man auf dem Heap deklariert, komplexe Datenstrukturen erstellen, wie es Stunde 19 erläutert.

10.9 Workshop

Nachdem Sie sich in dieser Stunde tiefgründiger mit Zeigern beschäftigt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu festigen.

10.9.1 Kontrollfragen

1. Wie lauten die C++-Schlüsselwörter, um Speicher auf dem Heap zu reservieren und wieder freizugeben?
2. Wann wird ein Objekt gelöscht (der Destruktor aufgerufen), wenn Sie es nicht selbst explizit löschen?
3. Nach welchen beiden Verfahren kann man auf Datenelemente mithilfe von Zeigern zugreifen? Welches ist besser?
4. Was ist ein wilder Zeiger?

10.9.2 Übungen

1. Entfernen Sie die Kommentarzeichen in Zeile 43 von Listing 10.5 (`constptr.cpp`). Was passiert? Was sagt Ihnen der Compiler?
2. Vielleicht haben Sie bemerkt, dass einige Listings `\n` und andere `std::endl` verwenden. In manchen kommt `std::cout` und `std::endl` vor, während andere Listings die `using`-Anweisung zusammen mit `cout` und `endl` nutzen. Probieren Sie in einigen der bisher vorgestellten Listings die verschiedenen Varianten aus. Welche lässt sich besser verwenden: `\n` oder `endl`? Welche ist einfacher: die mit dem Präfix `std::` oder die mit der `using`-Anweisung?
3. Modifizieren Sie `heapaccess.cpp` (Listing 10.2), sodass der Code ohne Elementverweis-Operator auskommt. Welche Version ist verständlicher, das Original oder das modifizierte Listing?

10.9.3 Antworten zu den Kontrollfragen

1. Mit `new` reserviert man Speicher auf dem Heap und mit `delete` gibt man den Speicher wieder frei.
2. Wenn ein Objekt nicht mehr gültig ist, wird es automatisch gelöscht. Hat der Programmierer ein Objekt in `main()` erzeugt und nicht gelöscht, wird der Destruktor aufgerufen, wenn `main()` terminiert. Die Ausgabe von Listing 10.1 zeigt, wie das geschieht.
3. Das eine Verfahren verwendet `(*pRags).GetAge()`, das andere `pRags->GetAge()`. Die Version mit dem Elementverweis-Operator (`->`) ist besser, weil sie bereits auf den ersten Blick die Absicht erkennen lässt.
4. Ein wilder Zeiger tritt auf, wenn man auf Speicher zugreift, auf den ein Zeiger verweist, nachdem man diesen Zeiger gelöscht hat. Es ist völlig unbekannt, wofür diese Speicherstelle jetzt verwendet wird!

KAPITEL 11

jetzt lerne ich

Referenzen

In dieser Stunde lernen Sie,

- was Referenzen sind,
- wie sich Referenzen von Zeigern unterscheiden,
- wie man Referenzen erzeugt und verwendet,
- welche Beschränkungen bei Referenzen vorhanden sind,
- wie man Werte und Objekte in und aus Funktionen als Referenz übergibt.

11.1 Was ist eine Referenz?

In den beiden letzten Stunden haben Sie gelernt, wie man Zeiger verwendet, Objekte im Heap manipuliert und auf diese Objekte indirekt verweist. Referenzen bieten nahezu die gleiche Leistung wie Zeiger, aber mit einer wesentlich einfacheren Syntax.

Eine *Referenz* ist ein Aliasname. Wenn man eine Referenz erzeugt, initialisiert man sie mit dem Namen eines anderen Objekts, dem *Ziel*. Von diesem Moment an wirkt die Referenz als alternativer Name für das Ziel und alles, was man mit der Referenz anstellt, bezieht sich tatsächlich auf das Ziel.

Das war's. Einige C++-Programmierer behaupten, dass Referenzen Zeiger sind. Das ist aber nicht korrekt. Auch wenn Referenzen oftmals mithilfe von Zeigern implementiert werden, ist das nur von Belang für die Compiler-Bauer. Als Programmierer muss man beide Konzepte – Referenzen und Zeiger – auseinanderhalten.

Zeiger sind Variablen, die die Adresse eines anderen Objekts aufnehmen. Referenzen sind Aliasnamen für ein Objekt.

11.2 Referenzen erzeugen

Die Deklaration einer Referenz besteht aus dem Typ des Zielobjekts, gefolgt vom Referenzoperator (&) und dem Namen der Referenz. Die Regeln für die Benennung von Referenzen entsprechen den gültigen Variablennamen. Im Buch erhalten Referenzen durchgängig ein vorangestelltes r. Zum Beispiel erzeugt man für eine Integer-Variable `someInt` eine Referenz mit der folgenden Anweisung:

```
int &rSomeRef = someInt;
```

Man liest das als »`rSomeRef` ist eine Referenz auf einen Integer, die mit einem Verweis auf `someInt` initialisiert ist«. Listing 11.1 zeigt, wie man Referenzen erzeugt und verwendet.

Der Referenzoperator

C++ verwendet für den Referenzoperator (&) dasselbe Symbol wie für den Adressoperator. Beim Referenzoperator steht es aber in der Deklaration.

Denken Sie daran, dass ein Sternchen (*) in der Deklaration bedeutet, dass die Variable ein Zeiger ist. In einer Anweisung bezeichnet es den Indirektionsoperator bei Zeigern oder den Multiplikationsoperator bei mathematischen Ausdrücken.

Listing 11.1: createreference.cpp – Referenzen erzeugen und verwenden

```
// Listing 11.1
// Referenzen erzeugen und verwenden
#include <iostream>

int main()
{
    int  intOne;
    int &rSomeRef = intOne;

    intOne = 5;
    std::cout << "intOne: " << intOne << std::endl;
    std::cout << "rSomeRef: " << rSomeRef << std::endl;

    rSomeRef = 7;
    std::cout << "intOne: " << intOne << std::endl;
    std::cout << "rSomeRef: " << rSomeRef << std::endl;
    return 0;
}
```

```
intOne: 5
rSomeRef: 5
intOne: 7
rSomeRef: 7
```

Zeile 7 deklariert die lokale `int`-Variable `intOne`. In Zeile 8 wird eine Referenz `rSomeRef` auf eine Integer-Zahl deklariert und mit `intOne` initialisiert. Wenn man eine Referenz deklariert, aber nicht initialisiert, erhält man einen Compiler-Fehler. Referenzen müssen initialisiert werden.

Zeile 10 weist `intOne` den Wert `5` zu. Die Anweisungen in den Zeilen 11 und 12 geben die Werte von `intOne` und `rSomeRef` aus. Natürlich sind sie gleich, da `rSomeRef` lediglich die Referenz auf `intOne` ist.

In Zeile 14 steht die Zuweisung von `7` an `rSomeRef`. Da es sich um eine Referenz handelt, ist es ein Alias für `intOne`. Die Zuweisung von `7` wirkt demzufolge auf `intOne`, wie es die Ausgaben in den Zeilen 15 und 16 belegen.

11.3 Der Adressoperator bei Referenzen

Wenn man die Adresse einer Referenz abfragt, erhält man die Adresse des Ziels der Referenz. Genau das ist das Wesen der Referenzen – sie sind Aliasnamen für das Ziel. Listing 11.2 verdeutlicht diesen Sachverhalt.

Listing 11.2: addressreference.cpp – Die Adresse einer Referenz ermitteln

```
 1: // Listing 11.2
 2: // Zeigt die Verwendung von Referenzen
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:     int  intOne;
 8:     int &rSomeRef = intOne;
 9:
10:     intOne = 5;
11:     std::cout << "intOne: " << intOne << std::endl;
12:     std::cout << "rSomeRef: " << rSomeRef << std::endl;
13:
14:     std::cout << "&intOne: "  << &intOne << std::endl;
15:     std::cout << "&rSomeRef: " << &rSomeRef << std::endl;
16:
17:     return 0;
18: }
```

```
intOne: 5
rSomeRef: 5
&intOne: 1245064
&rSomeRef: 1245064
```

Die Ausgabe kann bei Ihnen anders aussehen, da jeder Computer die Variablen bei anderen Adressen speichert, je nachdem, was noch im Speicher steht und wie viel Speicher verfügbar ist. Bei einem anderen als dem Borland-Compiler kann beispielsweise folgende Ausgabe erscheinen:

```
intOne: 5
rSomeRef: 5
&intOne:  0x0012FF7C
&rSomeRef: 0x0012FF7C
```

Auch hier wird `rSomeRef` als Referenz auf `intOne` initialisiert. Die Ausgabe zeigt diesmal die Adressen der beiden Variablen – sie sind identisch. C++ bietet keine Möglichkeit, auf die Adresse der Referenz selbst zuzugreifen, da sie im Gegensatz zu einem Zeiger oder einer anderen Variablen nicht von Bedeutung ist. Referenzen werden bei ihrer Erzeugung initialisiert und agieren immer als Synonyme für ihre Ziele, selbst wenn man den Adressoperator anwendet.

Bei einer Klasse wie zum Beispiel `President` kann man eine Instanz dieser Klasse wie folgt deklarieren:

```
President  George_W_Bush;
```

Dann lässt sich eine Referenz auf `President` deklarieren und mit diesem Objekt initialisieren:

```
President &Dubya = George_W_Bush;
```

Es gibt nur ein `President`-Objekt; beide Bezeichner beziehen sich auf dasselbe Objekt derselben Klasse. Alle Aktionen, die man auf `Dubya` ausführt, werden genauso auf `George_W_Bush` unternommen.

Achten Sie auf den Unterschied zwischen dem Symbol `&` in Zeile 8 von Listing 11.2, das eine Referenz auf `int` namens `rSomeRef` deklariert, und den Symbolen `&` in den Zeilen 14 und 15, die die Adressen der Integer-Variablen `intOne` und der Referenz `rSomeRef` zurückgeben.

Bei Referenzen arbeitet man normalerweise nicht mit dem Adressoperator. Man setzt die Referenz einfach so ein, als würde man direkt mit der Zielvariablen arbeiten. Zeile 12 zeigt dazu ein Beispiel.

Selbst erfahrene C++-Programmierer, die die Regel kennen, dass Referenzen nicht erneut zugewiesen werden können und immer Aliasnamen für ihr Ziel sind, wissen manchmal nicht, was beim erneuten Zuweisen einer Referenz passiert. Was wie eine Neuzuweisung aussieht, stellt sich als Zuweisung eines neuen Wertes an das Ziel heraus. Diese Tatsache zeigt Listing 11.3.

Listing 11.3: assignreference.cpp – Neuzuweisung einer Referenz

```
// Listing 11.3
// Neuzuweisung einer Referenz
#include <iostream>
using namespace std; // In dieser Datei std::-Objekte verwenden

int main()
{
    int  intOne;
    int &rSomeRef = intOne;

    intOne = 5;
    cout << "intOne:\t" << intOne << endl;
    cout << "rSomeRef:\t" << rSomeRef << endl;
    cout << "&intOne:\t"  << &intOne << endl;
    cout << "&rSomeRef:\t" << &rSomeRef << endl;

    int intTwo = 8;
    rSomeRef = intTwo;  // nicht, was Sie denken!
    cout << "\nintOne:\t" << intOne << endl;
    cout << "intTwo:\t" << intTwo << endl;
    cout << "rSomeRef:\t" << rSomeRef << endl;
    cout << "&intOne:\t"  << &intOne << endl;
    cout << "&intTwo:\t"  << &intTwo << endl;
    cout << "&rSomeRef:\t" << &rSomeRef << endl;
    return 0;
}
```

```
intOne:         5
rSomeRef:       5
&intOne:        1245064
&rSomeRef:      1245064

intOne:         8
intTwo:         8
rSomeRef:       8
&intOne:        1245064
&intTwo:        1245056
&rSomeRef:      1245064
```

Die Ausgabe kann bei Ihnen anders aussehen, da jeder Computer die Variablen bei anderen Adressen speichert, je nachdem, was noch im Speicher steht und wie viel Speicher verfügbar ist. Bei einem anderen als dem Borland-Compiler kann beispielsweise folgende Ausgabe erscheinen:

```
intOne:         5
rSomeRef:       5
&intOne:        0x0012FF7C
&rSomeRef:      0x0012FF7C

intOne:         8
intTwo:         8
rSomeRef:       8
&intOne:        0x0012FF7C
&intTwo:        0x0012FF74
&rSomeRef:      0x0012FF7C
```

Die Zeilen 8 und 9 deklarieren auch hier wieder eine Integer-Variable und eine Referenz auf einen Integer. In Zeile 11 erhält die Integer-Variable den Wert `5` zugewiesen und die Ausgabe der Werte und ihrer Adressen erfolgt in den Zeilen 12 bis 15.

Zeile 17 erzeugt die neue Variable `intTwo` und initialisiert sie mit dem Wert `8`. In Zeile 18 versucht der Programmierer, `rSomeRef` erneut als Alias für die Variable `intTwo` zuzuweisen. Allerdings passiert etwas anderes: `rSomeRef` wirkt nämlich weiterhin als Alias für `intOne`, sodass diese Zuweisung mit der folgenden gleichbedeutend ist:

```
intOne = intTwo;
```

Tatsächlich sind die in den Zeilen 19 bis 21 ausgegebenen Werte von `intOne` und `rSomeRef` gleich `intTwo`. Die Ausgabe der Adressen in den Zeilen 22 bis 24 beweist, dass sich `rSomeRef` weiterhin auf `intOne` und nicht auf `intTwo` bezieht.

Was Sie tun sollten

Verwenden Sie Referenzen, um einen Alias auf ein Objekt zu erzeugen.

Initialisieren Sie alle Referenzen.

Achtung

Versuchen Sie nicht, eine Referenz erneut zuzuweisen.

Verwechseln Sie den Adressoperator nicht mit dem Referenzoperator.

11.4 Was kann man referenzieren?

Alle Objekte, einschließlich der benutzerdefinierten Objekte, lassen sich referenzieren. Beachten Sie, dass man eine Referenz auf ein Objekt und nicht auf eine Klasse oder einen Datentyp wie `int` erzeugt. Beispielsweise schreibt man nicht:

```
int & rIntRef = int;    // falsch
```

Man muss `rIntRef` auf eine bestimmte Integer-Zahl initialisieren, etwa so:

```
int howBig = 200;
int & rIntRef = howBig;
```

Ebenso initialisiert man eine Referenz nicht auf eine Klasse `CAT`:

```
CAT & rCatRef = CAT;   // falsch
```

Es ist `rCatRef` auf ein bestimmtes `CAT`-Objekt zu initialisieren:

```
CAT Frisky;
CAT & rCatRef = Frisky;
```

Referenzen auf Objekte verwendet man genau wie das Objekt selbst. Auf Datenelemente und Methoden greift man mit dem normalen Zugriffsoperator (`.`) der Klasse zu. Und genau wie bei den vordefinierten Typen wirkt die Referenz als Alias für das Objekt.

11.5 Null-Zeiger und Null-Referenzen

Wenn man Zeiger löscht oder nicht initialisiert, sollte man ihnen null (`0`) zuweisen. Für Referenzen gilt das nicht. In der Tat darf eine Referenz nicht null sein, und ein Programm mit einer Referenz auf ein Null-Objekt ist unzulässig. Bei einem unzulässigen Programm kann nahezu alles passieren. Vielleicht läuft das Programm, vielleicht löscht es aber auch alle Dateien auf der Festplatte.

Die meisten Compiler unterstützen eine Null-Referenz, ohne sich darüber zu beschweren. Erst wenn man die Referenz verwendet, gibt es Ärger. Allerdings sollte man auf diese »Unterstützung« verzichten. Denn wenn man das Programm auf einer anderen Maschine oder mit einem anderen Compiler laufen lässt, können sich bei vorhandenen Null-Referenzen geheimnisvolle Fehler einschleichen.

11.6 Funktionsargumente als Referenz übergeben

In Stunde 5 haben Sie gelernt, dass Funktionen zwei Einschränkungen aufweisen: Die Übergabe von Argumenten erfolgt als Wert und die `return`-Anweisung kann nur einen einzigen Wert zurückgeben.

Die Übergabe von Werten an eine Funktion als Referenz hebt beide Einschränkungen auf. In C++ realisiert man die Übergabe als Referenz entweder mit Zeigern oder mit Referenzen. Die Syntax ist unterschiedlich, die Wirkung gleich: Anstatt eine Kopie des Objekts im Gültigkeitsbereich der Funktion zu erstellen, wird das eigentliche Originalobjekt an die Funktion übergeben. Damit kann die Funktion das betreffende Objekt verändern.

Die Funktion `swap` in Listing 11.4 vertauscht zwei Werte, wobei die Übergabe der Parameter als Wert erfolgt.

Listing 11.4: passby-value.cpp – Übergabe von Argumenten als Wert

```
 1: // Listing 11.4 Zeigt die Übergabe als Wert
 2: #include <iostream>
 3:
 4: void swap(int x, int y);
 5:
 6: int main()
 7: {
 8:     int x = 5, y = 10;
 9:
10:     std::cout << "Main. Vor Vertauschung, x: " << x
11:                                   << " y: " << y << "\n";
12:     swap(x,y);
13:     std::cout << "Main. Nach Vertauschung, x: " << x
14:                                   << " y: " << y << "\n";
15:     return 0;
16: }
17:
18: void swap (int x, int y)
19: {
20:     int temp;
21:
22:     std::cout << "Swap. Vor Vertauschung, x: " << x
23:                                   << " y: " << y << "\n";
24:
25:     temp = x;
26:     x = y;
27:     y = temp;
28:
29:     std::cout << "Swap. Nach Vertauschung, x: " << x
30:                                   << " y: " << y << "\n";
31:
32: }
```

```
Main. Vor Vertauschung, x: 5 y: 10
Swap. Vor Vertauschung, x: 5 y: 10
Swap. Nach Vertauschung, x: 10 y: 5
Main. Nach Vertauschung, x: 5 y: 10
```

Dieses Programm initialisiert zwei Variablen in `main()` und übergibt sie dann an die Funktion `swap()`, die auf den ersten Blick eine Vertauschung der Werte vornimmt. Inspiziert man aber die Werte erneut in `main()`, haben sie ihre Plätze nicht gewechselt!

Das Problem besteht hier darin, dass die Übergabe von `x` und `y` an die Funktion `swap()` als Wert erfolgt. Das heißt, die Funktion legt lokale Kopien dieser Variablen an. Man müsste also `x` und `y` als Referenz übergeben.

In C++ bieten sich hier zwei Möglichkeiten: Man kann die Parameter der Funktion `swap()` als Zeiger auf die Originalwerte ausbilden oder Referenzen auf die Originalwerte übergeben.

11.6.1 Parameter mit Zeigern übergeben

Die Übergabe eines Zeigers bedeutet, dass man die Adresse des Objekts übergibt. Daher kann die Funktion den Wert an dieser Adresse manipulieren. Damit `swap()` die eigentlichen Werte mithilfe von Zeigern vertauscht, deklariert man die Parameter der Funktion `swap()` als zwei `int`-Zeiger. Die Funktion dereferenziert diese beiden Zeiger und vertauscht damit wie beabsichtigt die Werte von `x` und `y`. Listing 11.5 verdeutlicht dieses Konzept.

Listing 11.5: passbyptr.cpp – Übergabe als Referenz mit Zeigern

```
 1: // Listing 11.5 Zeigt die Übergabe als Referenz
 2: #include <iostream>
 3:
 4: void swap(int *x, int *y);
 5: // Das "*" besagt, dass die Funktion Zeiger erwartet.
 6: int main()
 7: {
 8:     int x = 5, y = 10;
 9:
10:     std::cout << "Main. Vor Vertauschung, x: " << x
11:                                 << " y: " << y << "\n";
12:     swap(&x,&y);
13:     std::cout << "Main. Nach Vertauschung, x: " << x
14:                                 << " y: " << y << "\n";
15:     return 0;
16: }
17:
18: void swap (int *px, int *py)
19: {
```

```
20:      int temp;
21:
22:      std::cout << "Swap. Vor Vertauschung, *px: " << *px
23:                                  << " *py: " << *py << "\n";
24:
25:      temp = *px;
26:      *px = *py;
27:      *py = temp;
28:
29:      std::cout << "Swap. Nach Vertauschung, *px: " << *px
30:                                  << " *py: " << *py << "\n";
31: }
```

```
Main. Vor Vertauschung, x: 5 y: 10
Swap. Vor Vertauschung, *px: 5 *py: 10
Swap. Nach Vertauschung, *px: 10 *py: 5
Main. Nach Vertauschung, x: 10 y: 5
```

Bingo! Der geänderte Prototyp von `swap()` in Zeile 4 zeigt nun an, dass die beiden Parameter als Zeiger auf `int` und nicht als `int`-Variablen spezifiziert sind. Der Aufruf von `swap()` in Zeile 12 übergibt als Argumente die Adressen von `x` und `y`.

Die Funktion `swap()` deklariert in Zeile 20 die lokale Variable `temp`. Diese Variable braucht kein Zeiger zu sein, sie nimmt einfach den Wert von `*px` (das heißt, den Wert von `x` in der aufrufenden Funktion) während der Lebensdauer der Funktion auf. Nach dem Rücksprung aus der Funktion wird `temp` nicht mehr benötigt.

Zeile 25 weist `temp` den Wert an der Adresse von `px` zu. In Zeile 26 erhält `px` den Wert an der Adresse von `py`. In Zeile 27 wird der in `temp` zwischengespeicherte Wert (das heißt, der Originalwert an der Adresse von `px`) nach `py` übertragen.

In der aufrufenden Funktion sind die Werte, die als Adressen an `swap()` übergeben wurden, nun tatsächlich vertauscht.

11.6.2 Parameter mit Referenzen übergeben

Das obige Programm funktioniert zwar, die Syntax der Funktion `swap()` ist aber in zweierlei Hinsicht umständlich. Erstens führt die mehrfache Dereferenzierung der Zeiger in der Funktion `swap()` eher zu Programmfehlern und das Ganze ist schwer zu lesen. Zweitens legt die erforderliche Übergabe der Variablenadressen in der aufrufenden Funktion die inneren Abläufe von `swap()` für den Benutzer unnötig offen.

Ein Ziel von C++ ist, dass sich der Benutzer mit der Arbeitsweise einer Funktion nicht zu belasten braucht. Wenn man Zeiger übergibt, muss man in der aufrufenden Funktion Dinge erledigen, die eigentlich in die aufgerufene Funktion gehören. Die aufrufende Funktion muss wissen, dass sie die Adressen der zu vertauschenden Objekte zu übergeben hat.

Man sollte es der Funktion, die die Objekte vertauscht, überlassen, die Semantik der übergebenen Parameter zu interpretieren. Zu diesem Zweck setzt man Referenzen ein. Listing 11.6 zeigt eine Neufassung der Funktion `swap()` mit der Übergabe von Referenzen. Jetzt übergibt die aufrufende Funktion lediglich das Objekt und da die Parameter als Referenzen deklariert sind, erfolgt die Übergabe als Referenz. Die aufrufende Funktion muss keine speziellen Vorbereitungen mehr treffen.

Listing 11.6: passbyref.cpp – Die Funktion swap() mit Übergabe von Referenzen

```
// Listing 11.6 Zeigt die Parameterübergabe
// mit Referenzen
#include <iostream>

void swap(int &x, int &y);
// Das "&" besagt, dass die Funktion Referenzen erwartet.
int main()
{
    int x = 5, y = 10;

    std::cout << "Main. Vor Vertauschung, x: " << x
                                << " y: " << y << "\n";
    swap(x,y);
    std::cout << "Main. Nach Vertauschung, x: " << x
                                << " y: " << y << "\n";
    return 0;
}

void swap (int &rx, int &ry)
{
    int temp;

    std::cout << "Swap. Vor Vertauschung, rx: " << rx
                                << " ry: " << ry << "\n";

    temp = rx;
    rx = ry;
    ry = temp;

    std::cout << "Swap. Nach Vertauschung, rx: " << rx
                                << " ry: " << ry << "\n";
}
```

```
Main. Vor Vertauschung, x:5 y: 10
Swap. Vor Vertauschung, rx:5 ry:10
Swap. Nach Vertauschung, rx:10 ry:5
Main. Nach Vertauschung, x:10 y:5
```

Genau wie im Zeigerbeispiel deklariert dieses Programm in Zeile 9 zwei Variablen und gibt deren Werte in den Zeilen 11 und 12 aus. Der Aufruf der Funktion `swap()` erfolgt in Zeile 13. Dieses Mal übergibt aber die aufrufende Funktion einfach die Variablen `x` und `y` und nicht deren Adressen.

Beim Aufruf von `swap()` geht die Programmausführung zu Zeile 19, wo die Variablen als Referenzen gekennzeichnet sind. Für die Ausgabe der Werte in den Zeilen 23 und 24 sind keine speziellen Operatoren erforderlich. Es handelt sich um die Aliasnamen für die Originalwerte, die man unverändert einsetzen kann.

Das Vertauschen der Werte geschieht in den Zeilen 26 bis 28, die Ausgabe in den Zeilen 30 und 31. Dann geht die Programmausführung zurück zur aufrufenden Funktion. In den Zeilen 14 und 15 gibt `main()` die Werte aus. Da die Parameter an `swap()` als Referenzen deklariert sind, werden die Werte aus `main()` als Referenz übergeben und sind demnach in `main()` ebenfalls vertauscht.

Referenzen lassen sich genauso komfortabel und einfach wie normale Variablen verwenden, sind aber leistungsfähig wie Zeiger und erlauben wie diese die Übergabe als Referenz.

11.7 Header und Prototypen von Funktionen

Funktionen mit Referenzen als Parameter lassen sich einfacher handhaben und der Code ist verständlicher. Woher weiß aber die aufrufende Funktion, ob die Übergabe als Referenz oder als Wert stattfindet? Als Klient (oder Benutzer) von `swap()` muss der Programmierer sicherstellen, dass `swap()` tatsächlich die Parameter ändert bzw. vertauscht.

Hier finden wir eine weitere Einsatzmöglichkeit für den Funktionsprototyp. Normalerweise sind die Prototypen in einer Header-Datei zusammengefasst. Die im Prototyp deklarierten Parameter verraten dem Programmierer, dass die an `swap()` übergebenen Werte als Referenz zu übergeben sind und demnach geeignet vertauscht werden.

Handelt es sich bei `swap()` um eine Elementfunktion einer Klasse, lassen sich diese Informationen aus der – ebenfalls in einer Header-Datei untergebrachten – Klassendeklaration ablesen.

In C++ stützen sich die Klienten von Klassen und Funktionen auf die Header-Datei, um alle erforderlichen Angaben zu erhalten. Die Header-Datei wirkt als Schnittstelle zur Klasse oder Funktion. Die eigentliche Implementierung bleibt dem Klienten verborgen. Damit kann sich der Programmierer auf das unmittelbare Problem konzentrieren und die Klasse oder Funktion einsetzen, ohne sich um deren Arbeitsweise kümmern zu müssen.

11.8 Mehrere Werte zurückgeben

Wie bereits erwähnt, können Funktionen nur einen Wert zurückgeben. Was macht man nun, wenn zwei Werte aus einer Funktion zurückzugeben sind? Eine Möglichkeit zur Lösung dieses Problems besteht in der Übergabe von zwei Objekten an die Funktion und zwar als Referenz. Da eine Funktion bei Übergabe als Referenz die Originalobjekte ändern kann, lassen sich mit der Funktion praktisch zwei Informationsteile zurückgeben. Diese Lösung umgeht den Rückgabewert der Funktion, den man am besten für die Meldung von Fehlern vorsieht.

Auch hier kann man sowohl mit Referenzen als auch mit Zeigern arbeiten. Listing 11.7 zeigt eine Funktion, die drei Werte zurückgibt, zwei als Zeigerparameter und einen als Rückgabewert der Funktion.

Listing 11.7: returnwithptr.cpp – Rückgabe von Werten mit Zeigern

```
 1: // Listing 11.7
 2: // Mehrere Werte aus einer Funktion zurückgeben
 3: #include <iostream>
 4:
 5: short Factor(int, int*, int*);
 6:
 7: int main()
 8: {
 9:     int number, squared, cubed;
10:     short error;
11:
12:     std::cout << "Bitte eine Zahl eingeben (0 - 20): ";
13:     std::cin >> number;
14:
15:     error = Factor(number, &squared, &cubed);
16:
17:     if (!error)
18:     {
19:         std::cout << "Zahl: " << number << "\n";
20:         std::cout << "Quadrat: " << squared << "\n";
```

```
21:         std::cout << "Dritte Potenz: "  << cubed   << "\n";
22:     }
23:     else
24:         std::cout << "Fehler!!\n";
25:     return 0;
26: }
27:
28: short Factor(int n, int *pSquared, int *pCubed)
29: {
30:     short Value = 0;
31:     if (n > 20)
32:         Value = 1;
33:     else
34:     {
35:         *pSquared = n*n;
36:         *pCubed = n*n*n;
37:         Value = 0;
38:     }
39:     return Value;
40: }
```

```
Bitte eine Zahl eingeben (0-20): 3
Zahl: 3
Quadrat: 9
Dritte Potenz: 27
```

Zeile 9 definiert `number`, `squared` und `cubed` als `int`. Die Variable `number` nimmt die vom Benutzer eingegebene Zahl auf. Diese wird zusammen mit den Adressen von `squared` und `cubed` an die Funktion `Factor()` übergeben.

`Factor()` testet den ersten – als Wert übergebenen – Parameter. Ist er größer als 20 (der Maximalwert, den diese Funktion behandeln kann), setzt die Funktion die Variable `Value` auf einen Fehlerwert. Beachten Sie, dass der Rückgabewert aus `Factor()` entweder für diesen Fehlerwert oder den Wert 0 reserviert ist, wobei 0 die ordnungsgemäße Funktionsausführung anzeigt. Die Rückgabe des entsprechenden Wertes findet in Zeile 39 statt.

Die Funktion liefert die eigentlich benötigten Werte, das Quadrat und die dritte Potenz von `number`, nicht über den normalen Rückgabemechanismus, sondern sie ändert die Werte direkt mithilfe der Zeiger, die an die Funktion übergeben werden.

In den Zeilen 35 und 36 erfolgt die Zuweisung der Rückgabewerte an die Zeiger. Zeile 37 setzt `Value` auf den Wert für erfolgreiche Ausführung (0) und Zeile 39 gibt `Value` zurück.

Als Verbesserung dieses Programms könnte man Folgendes deklarieren:

```
enum ERROR_VALUE { SUCCESS, FAILURE};
```

Dann gibt man nicht 0 oder 1 zurück, sondern SUCCESS (alles OK) oder FAILURE (fehlerhafte Ausführung).

11.8.1 Werte als Referenz zurückgeben

Das Programm in Listing 11.7 funktioniert zwar, lässt sich aber mit der Übergabe von Referenzen anstelle von Zeigern wartungsfreundlicher und übersichtlicher gestalten. Listing 11.8 zeigt das gleiche Programm, allerdings in der neuen Fassung mit Übergabe von Referenzen und Rückgabe eines Aufzählungstyps (ERR_CODE) für den Fehlerwert.

Listing 11.8: returnwithref.cpp – Neufassung von Listing 11.7 mit Übergabe von Referenzen

```
// Listing 11.8
// Rückgabe mehrerer Werte aus einer Funktion
// mit Referenzen
#include <iostream>

enum ERR_CODE { SUCCESS, ERROR };

ERR_CODE Factor(int, int&, int&);

int main()
{
    int number, squared, cubed;
    ERR_CODE result;

    std::cout << "Bitte eine Zahl eingeben (0 - 20): ";
    std::cin >> number;

    result = Factor(number, squared, cubed);

    if (result == SUCCESS)
    {
        std::cout << "Zahl: " << number << "\n";
        std::cout << "Quadrat: " << squared << "\n";
        std::cout << "Dritte Potenz: "  << cubed   << "\n";
    }
    else
        std::cout << "Fehler!!\n";
    return 0;
}

ERR_CODE Factor(int n, int &rSquared, int &rCubed)
{
    if (n > 20)
        return ERROR;   // Einfacher Fehlercode
```

```
35:     else
36:     {
37:         rSquared = n*n;
38:         rCubed = n*n*n;
39:         return SUCCESS;
40:     }
41: }
```

```
Bitte eine Zahl eingeben (0-20): 3
Zahl: 3
Quadrat: 9
Dritte Potenz: 27
```

Listing 11.8 ist mit Listing 11.7 bis auf zwei Ausnahmen identisch. Der Aufzählungstyp `ERR_CODE` erlaubt es, die Fehlermeldungen in den Zeilen 34 und 39 sowie die Fehlerbehandlung in Zeile 20 komfortabler zu schreiben. Die größere Änderung besteht allerdings darin, dass `Factor()` nun für die Übernahme von Referenzen statt Zeigern auf `squared` und `cubed` ausgelegt ist. Die Arbeit mit diesen Parametern gestaltet sich damit einfacher und ist verständlicher.

11.9 Fragen und Antworten

F: Warum gibt es Referenzen, wenn sich das Gleiche auch mit Zeigern realisieren lässt?

A: Referenzen sind leichter zu verwenden und zu verstehen. Die Umleitung bleibt verborgen und man muss die Variable nicht wiederholt dereferenzieren.

F: Warum gibt es Zeiger, wenn Referenzen einfacher zu handhaben sind?

A: Referenzen dürfen nicht null sein und lassen sich nicht erneut zuweisen. Zeiger bieten mehr Flexibilität, sind aber etwas schwieriger einzusetzen.

11.10 Workshop

Nachdem Sie Referenzen am praktischen Beispiel kennen gelernt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu testen.

11.10.1 Kontrollfragen

1. Was ist eine Referenz?
2. Mit welchem Operator erzeugt man eine Referenz?
3. Wie lautet die Adresse einer Referenz?
4. Nach welchem Standardmechanismus übergibt C++ Variablen an eine aufgerufene Funktion? Mit welchen Verfahren lässt sich dieser Mechanismus außer Kraft setzen?

11.10.2 Übungen

1. Kombinieren Sie den in Listing 11.5 (`passbyptr.cpp`) und Listing 11.6 (`passbyref.cpp`) gezeigten Mechanismen, sodass die eine Variable mithilfe von Zeigern und die andere als Referenz übergeben wird. Daraus können Sie ableiten, dass sich beide Verfahren austauschbar einsetzen lassen.
2. Modifizieren Sie Listing 11.7 (`returnwithptr.cpp`), um Referenzen anstelle von Zeigern zu verwenden.
3. Teilen Sie Listing 11.7 (`returnwithptr.cpp`) auf drei Dateien auf: das Hauptprogramm (`returnwithptr.cpp`), die `Factor()`-Funktion (`factor.cpp`) und den Funktionsprototyp für die `Factor()`-Funktion (`factor.hpp`). Fügen Sie Ihrem Projekt die Datei `factor.cpp` hinzu und kompilieren Sie es. Das zeigt, wie sich Funktions- und Klassenbibliotheken gemeinsam nutzen lassen. Es ist möglich, die Funktion einmal zu kompilieren und dann den Maschinencode zusammen mit der Header-Datei (dem Funktionsprototyp) wiederzuverwenden – die Arbeit wird effizienter.

11.10.3 Antworten zu den Kontrollfragen

1. Eine Referenz ist ein Aliasname oder Synonym für eine andere Variable oder ein anderes Objekt.
2. Eine Referenz deklariert man mithilfe des Et-Zeichens (`&`). Referenzen muss man initialisieren, wenn man sie deklariert. Im Unterschied zu Zeigern sind »Null-Referenzen« nicht erlaubt.

3. Die Adresse einer Referenz ist die Adresse der Variablen oder des Objekts, für die/das der Aliasname steht. Wendet man den Adressoperator auf eine Referenz an, scheint es, dass im Gegensatz zu einem Zeiger kein separater Speicherplatz reserviert wird.
4. C++ übergibt Variablen standardmäßig als Wert an eine Funktion, d.h. als Kopie der Variablen und als Originalwert. Das verhindert, dass die Funktion den Originalwert ändern kann. Dieser Mechanismus lässt sich mit Zeigern umgehen – sie übergeben die Adresse des Originalwertes. Referenzen sind eine weitere Möglichkeit – hier wird der Alias für die Originalvariable übergeben.

KAPITEL 12

jetzt lerne ich

Referenzen und Zeiger – weiterführende Themen

In dieser Stunde lernen Sie,

- wie man Programme durch Übergabe als Referenz effizienter gestaltet,
- wie man entscheidet, wann Referenzen und wann Zeiger zu verwenden sind,
- wie man Speicherprobleme bei Verwendung von Zeigern vermeidet,
- wie man Fallen bei der Verwendung von Referenzen umgeht.

12.1 Übergabe als Referenz aus Gründen der Effizienz

Übergibt man ein Objekt an eine Funktion als Wert, legt die Funktion eine Kopie des Objekts an. Bei der Rückgabe eines Objekts aus einer Funktion als Wert wird eine weitere Kopie erstellt.

Bei größeren, benutzerdefinierten Objekten machen sich diese Kopien deutlich bemerkbar. Man belegt mehr Speicher als eigentlich erforderlich und letztendlich läuft das Programm langsamer.

Die Größe eines benutzerdefinierten Objekts auf dem Stack ergibt sich aus der Summe seiner Elementvariablen. Bei diesen kann es sich wiederum um benutzerdefinierte Objekte handeln und die Übergabe einer derartig massiven Struktur durch das Kopieren auf den Stack geht sehr zu Lasten der Leistung und des Speicherverbrauchs.

Andere Faktoren sind ebenfalls zu berücksichtigen. Bei den von Ihnen erzeugten Klassen werden diese temporären Kopien erzeugt, wenn der Compiler einen speziellen Konstruktor aufruft: den Kopierkonstruktor. In Stunde 13 lernen Sie, wie Kopierkonstruktoren arbeiten und wie man eigene Konstruktoren erzeugt. Momentan geben wir uns mit der Aussage zufrieden, dass beim Aufruf des Kopierkonstruktors eine temporäre Kopie des Objekts auf den Stack gestellt wird.

Beim Rücksprung aus der Funktion wird das temporäre Objekt abgebaut und der Destruktor des Objekts aufgerufen. Wenn man ein Objekt als Wert zurückgibt, muss eine Kopie dieses Objekts angelegt und ebenfalls wieder abgebaut werden.

Bei großen Objekten gehen diese Konstruktor- und Destruktor-Aufrufe zu Lasten der Geschwindigkeit und des Speicherverbrauchs. Listing 12.1 verdeutlicht das mit einer vereinfachten Version eines benutzerdefinierten Objekts: `SimpleCat`. Ein reales Objekt wäre wesentlich größer und umfangreicher. Aber auch an diesem Objekt lässt sich zeigen, wie oft die Aufrufe von Konstruktor und Destruktor stattfinden.

Listing 12.1 erzeugt das Objekt `SimpleCat` und ruft dann zwei Funktionen auf. Die erste Funktion übernimmt `Cat` als Wert und gibt das Objekt als Wert zurück. Die zweite Funktion übernimmt ihre Argumente als Referenz, d.h. sie erhält einen Zeiger auf das Objekt und nicht das Objekt selbst und gibt einen Zeiger auf das Objekt zurück.

Listing 12.1: passobjectsbyref.cpp – Objekte als Referenz übergeben

```
 1: // Listing 12.1
 2: // Zeiger auf Objekte übergeben
 3: #include <iostream>
 4:
 5: class SimpleCat
 6: {
 7: public:
 8:     SimpleCat ();                    // Konstruktor
 9:     SimpleCat(SimpleCat&);     // Kopierkonstruktor
10:     ~SimpleCat();                    // Destruktor
11: };
12:
13: SimpleCat::SimpleCat()
14: {
15:     std::cout << "Simple Cat Konstruktor...\n";
16: }
17:
18: SimpleCat::SimpleCat(SimpleCat&)
19: {
20:     std::cout << "Simple Cat Kopierkonstruktor...\n";
21: }
22:
```

```
23: SimpleCat::~SimpleCat()
24: {
25:     std::cout << "Simple Cat Destruktor...\n";
26: }
27:
28: SimpleCat FunctionOne (SimpleCat theCat);
29: SimpleCat* FunctionTwo (SimpleCat *theCat);
30:
31: int main()
32: {
33:     std::cout << "Eine Katze erzeugen...\n";
34:     SimpleCat Frisky;
35:     std::cout << "FunctionOne aufrufen...\n";
36:     FunctionOne(Frisky);
37:     std::cout << "FunctionTwo aufrufen...\n";
38:     FunctionTwo(&Frisky);
39:     return 0;
40: }
41:
42: // FunctionOne, Übergabe als Wert
43: SimpleCat FunctionOne(SimpleCat theCat)
44: {
45:     std::cout << "Function One. Rueckkehr...\n";
46:     return theCat;
47: }
48:
49: // functionTwo, Übergabe als Referenz
50: SimpleCat* FunctionTwo (SimpleCat  *theCat)
51: {
52:     std::cout << "Function Two. Rueckkehr...\n";
53:     return theCat;
54: }
```

```
1:  Eine Katze erzeugen...
2:  SimpleCat Konstruktor...
3:  FunctionOne aufrufen...
4:  SimpleCat Kopierkonstruktor...
5:  FunctionOne. Rueckkehr...
6:  SimpleCat Kopierkonstruktor...
7:  SimpleCat Destruktor...
8:  SimpleCat Destruktor...
9:  FunctionTwo aufrufen...
10: FunctionTwo. Rueckkehr...
11: SimpleCat Destruktor...
```

Zeilennummern in der Ausgabe

Die hier angegebenen Zeilennummern erscheinen nicht in der Ausgabe, sondern dienen nur als Hilfsmittel für die Analyse.

Die Zeilen 7 bis 11 deklarieren eine sehr vereinfachte Klasse `SimpleCat`. Der Konstruktor, Kopierkonstruktor und Destruktor geben jeweils eine Meldung aus, damit man über die Zeitpunkte der Aufrufe informiert ist.

In Zeile 33 gibt `main()` eine Meldung aus, die als Ausgabezeile 1 zu sehen ist. In Zeile 34 wird ein `SimpleCat`-Objekt instanziert. Das bewirkt den Aufruf des Konstruktors. Die entsprechende Meldung erscheint als Ausgabezeile 2.

In Zeile 35 meldet die Funktion `main()`, dass sie die Funktion `FunctionOne()` aufruft (Ausgabezeile 3). Da `FunctionOne()` mit Übergabe des Objekts `SimpleCat` als Wert aufgerufen wird, legt die Funktion ein lokales Objekt als Kopie des Objekts `SimpleCat` auf dem Stack an. Das bewirkt den Aufruf des Kopierkonstruktors, der die Ausgabezeile 4 erzeugt.

Die Programmausführung springt zur Zeile 45 in der aufgerufenen Funktion, die eine Meldung als Ausgabezeile 5 bringt. Die Funktion kehrt dann zurück und gibt dabei das Objekt `SimpleCat` als Wert zurück. Das erzeugt eine weitere Kopie des Objekts, wobei der Kopierkonstruktor aufgerufen und Ausgabezeile 6 produziert wird.

Das Programm weist den Rückgabewert aus der Funktion `FunctionOne()` keinem Objekt zu, sodass das für die Rückgabe erzeugte Objekt mit Aufruf des Destruktors (Meldung in Ausgabezeile 7) verworfen wird. Da `FunctionOne()` beendet ist, verliert die lokale Kopie ihren Gültigkeitsbereich und wird mit Aufruf des Destruktors (Meldung in Ausgabezeile 8) abgebaut.

Das Programm kehrt nach `main()` zurück und ruft `FunctionTwo()` mit Übergabe der Parameter als Referenz auf. Da die Funktion keine Kopie anlegt, gibt es auch keine Ausgabemeldung. `FunctionTwo()` produziert die als Ausgabezeile 10 erscheinende Meldung und gibt dann das Objekt `SimpleCat` – auch hier als Referenz – zurück. Aus diesem Grund finden hier ebenfalls keine Aufrufe von Konstruktor oder Destruktor statt.

Schließlich endet das Programm und `Frisky` verliert seinen Gültigkeitsbereich. Das führt zum letzten Aufruf des Destruktors und zur Meldung in Ausgabezeile 11.

Summa summarum: Aufgrund der Übergabe als Wert produziert der Aufruf der Funktion `FunctionOne()` zwei Aufrufe des Kopierkonstruktors und zwei Aufrufe des Destruktors, während der Aufruf von `FunctionTwo()` keinerlei derartige Aufrufe erzeugt.

12.1.1 Einen konstanten Zeiger übergeben

Die Übergabe eines Zeigers bei `FunctionTwo()` ist zwar effizienter, aber auch gefährlicher. `FunctionTwo()` darf nicht das übergebene Objekt `SimpleCat` ändern, obwohl die Funktion die Adresse von `SimpleCat` erhält. Damit genießt das Objekt nicht mehr den Schutz gegenüber Änderungen wie bei der Übergabe als Wert.

Die Übergabe als Wert verhält sich so, als würde man einem Museum eine Fotografie des eigenen Kunstwerks geben und nicht das Kunstwerk selbst. Krakelt jemand im Museum auf Ihrem Bild herum, bleibt Ihnen in jedem Fall das Original erhalten. Bei der Übergabe als Referenz übermittelt man dem Museum lediglich seine Heimatadresse und lädt die Besucher ein, das echte Meisterwerk im eigenen Haus anzusehen.

Wenn man die Sicherheit einer Übergabe als Wert und die Effizienz einer Übergabe als Referenz haben möchte, übergibt man einen konstanten Zeiger auf `SimpleCat`. Damit verhindert man den Aufruf nicht konstanter Methoden auf `SimpleCat` und schützt demzufolge das Objekt gegen Änderungen. Listing 12.2 verdeutlicht dieses Konzept.

Listing 12.2: passconst-ptr.cpp – Übergabe von konstanten Zeigern

```
 1: // Listing 12.2
 2: // Zeiger auf Objekte übergeben
 3: #include <iostream>
 4:
 5: class SimpleCat
 6: {
 7: public:
 8:     SimpleCat();
 9:     SimpleCat(SimpleCat&);
10:     ~SimpleCat();
11:
12:     int GetAge() const { return itsAge; }
13:     void SetAge(int age) { itsAge = age; }
14:
15: private:
16:     int itsAge;
17: };
18:
19: SimpleCat::SimpleCat()
20: {
21:     std::cout << "Simple Cat Konstruktor...\n";
22:     itsAge = 1;
23: }
24:
25: SimpleCat::SimpleCat(SimpleCat&)
26: {
27:     std::cout << "Simple Cat Kopierkonstruktor...\n";
28: }
```

```
29:
30: SimpleCat::~SimpleCat()
31: {
32:     std::cout << "Simple Cat Destruktor...\n";
33: }
34:
35: const SimpleCat * const
36: FunctionTwo (const SimpleCat * const theCat);
37:
38: int main()
39: {
40:     std::cout << "Eine Katze erzeugen...\n";
41:     SimpleCat Frisky;
42:     std::cout << "Frisky ist ";
43:     std::cout << Frisky.GetAge() << " Jahre alt.\n";
44:     int age = 5;
45:     Frisky.SetAge(age);
46:     std::cout << "Frisky ist ";
47:     std::cout << Frisky.GetAge() << " Jahre alt.\n";
48:     std::cout << "FunctionTwo aufrufen...\n";
49:     FunctionTwo(&Frisky);
50:     std::cout << "Frisky ist ";
51:     std::cout << Frisky.GetAge() << " Jahre alt.\n";
52:     return 0;
53: }
54:
55: // functionTwo übernimmt einen konstanten Zeiger
56: const SimpleCat * const
57: FunctionTwo (const SimpleCat * const theCat)
58: {
59:     std::cout << "Function Two. Rueckkehr...\n";
60:     std::cout << "Frisky ist jetzt " << theCat->GetAge();
61:     std::cout << " Jahre alt.\n";
62:     // theCat->SetAge(8);   const!
63:     return theCat;
64: }
```

```
Eine Katze erzeugen...
SimpleCat Konstruktor...
Frisky ist 1 Jahre alt.
Frisky ist 5 Jahre alt.
FunctionTwo aufrufen...
FunctionTwo. Rueckkehr...
Frisky ist jetzt 5 Jahre alt.
Frisky ist 5 Jahre alt.
SimpleCat Destruktor...
```

Die Klasse `SimpleCat` hat zwei Zugriffsfunktionen erhalten: die konstante Funktion `GetAge()` in Zeile 12 und die nicht konstante Funktion `SetAge()` in Zeile 13. Außerdem ist die Elementvariable `itsAge` in Zeile 16 neu hinzugekommen.

In den Definitionen von Konstruktor, Kopierkonstruktor und Destruktor stehen immer noch die Ausgabemeldungen für ihre Aufrufe. Allerdings findet überhaupt kein Aufruf des Kopierkonstruktors statt, da die Übergabe des Objekts als Referenz erfolgt und damit keine Kopien angelegt werden. Zeile 41 erzeugt ein Objekt, die Zeilen 42 und 43 geben dessen Standardwert für das Alter (`Age`) aus.

In Zeile 45 wird `itsAge` mit der Zugriffsfunktion `SetAge()` gesetzt und das Ergebnis in den Zeilen 46 und 47 ausgegeben. `FunctionOne()` kommt in diesem Programm nicht zum Einsatz. `FunctionTwo()` weist leichte Änderungen auf: Jetzt sind in den Zeilen 35 und 36 der Parameter und der Rückgabewert für einen konstanten Zeiger auf ein konstantes Objekt und die Rückgabe eines konstanten Zeigers auf ein konstantes Objekt deklariert.

Da die Übergabe der Parameter und des Rückgabewertes weiterhin als Referenz erfolgt, legt die Funktion keine Kopien an und ruft auch nicht den Kopierkonstruktor auf. Der Zeiger in `FunctionTwo()` ist jetzt allerdings eine Konstante und kann demnach nicht die nicht konstante Methode `SetAge()` aufrufen. Deshalb ist der Aufruf von `SetAge()` in Zeile 62 auskommentiert – das Programm ließe sich sonst nicht kompilieren.

Beachten Sie, dass das in `main()` erzeugte Objekt nicht konstant ist und `Frisky` die Funktion `SetAge()` aufrufen kann. Die Adresse dieses nicht konstanten Objekts wird an die Funktion `FunctionTwo()` übergeben. Da aber in `FunctionTwo()` der Zeiger als konstant deklariert ist, wird das Objekt wie ein konstantes Objekt behandelt!

12.2 Referenzen als Alternative

Listing 12.2 vermeidet zusätzliche Kopien durch konstante Zeiger auf konstante Objekte und spart damit die Aufrufe von Kopierkonstruktor und Destruktor ein. Gleichzeitig löst sich damit das Problem, dass es der aufgerufenen Funktion nicht möglich ist, Änderungen an den als Parameter übergebenen Objekten durchzuführen. Allerdings ist das Ganze immer noch etwas umständlich, weil die an die Funktion übergebenen Objekte Zeiger sind.

Da man weiß, dass die Parameter niemals Null sind, lässt sich mit der Funktion einfacher arbeiten, wenn man Referenzen anstelle von Zeigern übergibt. Dieses Konzept zeigt Listing 12.3 als Modifikation von Listing 12.2.

Listing 12.3: passreftoobj.cpp – Referenzen auf Objekte übergeben

```
// Listing 12.3
// Referenzen auf Objekte übergeben
#include <iostream>

class SimpleCat
{
public:
    SimpleCat();
    SimpleCat(SimpleCat&);
    ~SimpleCat();

    int GetAge() const { return itsAge; }
    void SetAge(int age) { itsAge = age; }

private:
    int itsAge;
};

SimpleCat::SimpleCat()
{
    std::cout << "Simple Cat Konstruktor...\n";
    itsAge = 1;
}

SimpleCat::SimpleCat(SimpleCat&)
{
    std::cout << "Simple Cat Kopierkonstruktor...\n";
}

SimpleCat::~SimpleCat()
{
    std::cout << "Simple Cat Destruktor...\n";
}

const SimpleCat & FunctionTwo (const SimpleCat & theCat);

int main()
{
    std::cout << "Eine Katze erzeugen...\n";
    SimpleCat Frisky;
    std::cout << "Frisky ist " << Frisky.GetAge()
              << " Jahre alt.\n";

    int age = 5;
    Frisky.SetAge(age);
    std::cout << "Frisky ist " << Frisky.GetAge()
              << " Jahre alt.\n";

```

```
49:     std::cout << "FunctionTwo aufrufen...\n";
50:     FunctionTwo(Frisky);
51:     std::cout << "Frisky ist " << Frisky.GetAge()
52:               << " Jahre alt.\n";
53:     return 0;
54: }
55:
56: // functionTwo übergibt eine Referenz auf ein konstantes Objekt
57: const SimpleCat & FunctionTwo (const SimpleCat & theCat)
58: {
59:     std::cout << "Function Two. Rueckkehr...\n";
60:     std::cout << "Frisky ist jetzt " << theCat.GetAge()
61:               << " Jahre alt.\n";
62:     // theCat.SetAge(8);   const!
63:     return theCat;
64: }
```

```
Eine Katze erzeugen...
SimpleCat Konstruktor...
Frisky ist 1 Jahre alt.
Frisky ist 5 Jahre alt.
FunctionTwo aufrufen
FunctionTwo. Rueckkehr...
Frisky ist jetzt 5 Jahre alt.
Frisky ist 5 Jahre alt.
SimpleCat Destruktor...
```

Die Ausgabe ist mit der von Listing 12.2 produzierten Ausgabe identisch. Die einzige signifikante Änderung besteht darin, dass `FunctionTwo()` jetzt die Übernahme und Rückgabe mit einer Referenz auf ein konstantes Objekt realisiert. Auch hier ist das Arbeiten mit Referenzen etwas einfacher als das Arbeiten mit Zeigern. Man erreicht die gleiche Einsparung und Effizienz sowie die Sicherheit, die durch `const` geboten wird.

12.3 Wann verwendet man Referenzen und wann Zeiger?

C++-Programmierer arbeiten lieber mit Referenzen als mit Zeigern, weil sie sauberer und einfacher zu verwenden sind. Allerdings kann man Referenzen nicht erneut zuweisen. Wenn man zuerst auf ein Objekt und dann auf ein anderes zeigen muss, ist die Verwendung eines Zeigers Pflicht. Referenzen dürfen nicht Null sein. Ist dennoch mit einem Null-Objekt zu rechnen, kann man nicht mit Referenzen arbeiten und muss auf Zeiger ausweichen. Möchte man

Speicher dynamisch auf dem Heap reservieren, muss man Zeiger verwenden, wie es die letzten Lektionen erläutert haben.

12.4 Referenzen auf nicht mehr vorhandene Objekte

Nachdem sich C++-Programmierer einmal mit der Übergabe als Referenz angefreundet haben, können sie kaum noch davon lassen. Man kann allerdings auch des Guten zuviel tun. Denken Sie daran, dass eine Referenz immer ein Alias auf irgendein anderes Objekt ist. Wenn man eine Referenz in oder aus einer Funktion übergibt, sollte man sich die kritische Frage stellen »was ist das Objekt, das ich unter einem Alias anspreche, und existiert es auch wirklich, wenn ich es verwende?«

Listing 12.4 verdeutlicht die Gefahr bei der Rückgabe einer Referenz auf ein Objekt, das nicht mehr existiert.

Listing 12.4: returnref.cpp – Rückgabe einer Referenz auf ein nicht mehr existierendes Objekt

```
// Listing 12.4
// Rückgabe einer Referenz auf ein Objekt,
// das nicht mehr existiert
#include <iostream>

class SimpleCat
{
public:
    SimpleCat (int age, int weight);
    ~SimpleCat() {}
    int GetAge() { return itsAge; }
    int GetWeight() { return itsWeight; }
private:
    int itsAge;
    int itsWeight;
};

SimpleCat::SimpleCat(int age, int weight):
itsAge(age), itsWeight(weight) {}

SimpleCat &TheFunction();

int main()
{
    SimpleCat &rCat = TheFunction();
    int age = rCat.GetAge();
    std::cout << "rCat ist " << age << " Jahre alt!\n";
    return 0;
}
```

```
30:
31: SimpleCat &TheFunction()
32: {
33:     SimpleCat Frisky(5,9);
34:     return Frisky;
35: }
```

Wenn Sie das Projekt mit dem Code von Listing 12.4 erstellen, liefert der Borland-Compiler folgende Fehlermeldung:

```
"returnref.cpp": E2363 Versuch, eine Referenz an die lokale Variable
'Frisky' zurückzugeben in Funktion TheFunction() in Zeile 34
```

Intelligente Compiler

Intelligente Compiler erkennen die Referenz auf ein Null-Objekt und melden einen Compiler-Fehler wie in der obigen Ausgabe zu sehen. Mit anderen Compilern lässt sich das Projekt möglicherweise anstandslos kompilieren und ausführen. Allerdings sei angemerkt, dass es sich nicht gerade um guten Programmierstil handelt und man sollte auf derartige Konstruktionen verzichten, selbst wenn sie der Compiler erlaubt. C++BuilderX gehört zu den intelligenten Compilern, die eine Fehlermeldung (siehe oben) liefern.

Die Zeilen 6 bis 16 deklarieren die Klasse `SimpleCat`. In Zeile 25 wird eine Referenz auf `SimpleCat` mit dem Ergebnis des Aufrufs der Funktion `TheFunction()` initialisiert. Die Funktion `TheFunction()`ist in Zeile 21 deklariert und gibt eine Referenz auf `SimpleCat` zurück.

Der Rumpf der Funktion `TheFunction()` deklariert ein lokales Objekt vom Typ `SimpleCat` und initialisiert dessen Alter (`age`) und Gewicht (`weight`). Dann gibt die Funktion dieses lokale Objekt als Referenz zurück. Intelligente Compiler fangen diesen Fehler ab und erlauben gar nicht erst den Start des Programms. Andere lassen die Programmausführung zu, was aber zu unvorhersehbaren Ergebnissen führt.

Bei Rückkehr der Funktion `TheFunction()` wird das lokale Objekt `Frisky` zerstört. (Der Autor versichert, dass das schmerzlos geschieht.) Die von dieser Funktion zurückgegebene Referenz wird zu einem Alias für ein nicht existentes Objekt und das geht irgendwann schief.

12.5 Referenzen auf Objekte im Heap zurückgeben

Man mag versucht sein, das Problem in Listing 12.4 zu lösen, indem man `Frisky` von der Funktion `TheFunction()` auf dem Heap erzeugen lässt. Damit existiert `Frisky` auch noch nach dem Rücksprung aus `TheFunction()`.

Das Problem bei dieser Variante ist: Was fängt man mit dem für `Frisky` zugewiesenen Speicher an, wenn die Arbeit damit beendet ist? Listing 12.5 verdeutlicht dieses Problem.

Listing 12.5: leaky.cpp – Speicherlücken

```
// Listing 12.5
// Beseitigen von Speicherlücken
#include <iostream>

class SimpleCat
{
public:
    SimpleCat (int age, int weight);
    ~SimpleCat() {}
    int GetAge() { return itsAge; }
    int GetWeight() { return itsWeight; }

private:
    int itsAge;
    int itsWeight;
};

SimpleCat::SimpleCat(int age, int weight):
itsAge(age), itsWeight(weight) {}

SimpleCat & TheFunction();

int main()
{
    SimpleCat & rCat = TheFunction();
    int age = rCat.GetAge();
    std::cout << "rCat ist " << age << " Jahre alt!\n";
    std::cout << "&rCat: " << &rCat << std::endl;
    // Wie wird man diesen Speicher wieder los?
    SimpleCat * pCat = &rCat;
    delete pCat;
    // Worauf verweist denn nun rCat?!?
    return 0;
}
```

```
36: SimpleCat &TheFunction()
37: {
38:     SimpleCat * pFrisky = new SimpleCat(5,9);
39:     std::cout << "pFrisky: " << pFrisky << std::endl;
40:     return *pFrisky;
41: }
```

```
pFrisky:  8861880
rCat ist 5 Jahre alt!
&rCat:    8861880
```

Die Ausgabe kann bei Ihrem Computer ein etwas anderes Aussehen haben, je nachdem, was noch im Speicher steht und wie viel Speicher verfügbar ist. Beispielsweise ist auch Folgendes möglich:

```
pFrisky:  0x00431CA0
rCat ist 5 Jahre alt!
&rCat:    0x00431CA0
```

Vorsicht beim Löschen

Dieses Programm lässt sich kompilieren und scheint zu arbeiten. Es handelt sich aber um eine Zeitbombe, die nur auf ihre Zündung wartet.

Die Funktion `TheFunction()` wurde geändert, sodass sie jetzt nicht mehr eine Referenz auf eine lokale Variable zurückgibt. Zeile 38 reserviert Speicher auf dem Heap und weist ihn einem Zeiger zu. Es folgt die Ausgabe der vom Zeiger gespeicherten Adresse. Dann wird der Zeiger dereferenziert und das `SimpleCat`-Objekt als Referenz zurückgegeben.

Zeile 25 weist das Ergebnis des Funktionsaufrufs von `TheFunction()` einer Referenz auf ein `SimpleCat`-Objekt zu. Über dieses Objekt wird das Alter der Katze ermittelt und in Zeile 27 ausgegeben.

Um sich davon zu überzeugen, dass die in `main()` deklarierte Referenz auf das in der Funktion `TheFunction()` auf dem Heap abgelegte Objekt verweist, wird der Adressoperator auf `rCat` angewandt. Tatsächlich erscheint die Adresse des betreffenden Objekts und diese stimmt mit der Adresse des Objekts auf dem Heap überein.

So weit so gut. Wie aber wird dieser Speicher freigegeben? Auf der Referenz kann man `delete` nicht aufrufen. Eine clevere Lösung ist die Erzeugung eines anderen Zeigers und dessen Initialisierung mit der Adresse, die man aus `rCat` erhält. Damit löscht man den Speicher und stopft die Speicherlücke. Ein Problem bleibt trotzdem noch: Worauf bezieht sich `rCat` nach Zeile 31? Wie bereits erwähnt, muss eine Referenz immer als Alias für ein tatsächliches Objekt agieren. Wenn sie auf ein Null-Objekt verweist (wie in diesem Fall), ist das Programm ungültig.

Null-Objekte

Man kann nicht genug darauf hinweisen, dass sich ein Programm mit einer Referenz auf ein Null-Objekt zwar kompilieren lässt, aber das Programm ist nicht zulässig und seine Leistung nicht vorhersehbar.

Für dieses Problem gibt es eigentlich zwei Lösungen. Bei der ersten gibt man einen Zeiger auf den in Zeile 38 reservierten Speicher zurück. Dann kann die aufrufende Funktion den Zeiger nach Abschluss der Arbeiten löschen. Ändern Sie dazu den Rückgabewert von `TheFunction()` in einen Zeiger (statt einer Referenz) und geben Sie den Zeiger anstelle des dereferenzierten Zeigers zurück:

```
SimpleCat * TheFunction()
{
   SimpleCat * pFrisky = new SimpleCat(5,9);
   std;;cout << "pFrisky: " << pFrisky << std::endl;
   return pFrisky;  // den Zeiger zurückgeben
}
```

Die alternative und anzustrebende Lösung besteht darin, das Objekt in der aufrufenden Funktion zu deklarieren und es dann als Referenz an `TheFunction()` zu übergeben. Dieses Verfahren hat den Vorteil, dass die Funktion, die den Speicher reserviert (die aufrufende Funktion), auch dafür verantwortlich ist, den Speicher wieder freizugeben. Wie der nächste Abschnitt zeigt, ist das die bevorzugte Variante.

12.6 Wem gehört der Zeiger?

Wenn man Speicher auf dem Heap reserviert, bekommt man einen Zeiger zurück. Es ist zwingend erforderlich, einen Zeiger auf diesen Speicher aufzubewahren, denn wenn der Zeiger verloren ist, lässt sich der Speicher nicht mehr löschen – es entsteht eine Speicherlücke.

Wenn man diesen Speicherblock zwischen Funktionen übergibt, nimmt eine der Funktionen den Zeiger »in Besitz«. Normalerweise wird der Wert im Block mittels Referenzen übergeben und die Funktion, die den Speicherblock angelegt hat, löscht ihn auch wieder. Das ist aber nur eine allgemeine Regel und kein eisernes Gesetz.

Gefahr ist im Verzug, wenn eine Funktion Speicher erzeugt und eine andere ihn freigibt. Unklare Besitzverhältnisse in Bezug auf den Zeiger können zu zwei Problemen führen: Man vergisst, den Zeiger zu löschen, oder löscht ihn zweimal. Beides bringt ernsthafte Schwierigkeiten für das Programm. Funktionen sollte man sicherheitshalber so konzipieren, dass sie den von ihnen reservierten Speicher auch selbst wieder freigeben.

Wenn Sie eine Funktion schreiben, die einen Speicherblock erzeugen muss und ihn dann zurück an die aufrufende Funktion übergibt, sollten Sie eine Änderung der Schnittstelle in Betracht ziehen. Lassen Sie die aufrufende Funktion den Speicher reservieren und übergeben Sie ihn an die Funktion als Referenz. Damit nehmen Sie die gesamte Speicherverwaltung aus dem Programm heraus und überlassen sie der Funktion, die auf das Löschen des Speichers vorbereitet ist.

Was Sie tun sollten

Übergeben Sie Parameter als Wert, wenn es erforderlich ist.

Geben Sie als Wert zurück, wenn es erforderlich ist.

Achtung

Übergeben Sie nicht als Referenz, wenn das referenzierte Objekt seinen Gültigkeitsbereich verlieren kann.

Verwenden Sie keine Referenzen auf Null-Objekte.

12.7 Fragen und Antworten

F: Warum verwendet man Zeiger, wenn Referenzen einfacher zu handhaben sind?

A: Referenzen dürfen nicht Null sein und lassen sich nicht erneut zuweisen. Zeiger bieten mehr Flexibilität, sind aber etwas schwieriger einzusetzen.

F: Warum gestaltet man die Rückgabe aus einer Funktion überhaupt als Wert?

A: Wenn das zurückzugebende Objekt lokal ist, muss man als Wert zurückgeben. Andernfalls gibt man eine Referenz auf ein nicht existentes Objekt zurück.

F: Warum gibt man nicht immer als Wert zurück, wenn die Rückgabe als Referenz gefährlich ist?

A: Die Rückgabe als Referenz ist weitaus effizienter. Man spart Speicher, und das Programm läuft schneller.

12.8 Workshop

Nachdem Sie sich mit weiterführenden Themen zu Zeigern und Referenzen beschäftigt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zum Compiler zu festigen.

12.8.1 Kontrollfragen

1. Warum ist es besser, als Referenz zu übergeben, wenn es sich um große Datenmengen handelt?
2. Weshalb verwendet man bei einer Übergabe als Referenz einen `const`-Zeiger?
3. Weshalb verwendet man bei der Übergabe als Referenz einen Zeiger anstelle einer Referenz?
4. Kann man eine Referenz auf eine Zeigervariable erzeugen?

12.8.2 Übungen

1. Teilen Sie Listing 12.1 (`passobjectsbyref.cpp`) auf drei Dateien auf: das Hauptprogramm und die Funktionen (`passobjectsbyref.cpp`), die Klassenmethoden (`SimpleCat.cpp`) und die Klassendeklaration (`SimpleCat.hpp`). Fügen Sie Ihrem Projekt die Datei `Simple.Cat.cpp` hinzu und kompilieren Sie es. Das zeigt, wie sich Funktions- und Klassenbibliotheken gemeinsam nutzen lassen. Es ist möglich, die Funktion einmal zu kompilieren und dann den Maschinencode zusammen mit der Header-Datei (dem Funktionsprototyp) wiederzuverwenden – die Arbeit wird effizienter.
2. Modifizieren Sie Listing 12.5 (`leaky.cpp`): Verwenden Sie Zeiger im Aufruf von `TheFunction()` und verhindern Sie mit der passenden Löschmethode, dass Speicherlücken entstehen.
3. Modifizieren Sie Listing 12.1 (`passobjectsbyref.cpp`) und Listing 12.3 (`passreftoobj.cpp`), um die Adressen der Variablen vor und nach den Funktionsaufrufen auszugeben. Damit erhalten Sie einen Einblick in die Abläufe.

12.8.3 Antworten zu den Kontrollfragen

1. Beim Standardmechanismus, der Übergabe als Wert, wird eine Kopie der übergebenen Variablen erzeugt. Bei einem großen Objekt kann das beträchtlichen Zeit- und Speicherbedarf bedeuten.
2. Durch das Schlüsselwort `const` verhindert der Compiler, dass die aufgerufene Funktion den Wert des Zeigers verändert. Man erhält den Schutz der Übergabe als Wert ohne die zusätzlichen Kosten für das Erstellen von Kopien zahlen zu müssen.
3. Wenn eine aufgerufene Funktion Speicher auf dem Heap reserviert, lässt sich diese Adresse mithilfe von Zeigern an den Aufrufer zurückgeben. Um eine Speicherlücke zu verhindern, darf man nicht vergessen, den reservierten Speicher zu gegebener Zeit freizugeben.
4. Warum nicht? Allerdings darf man dabei nicht den Überblick verlieren – denn Zeiger sind ohnehin keine leichte Kost für Einsteiger in C++.

KAPITEL 13

jetzt lerne ich

Funktionen – weiterführende Themen

In dieser Stunde lernen Sie,

- wie man Elementfunktionen überlädt,
- wie man Funktionen schreibt, die Klassen mit dynamisch zugewiesenen Variablen unterstützen,
- wie man Objekte initialisiert,
- wie man Kopierkonstruktoren erstellt.

13.1 Überladene Elementfunktionen

In Stunde 5 haben Sie gelernt, wie man Funktionspolymorphie – oder das Überladen von Funktionen – implementiert, indem man zwei oder mehrere Funktionen mit demselben Namen aber mit unterschiedlichen Parametern schreibt. Elementfunktionen von Klassen lassen sich ebenfalls überladen.

Die in Listing 13.1 dargestellte Klasse `Rectangle` hat zwei Funktionen namens `DrawShape()`. Die eine Funktion hat keine Parameter und zeichnet das `Rectangle`-Objekt auf der Basis der aktuellen Werte in der Klasse. Die andere Funktion übernimmt zwei Werte, `width` (Breite) und `height` (Höhe); sie zeichnet das `Rectangle`-Objekt mit den übergebenen Werten und ignoriert die aktuellen Klassenwerte.

Listing 13.1: overloadfunctions.cpp – Überladen von Elementfunktionen

```
// Listing 13.1 Überladen von Elementfunktionen
#include <iostream>

// Deklaration der Klasse Rectangle
class Rectangle
{
public:
    // Konstruktoren
    Rectangle(int width, int height);
    ~Rectangle(){}

    // Überladene Klassenfunktion DrawShape
    void DrawShape() const;
    void DrawShape(int aWidth, int aHeight) const;

private:
    int itsWidth;
    int itsHeight;
};

// Implementierung des Konstruktors
Rectangle::Rectangle(int width, int height)
{
    itsWidth = width;
    itsHeight = height;
}


// Überladene Elementfunktion DrawShape - übernimmt
// keine Werte. Zeichnet mit aktuellen Datenelementen.
void Rectangle::DrawShape() const
{
    DrawShape( itsWidth, itsHeight);
}


// Überladene Elementfunktion DrawShape - übernimmt
// zwei Werte und zeichnet nach diesen Parametern.
void Rectangle::DrawShape(int width, int height) const
{
    for (int i = 0; i<height; i++)
    {
        for (int j = 0; j< width; j++)
        {
            std::cout << "*";
        }
        std::cout << "\n";
    }
```

```
49: }
50:
51: // Rahmenprogramm zur Demo für überladene Funktionen
52: int main()
53: {
54:     // Ein Rechteck mit 30,5 initialisieren
55:     Rectangle theRect(30,5);
56:     std::cout << "DrawShape(): \n";
57:     theRect.DrawShape();
58:     std::cout << "\nDrawShape(40,2): \n";
59:     theRect.DrawShape(40,2);
60:     return 0;
61: }
```

```
DrawShape():
******************************
******************************
******************************
******************************
******************************

DrawShape(40,2):
****************************************
****************************************
```

Den für das Thema wichtigen Code finden Sie im Listing in den Zeilen 13 und 14, wo die Funktion `DrawShape()` überladen wird. Die Implementierung für diese überladenen Klassenmethoden ist in den Zeilen 29 bis 49 untergebracht. Beachten Sie, dass die Version von `DrawShape()`, die keine Parameter übernimmt, einfach die Version aufruft, die zwei Parameter übernimmt, und dabei die aktuellen Elementvariablen übergibt. Man sollte tunlichst vermeiden, den gleichen Code in zwei Funktionen zu wiederholen. Es ist immer schwierig und fehlerträchtig, beide Funktionen synchron zu halten, wenn man Änderungen an der einen oder anderen Funktion vornimmt.

Das Rahmenprogramm in den Zeilen 51 bis 61 erzeugt ein `Rectangle`-Objekt und ruft dann `DrawShape()` auf. Beim ersten Mal werden keine Parameter und beim zweiten Aufruf zwei Ganzzahlen übergeben.

Der Compiler entscheidet anhand der Anzahl und des Typs der eingegebenen Parameter, welche Methode aufzurufen ist. Man kann sich eine dritte überladene Funktion namens `DrawShape()` vorstellen, die eine Abmessung und einen Aufzählungstyp entweder für Breite oder Höhe – je nach Wahl des Benutzers – übernimmt.

13.2 Standardwerte

Genau wie »klassenlose« Funktionen einen oder mehrere Standardwerte haben können, gilt das auch für alle Elementfunktionen einer Klasse. Für die Deklaration von Standardwerten sind ebenfalls die gleichen Regeln anzuwenden, wie es Listing 13.2 verdeutlicht.

Listing 13.2: usingdefaults.cpp – Standardwerte verwenden

```
// Listing 13.2 Standardwerte in Elementfunktionen
#include <iostream>

// Deklaration der Klasse Rectangle
class Rectangle
{
public:
     // Konstruktoren
     Rectangle(int width, int height);
     ~Rectangle(){}
     void DrawShape(int aWidth, int aHeight,
     bool UseCurrentVals = false) const;
private:
     int itsWidth;
     int itsHeight;
};

// Implementierung des Konstruktors
Rectangle::Rectangle(int width, int height):
itsWidth(width),       // Initialisierungen
itsHeight(height)
{}                     // Leerer Rumpf

// Standardwerte für dritten Parameter
void Rectangle::DrawShape(
     int width,
     int height,
     bool UseCurrentValue
     ) const
{
     int printWidth;
     int printHeight;

     if (UseCurrentValue == true)
     {
         printWidth = itsWidth; // Aktuelle Klassenwerte verwenden
         printHeight = itsHeight;
     }
     else
     {
```

```
41:          printWidth = width;          // Parameterwerte verwenden
42:          printHeight = height;
43:      }
44:
45:      for (int i = 0; i<printHeight; i++)
46:      {
47:          for (int j = 0; j< printWidth; j++)
48:          {
49:              std::cout << "*";
50:          }
51:          std::cout << "\n";
52:      }
53:  }
54:
55: // Rahmenprogramm für Demo überladener Funktionen
56: int main()
57: {
58:      // Ein Rechteck mit 30,5 initialisieren
59:      Rectangle theRect(30,5);
60:      std::cout << "DrawShape(0,0,true)...\n";
61:      theRect.DrawShape(0,0,true);
62:      std::cout <<"DrawShape(40,2)...\n";
63:      theRect.DrawShape(40,2);
64:      return 0;
65: }
```

```
DrawShape(0,0,true)...
******************************
******************************
******************************
******************************
******************************

DrawShape(40,2)...
****************************************
****************************************
```

Listing 13.2 ersetzt die überladene Funktion `DrawShape()` durch eine einzelne Funktion mit Standardparametern. Die Funktion wird in den Zeilen 11 und 12 deklariert und übernimmt drei Parameter. Die ersten beiden, `aWidth` und `aHeight`, sind vom Typ `int`, der dritte, `UseCurrentValue`, ist vom Typ `bool` (`true` oder `false`) mit dem Standardwert `false`.

Die Implementierung für diese Funktion beginnt in Zeile 30. Die `if`-Anweisung ab Zeile 34 wertet den dritten Parameter, `UseCurrentValue`, aus. Ist er `true`, erhalten die lokalen Variablen `printWidth` und `printHeight` die Werte der Elementvariablen `itsWidth` bzw. `itsHeight`.

Ist `UseCurrentValue` gleich `false`, weil es sich entweder um den Standardwert `false` handelt oder der Benutzer diesen Wert so festgelegt hat, übernimmt die Funktion für `printWidth` und `printHeight` die beiden ersten Parameter.

Wenn `UseCurrentValue` gleich `true` ist, werden die Werte der beiden anderen Parameter ignoriert.

13.3 Standardwerte oder überladene Funktionen?

Die Programme der Listings 13.1 und 13.2 realisieren die gleiche Aufgabe, wobei aber die überladenen Funktionen in Listing 13.1 einfacher zu verstehen und zu handhaben sind. Wenn man außerdem eine dritte Variation benötigt – wenn vielleicht der Benutzer entweder die Breite oder die Höhe, jedoch nicht beide Werte, bereitstellen möchte – kann man die überladenen Funktionen leicht erweitern. Mit Standardwerten ist schnell die Grenze des sinnvoll Machbaren bei neuen Varianten erreicht.

Anhand der folgenden Punkte können Sie entscheiden, ob überladene Funktionen oder Standardwerte im konkreten Fall besser geeignet sind:

Überladene Funktionen empfehlen sich, wenn

- es keinen vernünftigen Standardwert gibt,
- unterschiedliche Algorithmen erforderlich sind,
- variierende Typen in der Parameterliste erforderlich sind.

13.4 Konstruktoren überladen

Wie alle Elementfunktionen lassen sich auch Konstruktoren überladen, was sie sehr leistungsfähig und flexibel macht.

Nehmen wir zum Beispiel ein `Rectangle`-Objekt mit zwei Konstruktoren an: Der erste übernimmt eine Länge und eine Breite und erstellt ein Rechteck dieser Größe. Der zweite übernimmt keine Werte und erzeugt ein Rechteck mit Standardgröße. Der Compiler wählt den richtigen Konstruktor genauso aus

wie bei überladenen Funktionen: anhand der Anzahl und des Typs der Parameter.

Konstruktoren kann man überladen, Destruktoren dagegen *nicht*. Per Definition haben Destruktoren immer genau die gleiche Signatur: den Namen der Klasse, dem eine Tilde (~) vorangeht, und keine Parameter.

13.5 Objekte initialisieren

Bis jetzt haben wir immer die Elementvariablen von Objekten im Rumpf des Konstruktors eingerichtet. Konstruktoren werden allerdings in zwei Stufen erzeugt: zuerst im Initialisierungsteil und dann im Rumpf des Konstruktors.

Die meisten Variablen lassen sich in beiden Stufen festlegen, entweder durch Initialisierung im Initialisierungsteil oder durch Zuweisung im Rumpf des Konstruktors. Eine saubere und oft auch effizientere Lösung besteht in der Initialisierung der Elementvariablen im Initialisierungsteil. Das folgende Beispiel zeigt, wie man Elementvariablen initialisiert:

```
CAT():        // Name und Parameter des Konstruktors
itsAge(5),    // Initialisierungsliste
itsWeight(8)
{ }           // Rumpf des Konstruktors
```

Nach der schließenden Klammer der Parameterliste des Konstruktors steht ein Doppelpunkt. Dann schreiben Sie den Namen der Elementvariablen und ein Klammernpaar. In die Klammern kommt der Ausdruck zur Initialisierung dieser Elementvariablen. Gibt es mehrere Initialisierungen, sind diese jeweils durch Komma zu trennen.

Konstanten ändern sich nicht!

Denken Sie daran, dass Referenzen und Konstanten initialisiert werden *müssen* und keine Zuweisungen erlauben. Wenn Referenzen oder Konstanten als Datenelemente vorkommen, muss man sie wie hier gezeigt initialisieren.

Weiter vorn wurde darauf hingewiesen, dass es effizienter ist, Elementvariablen zu initialisieren statt sie zuzuweisen. Um dies zu verstehen, muss man sich zunächst mit dem Kopierkonstruktor auseinandersetzen.

13.6 Der Kopierkonstruktor

Neben der Bereitstellung eines Standardkonstruktors und -destruktors liefert der Compiler auch einen Standardkopierkonstruktor. Der Aufruf des Kopierkonstruktors erfolgt jedes Mal, wenn die Kopie eines Objekts angelegt wird.

Übergibt man ein Objekt als Wert, entweder als Parameter oder als Rückgabewert einer Funktion, legt die Funktion eine temporäre Kopie des Objekts an. Handelt es sich um ein benutzerdefiniertes Objekt, wird der Kopierkonstruktor der Klasse aufgerufen.

Alle Kopierkonstruktoren übernehmen einen Parameter: eine Referenz auf ein Objekt derselben Klasse. Es empfiehlt sich, diese Referenz als konstant zu deklarieren, da der Konstruktor das übergebene Objekt nicht ändern muss. Zum Beispiel:

```
CAT(const CAT & theCat);
```

Hier übernimmt der `CAT`-Konstruktor eine konstante Referenz auf ein existierendes `CAT`-Objekt. Ziel des Kopierkonstruktors ist das Anlegen einer Kopie von `theCat`.

Der Standardkopierkonstruktor kopiert einfach jede Elementvariable von dem als Parameter übergebenen Objekt in die Elementvariablen des neuen Objekts. Man spricht hier von einer elementweisen (oder flachen) Kopie. Obwohl das bei den meisten Elementvariablen durchaus funktioniert, klappt das bei Elementvariablen, die Zeiger auf Objekte im Heap sind, schon nicht mehr wie beabsichtigt.

Eine *flache* oder *elementweise* Kopie kopiert die genauen Werte der Elementvariablen des einen Objekts in ein anderes Objekt. Zeiger in beiden Objekten verweisen letztendlich auf denselben Speicher. Eine *tiefe* Kopie überträgt andererseits die auf dem Heap reservierten Werte in neu zugewiesenen Speicher.

Wenn die `CAT`-Klasse eine Elementvariable `itsAge` enthält, die auf einen Integer im Heap zeigt, kopiert der Standardkonstruktor die übergebene Elementvariable `itsAge` von `CAT` in die neue Elementvariable `itsAge` von `CAT`. Die beiden Objekte zeigen dann auf denselben Speicher, wie es Abbildung 13.1 verdeutlicht.

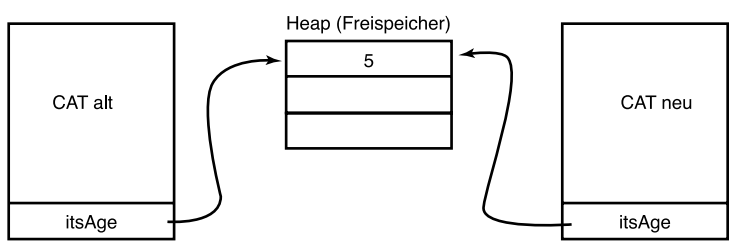


Abb. 13.1: Arbeiten mit dem Standardkopierkonstruktor

Das führt zur Katastrophe, wenn eines der beiden CAT-Objekte den Gültigkeitsbereich verliert. Der dann aufgerufene Destruktor versucht nämlich, den zugewiesenen Speicher aufzuräumen.

Nehmen wir im Beispiel an, dass das originale CAT-Objekt den Gültigkeitsbereich verliert. Der Destruktor dieses Objekts gibt den zugewiesenen Speicher frei. Die Kopie zeigt aber weiterhin auf diesen Speicherbereich, und wenn man auf diesen Speicher zugreift, stürzt das Programm ab. Abbildung 13.2 zeigt diesen Problemfall.

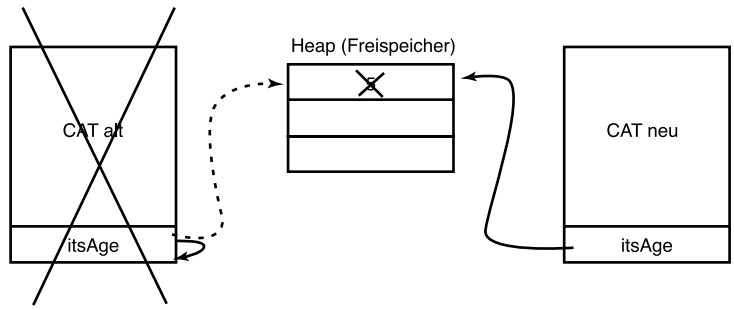


Abb. 13.2: Einen wilden Zeiger erzeugen

Die Lösung besteht darin, einen eigenen Kopierkonstruktor zu definieren und den Speicher wie in der Kopie erforderlich zuzuweisen. Anschließend kann man die alten Werte in den neuen Speicher kopieren. Diesen Vorgang bezeichnet man als *tiefe Kopie*. Listing 13.3 zeigt ein Programm, das nach diesem Verfahren arbeitet.

Listing 13.3: copyconstructors.cpp – Kopierkonstruktoren

```
// Listing 13.3
// Kopierkonstruktoren
#include <iostream>

class CAT
{
public:
    CAT();                          // Standardkonstruktor
    CAT (const CAT &);              // Kopierkonstruktor
    ~CAT();                         // Destruktor
    int GetAge() const { return *itsAge; }
    int GetWeight() const { return *itsWeight; }
    void SetAge(int age) { *itsAge = age; }

private:
    int *itsAge;
    int *itsWeight;
};

```

```
20: CAT::CAT()
21: {
22:     itsAge = new int;
23:     itsWeight = new int;
24:     *itsAge = 5;
25:     *itsWeight = 9;
26: }
27:
28: CAT::CAT(const CAT & rhs)
29: {
30:     itsAge = new int;
31:     itsWeight = new int;
32:     *itsAge = rhs.GetAge();
33:     *itsWeight = rhs.GetWeight();
34: }
35:
36: CAT::~CAT()
37: {
38:     delete itsAge;
39:     itsAge = 0;
40:     delete itsWeight;
41:     itsWeight = 0;
42: }
43:
44: int main()
45: {
46:     CAT frisky;
47:     std::cout << "Alter von Frisky: " << frisky.GetAge() <<
"\n";
48:     std::cout << "Alter von Frisky auf 6 setzen...\n";
49:     frisky.SetAge(6);
50:     std::cout << "Boots von Frisky erzeugen\n";
51:     CAT boots(frisky);
52:     std::cout << "Alter von Frisky: " << frisky.GetAge() <<
"\n";
53:     std::cout << "Alter von Boots: " << boots.GetAge() << "\n";
54:     std::cout << "Alter von Frisky auf 7 setzen...\n";
55:     frisky.SetAge(7);
56:     std::cout << "Alter von Frisky: " << frisky.GetAge() <<
"\n";
57:     std::cout << "Alter von Boots: " << boots.GetAge() << "\n";
58:     return 0;
59: }
```

```
Alter von Frisky: 5
Alter von Frisky auf 6 setzen...
Boots von Frisky erzeugen
Alter von Frisky: 6
Alter von Boots:  6
Alter von Frisky auf 7 setzen...
Alter von Frisky: 7
Alter von Boots: 6
```

Die Zeilen 5 bis 18 deklarieren die Klasse `CAT`. In Zeile 8 steht die Deklaration eines Standardkonstruktors, in Zeile 9 die Deklaration eines Kopierkonstruktors.

Das Programm deklariert in den Zeilen 16 und 17 zwei Elementvariablen als Zeiger auf einen Integer. Normalerweise gibt es kaum einen Grund, dass eine Klasse Elementvariablen vom Typ `int` als Zeiger speichert. Hier aber soll dies verdeutlichen, wie man Elementvariablen auf dem Heap verwaltet.

Der Standardkonstruktor in den Zeilen 20 bis 26 reserviert auf dem Heap Platz für zwei `int`-Variablen und weist ihnen dann Werte zu.

Der Kopierkonstruktor beginnt in Zeile 28. Der Parameter ist wie bei einem Kopierkonstruktor üblich mit `rhs` benannt, was für *right-hand side* – zur rechten Seite – steht. Bei den Zuweisungen in den Zeilen 32 und 33 steht das als Parameter übergebene Objekt auf der rechten Seite des Gleichheitszeichens. Die Funktionsweise stellt sich wie folgt dar:

– Die Zeilen 30 und 31 reservieren Speicher auf dem Heap. Dann überträgt das Programm die Werte aus dem existierenden `CAT`-Objekt in die neuen Speicherstellen (Zeilen 32 und 33).

– Der Parameter `rhs` ist ein `CAT`-Objekt, dessen Übergabe an den Kopierkonstruktor als konstante Referenz erfolgt. Die Elementfunktion `rhs.GetAge()` gibt den Wert aus dem Speicher zurück, auf den die Elementvariable `itsAge` von `rhs` zeigt. Als `CAT`-Objekt verfügt `rhs` über alle Elementvariablen wie jedes andere `CAT`-Objekt.

– Beim Aufruf des Kopierkonstruktors zum Erzeugen eines neuen `CAT`-Objekts wird ein vorhandenes `CAT`-Objekt als Parameter übergeben.

Abbildung 13.3 zeigt die Abläufe. Die Werte, auf die das existierende `CAT`-Objekt verweist, werden in den für das neue `CAT`-Objekt zugewiesenen Speicher kopiert.

Zeile 46 erzeugt ein CAT-Objekt namens frisky. Zeile 47 gibt das Alter von frisky aus und setzt es dann in Zeile 49 auf den Wert 6. Die Anweisung in Zeile 51 erzeugt das neue CAT-Objekt boots mit dem Kopierkonstruktor und übergibt dabei frisky. Hätte man frisky als Parameter an eine Funktion übergeben, würde der Compiler den gleichen Aufruf des Kopierkonstruktors ausführen.

Die Zeilen 52 und 53 geben das Alter beider CAT-Objekte aus. Wie erwartet hat boots das Alter von frisky (nämlich 6) und nicht den Standardwert von 5. Zeile 55 setzt das Alter von frisky auf 7, und Zeile 56 gibt erneut das Alter aus. Dieses Mal ist das Alter von frisky gleich 7, während das Alter von boots bei 6 bleibt. Damit ist nachgewiesen, dass sich die Objekte in separaten Speicherbereichen befinden.

Wenn die CAT-Objekte ihren Gültigkeitsbereich verlieren, findet automatisch der Aufruf ihrer Destruktoren statt. Die Implementierung des CAT-Destruktors ist in den Zeilen 38 bis 44 zu finden. Der Aufruf von delete auf den beiden Zeigern itsAge und itsWeight gibt den zugewiesenen Speicher an den Heap zurück. Aus Sicherheitsgründen ist eine erneute Zuweisung von NULL an beide Zeiger vorgesehen.

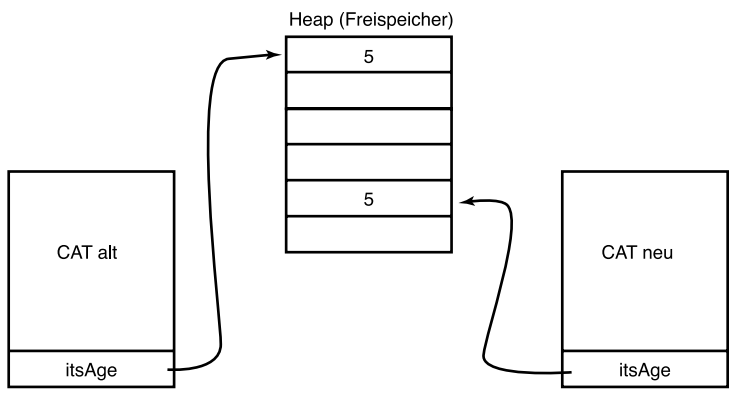


Abb. 13.3: Darstellung einer tiefen Kopie

13.7 Fragen und Antworten

F: Warum verwendet man überhaupt Standardwerte, wenn man eine Funktion überladen kann?

A: Es ist einfacher, nur eine statt zwei Funktionen zu verwalten. Oftmals ist eine Funktion mit Standardparametern auch verständlicher und man muss sich nicht mit zwei verschiedenen Funktionsrümpfen auseinander setzen.

Darüber hinaus bildet die Aktualisierung nur einer der Funktionen bei vergessener Aktualisierung der zweiten Funktion eine beliebte Fehlerquelle.

F: Warum verwendet man nicht immer Standardwerte, wenn man die Probleme mit überladenen Funktionen kennt?

A: Überladene Funktionen bieten Möglichkeiten, die bei Standardvariablen nicht verfügbar sind, beispielsweise die Variation der Parameterliste nach dem Typ statt nur nach der Anzahl.

F: Wie entscheidet man beim Schreiben eines Klassenkonstruktors, was im Initialisierungsabschnitt und was im Rumpf des Konstruktors stehen muss?

A: Als Faustregel sollte man so viel wie möglich in der Initialisierungsphase erledigen – das heißt, alle Elementvariablen an dieser Stelle mit einem Anfangswert versehen. Bestimmte Dinge, wie Berechnungen und Ausgabeanweisungen, muss man im Rumpf des Konstruktors unterbringen.

F: Kann eine überladene Funktion einen Standardparameter haben?

A: Ja. Es gibt keinen Grund, auf die Kombination dieser leistungsfähigen Merkmale zu verzichten. Die überladenen Funktionen (eine oder auch mehrere) können jeweils eigene Standardwerte haben, die den üblichen Regeln für Standardvariablen in einer beliebigen Funktion entsprechen.

13.8 Workshop

Nachdem Sie sich tiefgründiger mit Funktionen beschäftigt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu diesem Thema zu festigen.

13.8.1 Kontrollfragen

1. Woher weiß der Compiler bei überladenen Elementfunktionen, welche Version er aufrufen soll?
2. Kann man überladene Elementfunktionen mit Standardwerten verwenden?
3. Warum überlädt man einen Konstruktor?
4. Was bewirkt der Kopierkonstruktor?

13.8.2 Übungen

1. Modifizieren Sie Listing 13.1 (`overloadfunctions.cpp`) und erstellen Sie eine weitere `DrawShape`-Version mit zwei Integer-Parametern, die Standardwerte übernehmen. Lässt sich dieses Programm kompilieren und ausführen?
2. Modifizieren Sie Listing 13.3 (`copyconstructors.cpp`), um das Alter von `boots` zu ändern, nachdem das Alter von `frisky` geändert wurde. Wirkt sich die Altersänderung von `boots` auf `frisky` aus? Sie haben bereits gesehen, dass eine Altersänderung bei `frisky` nicht das Alter von `boots` beeinflusst.
3. Was würde beim Programm nach Listing 13.3 (`copyconstructors.cpp`) passieren (und insbesondere nicht passieren), wenn man den Standarddestruktor anstelle des hier angegebenen Destruktors verwendet?

13.8.3 Antworten zu den Kontrollfragen

1. Anhand der Anzahl und der Typen der Argumente.
2. Selbstverständlich, solange sich die Anzahl und die Typen der Argumente unter den verschiedenen Funktionsversionen eindeutig unterscheiden (weil der Compiler anhand dieser Angaben die richtige Version auswählt).
3. Konstruktoren überlädt man, um Flexibilität für das Erstellen eines Objekts zu bieten. So kann man einen Konstruktor ohne Parameter bereitstellen, einen anderen mit Parametern und noch andere Konstruktoren mit unterschiedlichen Arten oder Sätzen von Parametern.
4. Ein Programm ruft den Kopierkonstruktor jedes Mal auf, wenn die Kopie eines Objekts zu erstellen ist – entweder explizit, um ein Objekt von einem bereits vorhandenen zu erstellen, oder bei der Übergabe als Wert an eine aufgerufene Funktion. Der Compiler stellt einen Standardkopierkonstruktor bereit, der sich aber insbesondere bei komplexen Objekten oftmals nicht so verhält, wie eigentlich erwartet.

KAPITEL 14

jetzt lerne ich

Überladen von Operatoren

In dieser Stunde lernen Sie,

- wie man Elementfunktionen überlädt,
- wie man den Zuweisungsoperator zur Speicherverwaltung überlädt,
- wie man Funktionen schreibt, um Klassen mit dynamisch zugewiesenen Variablen zu unterstützen.

14.1 Operatoren überladen

C++ verfügt über eine Reihe vordefinierter Typen, beispielsweise `int`, `float` oder `char`. Zu jedem dieser Typen gehören verschiedene vordefinierte Operatoren wie Addition (+) und Multiplikation (*). In C++ können Sie diese Operatoren auch in eigene Klassen aufnehmen.

Listing 14.1 erzeugt die neue Klasse `Counter`, um das Überladen von Operatoren umfassend untersuchen zu können. Ein `Counter`-Objekt realisiert Zählvorgänge in Schleifen und anderen Konstruktionen, wo man eine Zahl inkrementieren, dekrementieren oder in ähnlicher Weise behandeln muss.

Listing 14.1: counter.cpp – Die Klasse Counter

```
1: // Listing 14.1
2: // Die Klasse Counter
3: #include <iostream>
4:
5: class Counter
6: {
7: public:
```

```
 8:     Counter();
 9:     ~Counter(){}
10:     int GetItsVal()const { return itsVal; }
11:     void SetItsVal(int x) {itsVal = x; }
12:
13: private:
14:     int itsVal;
15: };
16:
17: Counter::Counter():
18: itsVal(0)
19: {}
20:
21: int main()
22: {
23:     Counter i;
24:     std::cout << "Der Wert von i: " << i.GetItsVal()
25:         << std::endl;
26:     return 0;
27: }
```

```
Wert von i: 0
```

Die in den Zeilen 5 bis 15 definierte Klasse ist eigentlich recht nutzlos. Die einzige Elementvariable ist ein `int`. Der in Zeile 8 deklarierte und in Zeile 19 implementierte Standardkonstruktor initialisiert die Elementvariable `itsVal` mit `0`.

Im Gegensatz zu einem echten, vordefinierten, »Vollblut«-`int` lässt sich das `Counter`-Objekt weder inkrementieren noch dekrementieren, addieren, zuweisen oder anderweitig manipulieren. Noch dazu gestaltet sich die Ausgabe seines Wertes wesentlich schwieriger!

14.1.1 Eine Inkrement-Funktion schreiben

Das Überladen von Operatoren stellt einen großen Teil der Funktionalität wieder her, die benutzerdefinierten Klassen wie `Counter` verwehrt bleibt. Implementiert man einen Operator für eine Klasse, spricht man vom *Überladen* dieses Operators. Listing 14.2 zeigt, wie man den Inkrement-Operator überlädt.

Listing 14.2: overloadincrement.cpp – Den Inkrement-Operator überladen

```
 1: // Listing 14.2
 2: // Den Inkrement-Operator überladen
 3: #include <iostream>
 4:
 5: class Counter
 6: {
 7: public:
 8:     Counter();
 9:     ~Counter(){}
10:     int GetItsVal()const { return itsVal; }
11:     void SetItsVal(int x) {itsVal = x; }
12:     void Increment() { ++itsVal; }
13:     const Counter& operator++ ();
14:
15: private:
16:     int itsVal;
17: };
18:
19: Counter::Counter():
20: itsVal(0)
21: {}
22:
23: const Counter& Counter::operator++()
24: {
25:     ++itsVal;
26:     return *this;
27: }
28:
29: int main()
30: {
31:     Counter i;
32:     std::cout << "Wert von i: " << i.GetItsVal()
33:         << std::endl;
34:     i.Increment();
35:     std::cout << "Wert von i: " << i.GetItsVal()
36:         << std::endl;
37:     ++i;
38:     std::cout << "Wert von i: " << i.GetItsVal()
39:         << std::endl;
40:     Counter a = ++i;
41:     std::cout << "Wert von a: " << a.GetItsVal();
42:     std::cout << " und von i: " << i.GetItsVal() << std::endl;
43:     return 0;
44: }
```

```
Wert von i: 0
Wert von i: 1
Wert von i: 2
Wert von a: 3 und von i: 3
```

Den Aufruf des Inkrement-Operators

```
++i;
```

in Zeile 37 interpretiert der Compiler als Aufruf der Implementierung von `operator++` in den Zeilen 23 bis 27, die ihre Elementvariable `itsValue` inkrementiert und dann den Zeiger `this` dereferenziert, um das aktuelle Objekt zurückzugeben. Damit steht ein `Counter`-Objekt bereit, das sich an `a` zuweisen lässt. Falls ein derartiges `Counter`-Objekt Speicher reserviert, sollte man den Kopierkonstruktor redefinieren. Im vorliegenden Fall genügt der Standardkopierkonstruktor.

Der zurückgegebene Wert ist eine `Counter`-Referenz. Damit vermeidet man die Erzeugung eines zusätzlichen temporären Objekts. Die Referenz ist konstant – die Funktion, die diesen `Counter` verwendet, soll nämlich nicht den Wert ändern.

14.1.2 Den Postfix-Operator überladen

Was wäre nun, wenn Sie den Inkrement-Operator in der Postfix-Version überladen möchten? Hier hat der Compiler ein Problem. Wie lässt sich zwischen Präfix und Postfix unterscheiden? Per Konvention stellt man eine Integer-Variable als Parameter der Operator-Deklaration bereit. Der Wert des Parameters wird ignoriert – er dient nur als Flag (Signal), dass es sich um den Postfix-Operator handelt.

14.1.3 Unterschied zwischen Präfix und Postfix

Bevor man den Postfix-Operator schreiben kann, muss man den Unterschied zum Präfix-Operator kennen. Präfix heißt »inkrementiere und hole dann«, während Postfix »hole und inkrementiere dann« bedeutet.

Demnach kann der Präfix-Operator einfach den Wert inkrementieren und dann das Objekt selbst zurückgeben, während der Postfix-Operator den Wert zurückgeben muss, der *vor* der Inkrementierung vorhanden war. Dazu ist ein temporäres Objekt zu erzeugen. Es nimmt den Originalwert auf, wenn man den Wert des Originalobjekts inkrementiert. Da der Postfix-Operator nach dem *Originalwert* fragt und nicht nach dem inkrementierten Wert, gibt man das temporäre Objekt zurück.

Untersuchen wir das noch einmal

Schreibt man

```
a = x++;
```

und hatte x den Wert 5, erhält a nach dieser Anweisung den Wert 5 und x den Wert 6. Zunächst wird der Wert aus x geholt und an a zugewiesen. Daran schließt sich die Inkrementierung von x an. Wenn x ein Objekt ist, muss der Postfix-Operator den Originalwert (5) in einem temporären Objekt aufbewahren, den Wert von x auf 6 inkrementieren und dann das temporäre Objekt zurückgeben, um dessen Wert an a zuzuweisen.

Da das temporäre Objekt bei Rückkehr der Funktion den Gültigkeitsbereich verliert, ist es als Wert und nicht als Referenz zurückzugeben.

Listing 14.3 demonstriert die Verwendung der Präfix- und Postfix-Operatoren.

Listing 14.3: preandpostfix.cpp – Präfix- und Postfix-Operatoren

```
 1: // Listing 14.3
 2: // Gibt den dereferenzierten this-Zeiger zurück
 3: #include <iostream>
 4:
 5: class Counter
 6: {
 7: public:
 8:      Counter();
 9:      ~Counter(){}
10:      int GetItsVal()const { return itsVal; }
11:      void SetItsVal(int x) {itsVal = x; }
12:      const Counter& operator++ ();      // Präfix
13:      const Counter operator++ (int);    // Postfix
14: private:
15:      int itsVal;
16: };
17:
18: Counter::Counter():
19: itsVal(0)
20: {}
21:
22: const Counter& Counter::operator++()    // Präfix
23: {
24:      ++itsVal;
25:      return *this;
26: }
27:
```

```
28: const Counter Counter::operator++(int)  // Postfix
29: {
30:      Counter temp(*this);
31:      ++itsVal;
32:      return temp;
33: }
34:
35: int main()
36: {
37:      Counter i;
38:      std::cout << "Der Wert von i: " << i.GetItsVal()
39:           << std::endl;
40:      i++;
41:      std::cout << "Der Wert von i: " << i.GetItsVal()
42:           << std::endl;
43:      ++i;
44:      std::cout << "Der Wert von i: " << i.GetItsVal()
45:           << std::endl;
46:      Counter a = ++i;
47:      std::cout << "Der Wert von a: " << a.GetItsVal();
48:      std::cout << " und von i: " << i.GetItsVal() << std::endl;
49:      a = i++;
50:      std::cout << "Der Wert von a: " << a.GetItsVal();
51:      std::cout << " und von i: " << i.GetItsVal() << std::endl;
52:      return 0;
53: }
```

```
Wert von i: 0
Wert von i: 1
Wert von i: 2
Wert von a: 3 und von i: 3
Wert von a: 3 und von i: 4
```

Die Deklaration des Postfix-Operators steht in Zeile 13 und die Implementierung in den Zeilen 29 bis 34. Beachten Sie, dass der Aufruf des Präfix-Operators in Zeile 44 keine als Flag dienende Integer-Zahl (x) enthält, sondern die normale Syntax verwendet. Der Postfix-Operator kennzeichnet mit einem Flag-Wert (x), dass es sich um Postfix und nicht um Präfix handelt. Der Flag-Wert (x) selbst wird allerdings nicht weiter benötigt.

14.1.4 Der Operator operator+

Der Inkrement-Operator ist ein *unärer* Operator, wirkt also nur auf einen Term. Dagegen ist der Additionsoperator (+) ein binärer Operator, da er zwei Terme (oder Operanden) verknüpft. Wie implementiert man nun das Überladen des Additionsoperators für `Counter`?

Man muss in der Lage sein, zwei Counter-Variablen zu deklarieren und sie dann zu addieren, wie es folgendes Beispiel zeigt:

```
Counter varOne, varTwo, varThree;
VarThree = VarOne + VarTwo;
```

Auch hier kann man zuerst eine Funktion Add() schreiben, die ein Counter-Objekt als Argument übernimmt, die Werte addiert und dann ein Counter-Objekt mit dem Ergebnis zurückgibt. Listing 14.4 zeigt diese Lösung.

Listing 14.4: addfunction.cpp – Die Funktion Add()

```
 1: // Listing 14.4
 2: // Die Funktion Add()
 3: #include <iostream>
 4:
 5: class Counter
 6: {
 7: public:
 8:     Counter();
 9:     Counter(int initialValue);
10:     ~Counter(){}
11:     int GetItsVal()const { return itsVal; }
12:     void SetItsVal(int x) {itsVal = x; }
13:     Counter Add(const Counter &);
14:
15: private:
16:     int itsVal;
17:
18: };
19:
20: Counter::Counter(int initialValue):
21: itsVal(initialValue)
22: {}
23:
24: Counter::Counter():
25: itsVal(0)
26: {}
27:
28: Counter Counter::Add(const Counter & rhs)
29: {
30:     return Counter(itsVal+ rhs.GetItsVal());
31: }
32:
33: int main()
34: {
35:     Counter varOne(2), varTwo(4), varThree;
36:     varThree = varOne.Add(varTwo);
37:     std::cout << "varOne: " << varOne.GetItsVal()<< std::endl;
38:     std::cout << "varTwo: " << varTwo.GetItsVal() << std::endl;
39:     std::cout << "varThree: " << varThree.GetItsVal()
```

```
40:         << std::endl;
41:     return 0;
42: }
```

```
varOne: 2
varTwo: 4
varThree: 6
```

Die Deklaration der Funktion `Add()` steht in Zeile 13. Die Funktion übernimmt eine konstante `Counter`-Referenz. Diese stellt den Wert dar, der auf das aktuelle Objekt zu addieren ist. Die Funktion gibt ein `Counter`-Objekt zurück. Dieses Ergebnis ist der linken Seite der Anweisung wie in Zeile 36 zuzuweisen. Das heißt, `varOne` ist das Objekt, `varTwo` ist der Parameter an die Funktion `Add()` und das Ergebnis weist man an `varThree` zu.

Um `varThree` zu erzeugen, ohne einen Wert dafür zu initialisieren, ist ein Standardkonstruktor erforderlich. Der Standardkonstruktor initialisiert `itsVal` mit `0`, wie es aus den Zeilen 24 bis 26 ersichtlich ist. Da `varOne` und `varTwo` mit einem Wert ungleich Null zu initialisieren sind, wurde ein weiterer Konstruktor erzeugt (Zeilen 20 bis 22). Eine andere Lösung für dieses Problem wäre die Bereitstellung des Standardwertes `0` für den in Zeile 9 deklarierten Konstruktor.

14.1.5 Den operator+ überladen

Die Funktion `Add()` selbst ist in den Zeilen 28 bis 31 von Listing 14.4 zu sehen. Sie funktioniert zwar, ist aber in ungewöhnlicher Weise einzusetzen. Überlädt man `operator+`, lässt sich die Klasse `Counter` eher so verwenden, wie man es bei anderen Operatoren gewohnt ist. Listing 14.5 zeigt diese Lösung.

Listing 14.5: plusoperator.cpp – Der operator+

```
 1: // Listing 14.5
 2: // operator+ überladen
 3: #include <iostream>
 4:
 5: class Counter
 6: {
 7: public:
 8:     Counter();
 9:     Counter(int initialValue);
10:     ~Counter(){}
11:     int GetItsVal()const { return itsVal; }
12:     void SetItsVal(int x) {itsVal = x; }
13:     Counter operator+ (const Counter &);
14: private:
15:     int itsVal;
```

```
16: };
17:
18: Counter::Counter(int initialValue):
19: itsVal(initialValue)
20: {}
21:
22: Counter::Counter():
23: itsVal(0)
24: {}
25:
26: Counter Counter::operator+ (const Counter & rhs)
27: {
28:     return Counter(itsVal + rhs.GetItsVal());
29: }
30:
31: int main()
32: {
33:     Counter varOne(2), varTwo(4), varThree;
34:     varThree = varOne + varTwo;
35:     std::cout << "varOne: " << varOne.GetItsVal()<< std::endl;
36:     std::cout << "varTwo: " << varTwo.GetItsVal() << std::endl;
37:     std::cout << "varThree: " << varThree.GetItsVal()
38:         << std::endl;
39:     return 0;
40: }
```

```
varOne: 2
varTwo: 4
varThree: 6
```

Der Compiler interpretiert den Additionsoperator in Zeile 34

```
varThree = varOne + varTwo;
```

so, als hätte man

```
varThree = varOne.operator+(varTwo);
```

geschrieben, und ruft die in Zeile 13 deklarierte und in den Zeilen 26 bis 29 definierte Methode `operator+` auf.

Vergleichen Sie das mit der Deklaration und der Definition der Funktion `Add()` im vorherigen Listing – sie sind nahezu identisch. Die Syntax unterscheidet sich allerdings grundlegend. Gegenüber der Anweisung

```
varThree = varOne.Add(varTwo);
```

ist die folgende Formulierung gängiger:

```
varThree = varOne + varTwo;
```

Durch diese kleine Änderung lässt sich das Programm einfacher anwenden und besser verstehen. Sie wissen nun, dass man Operatoren überladen und explizite Funktionsaufrufe ersetzen kann.

14.1.6 Einschränkungen beim Überladen von Operatoren

Operatoren von vordefinierten Typen (wie zum Beispiel `int`) lassen sich nicht überladen. Man kann weder die Rangfolge noch die Art – unär oder binär – des Operators ändern. Es lassen sich auch keine neuen Operatoren daraus aufbauen. Beispielsweise ist es nicht möglich, den Operator ** als Exponentialoperator (»Potenz«-Operator) zu deklarieren.

14.1.7 Was überlädt man?

Das Überladen von Operatoren ist einer der Aspekte von C++, den neue Programmierer zu häufig und missbräuchlich anwenden. Es ist zwar verlockend, neue und interessante Einsatzfälle für die ungewöhnlicheren Operatoren auszuprobieren, dies führt aber beständig zu Code, der verwirrend und schwer zu lesen ist.

Es mag amüsant sein, den +-Operator zur Subtraktion und den *-Operator zur Addition zu »überreden«. Ein professioneller Programmierer ist aber über derartige Späße erhaben. Die größere Gefahr liegt in der zwar gut gemeinten aber unüblichen Verwendung eines Operators – zum Beispiel + für die Verkettung einer Zeichenfolge oder / für die Teilung eines Strings. Auch wenn man gut beraten ist, auf diese Anwendungsfälle vorbereitet zu sein, man sollte trotzdem Vorsicht walten lassen. Rufen wir uns das Ziel beim Überladen von Operatoren ins Bewusstsein: die Erhöhung der Brauchbarkeit und Verständlichkeit.

14.1.8 Der operator =

Wie Sie wissen, stellt der Compiler einen Standardkonstruktor, einen Destruktor und einen Kopierkonstruktor bereit. Als vierte und letzte Funktion bietet der Compiler den Zuweisungsoperator (`operator=()`), wenn man keinen eigenen Operator spezifiziert.

Der Aufruf dieses Operators erfolgt bei der Zuweisung eines Objekts. Dazu folgendes Beispiel:

```
CAT catOne(5,7);
CAT catTwo(3,4);
// ... hier steht anderer Code
catTwo = catOne;
```

Diese Anweisungen erzeugen `catOne` und die Parameter an den Konstruktor initialisieren `itsAge` mit `5` und `itsWeight` mit `7`. Dann wird `catTwo` erzeugt und mit den Werten `3` und `4` initialisiert. Schließlich erscheint der Zuweisungsoperator (`operator=`):

```
catTwo = catOne
```

Diese Anweisung weist den Werten `itsAge` und `itsWeight` von `catTwo` die Werte von `catOne` zu. Nach Ausführung dieser Anweisung hat also `catTwo.itsAge` den Wert `5` und `catTwo.itsWeight` den Wert `7`.

Beachten Sie, dass in diesem Fall kein Aufruf des Kopierkonstruktors erfolgt. Das Objekt `catTwo` existiert bereits, man muss es nicht konstruieren. Der Compiler ist intelligent genug, um den Zuweisungsoperator aufzurufen.

In Stunde 13 haben Sie den Unterschied zwischen einer flachen – oder elementweisen – Kopie einerseits und einer tiefen – oder vollständigen – Kopie andererseits kennen gelernt. Eine flache Kopie kopiert einfach die Elemente und beide Objekte zeigen schließlich auf denselben Bereich im Heap. Eine tiefe Kopie reserviert einen neuen Speicherbereich. Sehen Sie sich dazu gegebenenfalls noch einmal Abbildung 13.1 an.

Das gleiche Problem mit dem Kopierkonstruktor tritt auch hier bei der Zuweisung zutage. Zudem ist Folgendes zu beachten: Das Objekt `catTwo` existiert bereits und hat Speicher reserviert. Diesen Speicher muss man löschen, wenn man Speicherlücken vermeiden möchte. Bei der Implementierung des Zuweisungsoperators ist daher zuerst der Speicher zu löschen, der seinen Zeigern zugewiesen ist. Was passiert aber, wenn man `catTwo` an sich selbst wie folgt zuweist:

```
catTwo = catTwo;
```

Kaum jemand schreibt so etwas absichtlich, doch das Programm muss diesen Fall behandeln können. Derartige Anweisungen können nämlich auch zufällig entstehen, wenn referenzierte und dereferenzierte Zeiger die Tatsache verdecken, dass die Zuweisung des Objekts auf sich selbst vorliegt.

Wenn man dieses Problem nicht umsichtig behandelt, löscht `catTwo` die eigene Speicherzuweisung. Steht dann das Kopieren von der rechten Seite der Zuweisung an die linke an, haben wir ein Problem: Der Speicher ist nicht mehr vorhanden.

Zur Absicherung muss der Zuweisungsoperator prüfen, ob auf der rechten Seite das Objekt selbst steht. Dazu untersucht er den Zeiger `this`. Listing 14.6 zeigt eine Klasse mit einem Zuweisungsoperator.

Listing 14.6: assignoperator.cpp – Ein Zuweisungsoperator

```
 1: // Listing 14.6
 2: // Kopierkonstruktoren
 3: #include <iostream>
 4:
 5: class CAT
 6: {
 7: public:
 8:     CAT(); // Standardkonstruktor
 9:     // Aus Platzgründen auf Kopierkonstruktor und Destruktor
       // verzichtet!
10:     int GetAge() const { return *itsAge; }
11:     int GetWeight() const { return *itsWeight; }
12:     void SetAge(int age) { *itsAge = age; }
13:     CAT operator=(const CAT &);
14:
15: private:
16:     int *itsAge;
17:     int *itsWeight;
18: };
19:
20: CAT::CAT()
21: {
22:     itsAge = new int;
23:     itsWeight = new int;
24:     *itsAge = 5;
25:     *itsWeight = 9;
26: }
27:
28:
29: CAT CAT::operator=(const CAT & rhs)
30: {
31:     if (this == &rhs)
32:         return *this;
33:     delete itsAge;
34:     delete itsWeight;
35:     itsAge = new int;
36:     itsWeight = new int;
37:     *itsAge = rhs.GetAge();
38:     *itsWeight = rhs.GetWeight();
39:     return *this;
40: }
41:
42:
43: int main()
44: {
45:     CAT frisky;
46:     std::cout << "Alter von Frisky: " << frisky.GetAge()
47:         << std::endl;
```

```
48:     std::cout << "Alter von Frisky auf 6 setzen...\n";
49:     frisky.SetAge(6);
50:     CAT whiskers;
51:     std::cout << "Alter von Whiskers: " << whiskers.GetAge()
52:         << std::endl;
53:     std::cout << "Frisky auf Whiskers kopieren...\n";
54:     whiskers = frisky;
55:     std::cout << "Alter von Whiskers: " << whiskers.GetAge()
56:         << std::endl;
57:     return 0;
58: }
```

```
Alter von Frisky: 5
Alter von Frisky auf 6 setzen...
Alter von Whiskers: 5
Frisky auf Whiskers kopieren...
Alter von Whiskers: 6
```

Listing 14.6 enthält die bekannte `CAT`-Klasse, verzichtet aber aus Platzgründen auf den Kopierkonstruktor und den Destruktor. In Zeile 13 steht die Deklaration des Zuweisungsoperators, in den Zeilen 29 bis 40 die Definition.

Der Test in Zeile 31 prüft, ob das aktuelle Objekt (das heißt, das auf der linken Seite der Zuweisung stehende `CAT`-Objekt) dasselbe ist, wie das zuzuweisende `CAT`-Objekt. Dazu vergleicht man die Adresse von `rhs` mit der im Zeiger `this` gespeicherten Adresse.

Den Gleichheitsoperator (`==`) kann man natürlich ebenfalls überladen und damit selbst festlegen, was Gleichheit bei Objekten zu bedeuten hat.

In den Zeilen 33 bis 36 werden die Elementvariablen gelöscht und dann auf dem Heap neu angelegt. Das ist zwar nicht zwingend erforderlich, gehört aber zum guten Programmierstil und vermeidet Speicherlücken, wenn Sie mit Objekten variabler Länge arbeiten, die ihren Zuweisungsoperator nicht überladen.

14.2 Umwandlungsoperatoren

Was passiert, wenn man eine Variable eines vordefinierten Typs wie etwa `int` oder `unsigned short` einem Objekt einer benutzerdefinierten Klasse zuweist? Listing 14.7 bedient sich wieder der `Counter`-Klasse und versucht, eine Variable vom Typ `int` an ein `Counter`-Objekt zuzuweisen.

Seien Sie nicht überrascht!

Listing 14.7 lässt sich nicht kompilieren! Die Fehlermeldung im Analyseabschnitt erläutert, warum.

Listing 14.7: counterconvert.cpp – Versuch, einem Counter-Objekt einen int zuzuweisen

```
 1: // Listing 14.7
 2: // Dieser Code lässt sich nicht kompilieren!
 3: #include <iostream>
 4:
 5: class Counter
 6: {
 7: public:
 8:     Counter();
 9:     ~Counter(){}
10:     int GetItsVal()const { return itsVal; }
11:     void SetItsVal(int x) {itsVal = x; }
12: private:
13:     int itsVal;
14: };
15:
16: Counter::Counter():
17: itsVal(0)
18: {}
19:
20: int main()
21: {
22:     int theShort = 5;
23:     Counter theCtr = theShort;
24:     std::cout << "theCtr: " << theCtr.GetItsVal()
25:         << std::endl;
26:     return 0;
27: }
```

```
"counterconvert.cpp": E2034 Konvertierung von 'int' nach 'Counter'
nicht möglich in Funktion main() in Zeile 23
```

Die in den Zeilen 5 bis 14 deklarierte Klasse `Counter` hat nur einen Standardkonstruktor und deklariert keine besondere Methode für die Umwandlung eines `int` in ein `Counter`-Objekt, sodass Zeile 23 einen Compiler-Fehler produziert. Der Compiler kann nicht erkennen, dass der Wert einer angegebenen `int`-Variablen an die Elementvariable `itsVal` zuzuweisen ist, sofern man das nicht ausdrücklich spezifiziert.

Zu diesem Zweck erzeugt die korrigierte Lösung in Listing 14.8 einen Umwandlungsoperator: einen Konstruktor, der einen `int` übernimmt und ein `Counter`-Objekt produziert.

Listing 14.8: counterconvert 2.cpp – Einen int in ein Counter-Objekt konvertieren

```
 1: // Listing 14.8
 2: // Konstruktor als Umwandlungsoperator
 3: #include <iostream>
 4:
 5: class Counter
 6: {
 7: public:
 8:     Counter();
 9:     Counter(int val);
10:     ~Counter(){}
11:     int GetItsVal()const { return itsVal; }
12:     void SetItsVal(int x) {itsVal = x; }
13: private:
14:     int itsVal;
15: };
16:
17: Counter::Counter():
18: itsVal(0)
19: {}
20:
21: Counter::Counter(int val):   // Konstruktor überladen
22: itsVal(val)
23: {}
24:
25: int main()
26: {
27:     int theShort = 5;
28:     Counter theCtr = theShort;
29:     std::cout << "theCtr: " << theCtr.GetItsVal() << std::endl;
30:     return 0;
31: }
```

```
theCtr: 5
```

Als wesentliche Änderung wird in Zeile 9 der Konstruktor überladen, um einen `int` zu übernehmen. Die Implementierung des Konstruktors, der ein `Counter`-Objekt aus einem `int` erzeugt, steht in den Zeilen 21 bis 23.

Mit diesen Angaben kann der Compiler den Konstruktor – der einen `int` als Argument übernimmt – aufrufen. Sehen Sie sich nun an, was bei der umgekehrten Zuweisung wie im folgenden Beispiel passiert:

```
1:  Counter theCtr(5);
2:  int theShort = theCtr;
3:  cout << "theShort : " << theShort  << endl;
```

Auch hier erhält man einen Compiler-Fehler. Obwohl der Compiler jetzt weiß, wie man ein `Counter`-Objekt aus einem `int` erzeugt, bleibt ihm der umgekehrte Vorgang weiterhin ein Rätsel.

14.2.1 Der Operator int()

Für diese und ähnliche Probleme bietet C++ die Umwandlungsoperatoren, die man in eine Klasse aufnehmen kann. Damit lässt sich in einer Klasse festlegen, wie implizite Konvertierungen in vordefinierte Typen auszuführen sind. Listing 14.9 verdeutlicht dies. Es sei allerdings darauf hingewiesen, dass Umwandlungsoperatoren keinen Rückgabewert spezifizieren, obwohl sie tatsächlich einen konvertierten Wert zurückgeben.

Listing 14.9: counterconvertint.cpp – Konvertieren von Counter nach int()

```
 1: // Listing 14.9
 2: // Umwandlungsoperator
 3: #include <iostream>
 4:
 5: class Counter
 6: {
 7: public:
 8:     Counter();
 9:     Counter(int val);
10:     ~Counter(){}
11:     int GetItsVal()const { return itsVal; }
12:     void SetItsVal(int x) {itsVal = x; }
13:     operator unsigned short();
14: private:
15:     int itsVal;
16: };
17:
18: Counter::Counter():
19: itsVal(0)
20: {}
21:
22: Counter::Counter(int val):
23: itsVal(val)
24: {}
25:
26: Counter::operator unsigned short ()
27: {
```

```
28:     return ( int (itsVal) );
29: }
30:
31: int main()
32: {
33:     Counter ctr(5);
34:     int theShort = ctr;
35:     std::cout << "theShort: " << theShort << std::endl;
36:     return 0;
37: }
```

```
theShort: 5
```

Zeile 13 deklariert den Umwandlungsoperator. Beachten Sie, dass er keinen Rückgabewert hat. Die Implementierung dieser Funktion steht in den Zeilen 26 bis 29. Zeile 28 gibt den Wert von `itsVal` konvertiert in einen `int` zurück.

Der Compiler weiß jetzt, wie `int`-Variablen in `Counter`-Objekte und umgekehrt umzuwandeln sind, und man kann sie ohne weiteres einander zuweisen.

14.3 Fragen und Antworten

F: Warum überlädt man eigentlich einen Operator, wenn man auch einfach eine Elementmethode erzeugen kann?

A: Es ist einfacher, Operatoren zu überladen, wenn ihr Verhalten ohne weiteres erkennbar ist. In der Klasse lässt sich damit die Funktionalität der vordefinierten Typen nachbilden.

F: Worin besteht der Unterschied zwischen dem Kopierkonstruktor und dem Zuweisungsoperator?

A: Der Kopierkonstruktor erzeugt ein *neues* Objekt mit denselben Werten wie ein existierendes Objekt. Der Zuweisungsoperator ändert ein *existierendes* Objekt, sodass es dieselben Werte wie das andere Objekt annimmt.

F: Was passiert mit dem `int`, den man in Postfix-Operatoren verwendet?

A: Nichts. Dieser `int` wird lediglich als Flag (Signal) für das Überladen der Postfix- und Präfix-Operatoren und im Übrigen nicht weiter verwendet.

14.4 Workshop

Nachdem Sie sich mit überladenen Operatoren beschäftigt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu diesem Thema zu vertiefen.

14.4.1 Kontrollfragen

1. Warum kann man keine komplett neuen Operatoren wie zum Beispiel ** als Potenzoperator erzeugen?
2. Warum redefiniert man nicht einfach das Verhalten eines vorhandenen Operators wie ^ vollständig neu in der Bedeutung Potenzierung?
3. Warum unterscheidet sich die Syntax für das Überladen bei Präfix- und Postfix-Inkrement/Dekrement?
4. Was bewirken Umwandlungsoperatoren?

14.4.2 Übungen

1. Modifizieren Sie Listing 14.6 (`assignoperator.cpp`), um den Gleichheitsoperator (==) zu überladen. Vergleichen Sie mit diesem Operator das Alter von `frisky` und `whiskers`.
2. Modifizieren Sie Listing 14.3 (`preandpostfix.cpp`), um `itsVal` zu dekrementieren, selbst wenn der überladene Operator ein Inkrement realisiert. Wenn der Code funktioniert, lassen Sie ihn für eine Stunde oder einen Tag ruhen. Sehen Sie sich das Programm dann erneut an. Ist es Ihnen immer noch ohne weiteres klar, wie die Operatoren arbeiten? Oder sind Sie sich nicht ganz sicher, wenn Sie das Programm ausführen? Das dürfte deutlich machen, warum man mit überladenen Operatoren keinen Unfug treiben sollte.
3. Überladen Sie in Listing 14.5 (`plusoperator.cpp`) auch den Minusoperator, um eine einfache Subtraktion zu realisieren. Führen Sie einige Beispiele aus. Spielt die Reihenfolge der Objekte in der Anweisung eine Rolle?

14.4.3 Antworten zu den Kontrollfragen

1. Weil das eine Änderung des Compilers notwendig machen würde – da ** nicht zur Sprache gehört, kann der Compiler nichts damit anfangen.
2. Es ist möglich und manchmal auch sinnvoll. Allerdings kann es andere Programmierer verwirren (und selbst Sie, wenn Sie sich das Programm

längere Zeit nicht angesehen haben). Die Sprache verhindert dieses Vorgehen nicht, nur sollten Sie darauf achten, sich nicht selbst das Leben schwer zu machen.

3. Weil das Verhalten vollkommen anders ist. Bei der Präfix-Notation erhält man den Wert, nachdem der Operator angewandt wurde. Bei der Postfix-Notation erhält man den Originalwert und erst dann findet die Operation statt. Der Code sollte dieses Verhalten nachbilden. Sicherlich ist das nicht zwingend erforderlich, doch wenn Sie den Hass Ihrer Kollegen nicht auf sich ziehen wollen, sollten Sie diesem Muster folgen.
4. Mit ihnen konvertiert man den Objekttyp in einen nativen Typ. In dieser Lektion haben wir ein Zählerobjekt betrachtet und das Ergebnis als einfache Ganzzahl ausgegeben. Für diesen Zweck haben wir Umwandlungsoperatoren eingesetzt.

KAPITEL 15

jetzt lerne ich

Arrays

In dieser Stunde lernen Sie,

- was Arrays sind und wie man sie deklariert,
- was Strings sind und wie man sie mit Zeichen-Arrays erstellt,
- welche Beziehung zwischen Arrays und Zeigern besteht,
- wie man Zeigerarithmetik bei Arrays anwendet.

15.1 Was ist ein Array?

Ein *Array* (Datenfeld) ist eine Sammlung von Objekten, die alle den gleichen Typ haben. Man kann sich ein Array als Folge von Speicherstellen vorstellen. Die einzelnen Speicherstellen bezeichnet man als Elemente des Arrays.

Man deklariert ein Array, indem man den Typ, gefolgt vom Namen des Arrays und dem Index schreibt. Der *Index* bezeichnet die Anzahl der Elemente im Array und wird in eckige Klammern eingeschlossen.

Zum Beispiel deklariert

```
long LongArray[25];
```

ein Array namens `LongArray` mit 25 Zahlen des Typs `long`. Der Compiler reserviert bei dieser Deklaration einen Speicherbereich für die Aufnahme aller 25 Elemente. Da jede Zahl vom Typ `long` genau 4 Byte erfordert, reserviert diese Deklaration 100 zusammenhängende Bytes im Speicher wie es Abbildung 15.1 verdeutlicht.

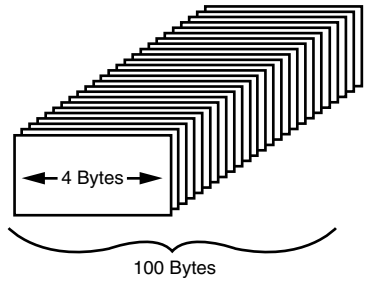


Abb. 15.1: Ein Array deklarieren

15.2 Array-Elemente

Auf die Elemente des Arrays greift man über einen Offset bezüglich des Array-Namens zu. Den Offset schreibt man nach dem Array-Namen in eckigen Klammern. Die Zählung der Array-Elemente beginnt bei `0`. Das erste Element des Arrays `arrayName` heißt demzufolge `arrayName[0]`. Im obigen Beispiel steht `LongArray[0]` für das erste Array-Element, `LongArray[1]` für das zweite usw.

Diese Zählweise ist genau zu beachten. Das Array `SomeArray[3]` enthält drei Elemente: `SomeArray[0]`, `SomeArray[1]` und `SomeArray[2]`. Allgemein ausgedrückt verfügt `SomeArray[n]` über `n` Elemente, die mit `SomeArray[0]` bis `SomeArray[n-1]` nummeriert sind.

Die Elemente in `LongArray[25]` sind demnach von `LongArray[0]` bis `LongArray[24]` nummeriert. Listing 15.1 zeigt, wie man ein Array mit fünf Ganzzahlen deklariert und jedes Element mit einem Wert füllt.

Listing 15.1: intarray.cpp – Ein Array für Integer-Zahlen

```
// Listing 15.1 - Arrays
#include <iostream>

int main()
{
    int myArray[5];
    for (int i=0; i<5; i++)  // 0-4
    {
        std::cout << "Wert fuer myArray[" << i << "]: ";
        std::cin >> myArray[i];
    }
    for (int i = 0; i<5; i++)
        std::cout << i << ": " << myArray[i] << "\n";
    return 0;
}
```

```
Wert fuer myArray[0]:  3
Wert fuer myArray[1]:  6
Wert fuer myArray[2]:  9
Wert fuer myArray[3]:  12
Wert fuer myArray[4]:  15

0: 3
1: 6
2: 9
3: 12
4: 15
```

Zeile 6 deklariert das Array `myArray`, das fünf Integer-Variablen aufnimmt. Zeile 7 richtet eine Schleife ein, die von 0 bis 4 zählt und damit den Indexbereich für ein 5-elementiges Array überstreicht. Der Benutzer gibt einen Wert ein, den das Programm im richtigen Element des Arrays speichert.

Der erste Wert kommt nach `myArray[0]`, der zweite nach `myArray[1]` usw. Die zweite `for`-Schleife gibt alle Werte auf dem Bildschirm aus.

In C++ beginnen Arrays mit dem Index 0

Die Indizierung von Arrays beginnt bei 0 und nicht bei 1. Hier liegt die Ursache für viele Fehler, die gerade Neueinsteigern in C++ passieren. Man sollte immer daran denken, dass ein Array mit 10 Elementen von `ArrayName[0]` bis `ArrayName[9]` nummeriert wird und es kein Element `ArrayName[10]` gibt. Wenn man die Schleifenbedingungen in Listing 15.1 in den Zeilen 7 und 12 mit `<=5` statt `<5` kodiert, überschreitet das Programm die Array-Grenzen.

15.3 Array-Grenzen überschreiten

Der Compiler berechnet anhand der Größe jedes Elements und dem angegebenen Index, wohin ein bestimmter Wert in ein Array zu schreiben ist. Angenommen, es ist ein Wert in `LongArray[5]` – also im sechsten Element – einzutragen. Der Compiler multipliziert den Offset – 5 – mit der Größe des Einzelelements – in diesem Beispiel 4. Es ergibt sich eine Verschiebung (Offset) von 20 Bytes bezüglich des Array-Beginns. An diese Stelle schreibt das Programm den Wert.

Wenn man zum Beispiel in das Element `LongArray[50]` schreiben möchte, ignoriert der Compiler die Tatsache, dass es kein derartiges Element gibt, berechnet die Verschiebung zum ersten Element – 200 Bytes – und schreibt den

Wert über den bereits an dieser Stelle gespeicherten Wert. Dort können allerdings nahezu beliebige Daten stehen, sodass das Überschreiben mit den neuen Daten zu unvorhersehbaren Ergebnissen führen kann. Wenn man Glück hat, stürzt das Programm sofort ab. Andernfalls erhält man wesentlich später im Programm seltsame Ergebnisse und kann nur schwer herausfinden, was hier schief gegangen ist.

Der Compiler verhält sich wie ein Blinder, der die Entfernung von seinem Haus abschreitet. Er beginnt beim ersten Haus, `MainStreet[0]`. Wenn man ihn bittet, zum sechsten Haus in der Main Street zu gehen, denkt er sich: »Ich muss noch weitere fünf Häuser gehen. Jedes Haus ist vier große Schritte lang. Ich gehe also noch weitere 20 Schritte.« Bittet man ihn, zur `MainStreet[100]` zu gehen und die Main Street hat nur 25 Häuser, wird er 400 Schritte zurücklegen. Bevor er allerdings an seinem Ziel ankommt, läuft er vielleicht in einen fahrenden Bus. Passen Sie also auf, wo Sie den Mann hinschicken.

15.4 Fence Post Errors

Es kommt so häufig vor, dass man die Grenzen des Arrays um genau ein Element überschreitet, dass dieser Fehler einen eigenen Namen bekommen hat: *Fence Post Error*, zu deutsch etwa »Zaunpfahlfehler«. Gerade bei schnellen Denkspielen begegnet man der Frage, wie viele Zaunpfähle für einen Zaun von 10 Meter Länge erforderlich sind, wenn man einen Pfahl pro Meter benötigt. Die unüberlegte Antwort lautet oft 10, obwohl es natürlich 11 sind. Abbildung 15.2 verdeutlicht das.

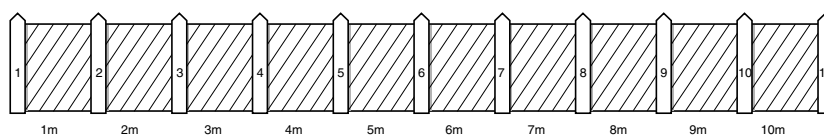


Abb. 15.2: Fence Post Error

Aufgrund dieser Zählweise haben sich Programmierer schon unzählige Male die Haare gerauft. Mit der Zeit geht es aber in Fleisch und Blut über, dass ein 25-elementiges Array nur bis zum Element 24 nummeriert ist und dass man beim Zählen bei 0 beginnen muss. (C++-Programmierer wundern sich oft, warum Häuser keine Etage 0 haben. In der Tat sind manche so schlau, im Fahrstuhl die Taste 4 zu drücken, wenn sie in den fünften Stock möchten.)

15.5 Arrays initialisieren

Bei der Deklaration eines einfachen Arrays mit vordefinierten Typen wie Integer-Zahlen und Zeichen kann man das Array auch gleich initialisieren. Dazu schreibt man nach dem Array-Namen ein Gleichheitszeichen und eine in geschweifte Klammern eingeschlossene Liste mit den durch Komma getrennten Werten. Beispielsweise deklariert

```
int IntegerArray[5] = { 10, 20, 30, 40, 50 };
```

das Array `IntegerArray` mit fünf Ganzzahlen. Das Element `IntegerArray[0]` erhält den Wert `10` zugewiesen, das Element `IntegerArray[1]` den Wert `20` usw.

Wenn man die Größe des Arrays nicht angibt, wird ein Array entsprechend der spezifizierten Initialisierungswerte erzeugt. Schreibt man daher

```
int IntegerArray[] = { 10, 20, 30, 40, 50 };
```

erzeugt man exakt das gleiche Array wie im vorherigen Beispiel.

Muss man die Größe des Arrays ermitteln, kann man sie mit der Funktion `sizeof()`berechnen. Zum Beispiel setzt

```
const int IntegerArrayLength = sizeof(IntegerArray)/
sizeof(IntegerArray[0]);
```

die konstante `int`-Variable `IntegerArrayLength` auf das Ergebnis, das man aus der Division der Gesamtgröße des Arrays durch die Größe eines einzelnen Elements im Array erhält. Der Quotient gibt die Anzahl der Elemente im Array an.

Mehr als die für das Array deklarierten Elemente lassen sich nicht initialisieren. Die Anweisung

```
int IntegerArray[5] = { 10, 20, 30, 40, 50, 60};
```

führt zu einem Compiler-Fehler, da man ein 5-elementiges Array deklariert hat, aber sechs Werte initialisieren will. Dagegen ist die folgende Anweisung zulässig:

```
int IntegerArray[5] = { 10, 20};
```

Obwohl nicht initialisierte Array-Elemente keine garantierten Werte enthalten, werden Gruppierungen praktisch immer auf 0 gesetzt. Wenn man ein Array-Element nicht initialisiert, erhält es automatisch den Wert 0.

15.6 Arrays von Objekten

Jedes Objekt – gleichgültig ob vordefiniert oder benutzerdefiniert – lässt sich in einem Array speichern. Wenn man das Array deklariert, teilt man dem Compiler den zu speichernden Typ und die Anzahl der Objekte mit, für die Platz zu reservieren ist. Der Compiler weiß anhand der Klassendeklaration, wie viel Speicher jedes Objekt belegt. Die Klasse muss über einen Standardkonstruktor ohne Argumente verfügen, damit sich das Objekt bei der Definition des Arrays erzeugen lässt.

Standardkonstruktor

Denken Sie daran, dass der Standardkonstruktor *nicht* unbedingt der Konstruktor ist, den der Compiler bereitstellt, sondern ein Konstruktor ohne Parameter, den entweder der Compiler oder der Programmierer erstellt.

Der Zugriff auf Datenelemente in einem Array mit Objekten läuft in zwei Schritten ab. Man identifiziert das Element des Arrays mit dem Index-Operator ([ ]) und fügt dann den Elementoperator (.) an, um auf die betreffende Elementvariable zuzugreifen. Listing 15.2 zeigt, wie man ein Array mit fünf CAT-Objekten erzeugt.

Listing 15.2: objarray.cpp – Ein Array von Objekten erzeugen

```
// Listing 15.2 - Ein Array von Objekten
#include <iostream>

class CAT
{
public:
    CAT() { itsAge = 1; itsWeight=5; } // Standardkonstruktor
    ~CAT() {}                          // Destruktor
    int GetAge() const { return itsAge; }
    int GetWeight() const { return itsWeight; }
    void SetAge(int age) { itsAge = age; }

private:
    int itsAge;
    int itsWeight;
};

int main()
{
    CAT Litter[5];    // Ein Array von 5 Objekten erzeugen
    int i;
    for (i = 0; i < 5; i++)
        Litter[i].SetAge(2*i +1);

```

```
25:     for (i = 0; i < 5; i++)
26:         std::cout << "Cat #" << i+1<< ": "
27:             << Litter[i].GetAge() << std::endl;
28:     return 0;
29: }
```

```
Katze #1: 1
Katze #2: 3
Katze #3: 5
Katze #4: 7
Katze #5: 9
```

Die Zeilen 4 bis 16 deklarieren die Klasse `CAT`. Diese Klasse muss über einen Standardkonstruktor verfügen, damit sich `CAT`-Objekte in einem Array erzeugen lassen. Denken Sie daran, dass der Compiler keinen Standardkonstruktor bereitstellt, wenn man irgendeinen anderen Konstruktor erzeugt – in diesem Fall müssen Sie einen eigenen Standardkonstruktor erstellen. Zeile 20 reserviert ein Array von 5 `CAT`-Objekten namens `Litter`.

Die erste `for`-Schleife (in den Zeilen 22 und 23) setzt das Alter aller fünf `CAT`-Objekte im Array. Die zweite `for`-Schleife (in den Zeilen 25 bis 27) greift auf alle Elemente des Arrays zu und ruft `GetAge()` auf.

Der Aufruf der `GetAge()`-Methode für die einzelnen `CAT`-Objekte erfolgt über den Namen des Array-Elements, `Litter[i]`, gefolgt vom Punktoperator (`.`) und der Elementfunktion.

15.7 Mehrdimensionale Arrays

Ein (eindimensionales) Array kann man sich als Datenzeile vorstellen. Ein zweidimensionales Array lässt sich als Datentabelle mit Zeilen und Spalten auffassen, wobei die eine Dimension jede Zeile und die zweite Dimension jede Spalte repräsentiert. Ein dreidimensionales Array ähnelt einem Quader: Eine Dimension gibt die Breite an, eine zweite Dimension die Höhe und eine dritte Dimension die Tiefe. Es sind auch Arrays mit mehr als drei Dimensionen möglich, auch wenn das über unsere räumliche Vorstellungskraft hinausgeht.

Wenn man Arrays deklariert, stellt man jede Dimension durch einen entsprechenden Index im Array dar. Ein zweidimensionales Array hat demzufolge zwei Indizes und ein dreidimensionales drei Indizes. Prinzipiell ist die Anzahl der Dimensionen nicht begrenzt, obwohl man in der Praxis kaum mit mehr als zwei Dimensionen arbeitet.

Ein Schachbrett liefert ein gutes Beispiel für ein zweidimensionales Array. Die acht Reihen sind der einen Dimension, die acht Spalten der anderen Dimension zugeordnet. Abbildung 15.3 zeigt dieses Konzept.

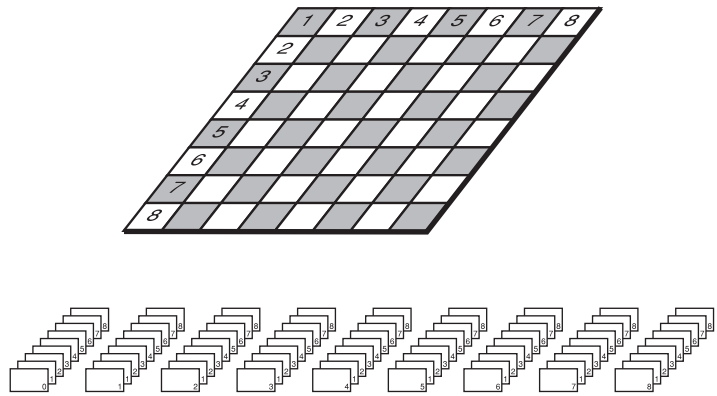


Abb. 15.3: Ein Schachbrett und ein zweidimensionales Array

Als Beispiel sei eine Klasse `SQUARE` gegeben. Ein Array `Board` zur Darstellung des Schachbretts deklariert man dann wie folgt:

```
SQUARE Board[8][8];
```

Die gleichen Daten ließen sich auch in einem eindimensionalen Array für 64 Quadrate unterbringen:

```
SQUARE Board[64];
```

Diese Darstellung lehnt sich allerdings nicht so eng an das reale Objekt wie ein zweidimensionales Array an. Bei Spielbeginn steht der König in der vierten Spalte der ersten Reihe. Da die Zählung bei 0 beginnt, entspricht diese Position

```
Board[0][3];
```

vorausgesetzt, dass der erste Index der Zeile und der zweite Index der Spalte zugeordnet ist. Die Anordnung aller Brettpositionen ist aus Abbildung 15.3 ersichtlich.

Platzbedarf von Arrays

Der Platzbedarf mehrdimensionaler Arrays kann schnell den verfügbaren Speicher überschreiten. Die Gesamtzahl der Elemente ergibt sich aus der Multiplikation der Indizes aller Dimensionen.

15.7.1 Mehrdimensionale Arrays initialisieren

Um mehrdimensionale Arrays zu initialisieren, weist man den Array-Elementen die Liste der Werte der Reihe nach zu, wobei sich jeweils der letzte Index ändert, während der vorhergehende konstant bleibt. Beim Array

```
int theArray[5][3]
```

kommen zum Beispiel die ersten drei Elemente nach `theArray[0]`, die nächsten drei nach `theArray[1]` usw.

Zur Initialisierung des Arrays schreibt man

```
int theArray[5][3] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 }
```

Diese Anweisung lässt sich übersichtlicher darstellen, wenn man die einzelnen Initialisierungen mit geschweiften Klammern gruppiert:

```
int theArray[5][3] = {  {1,2,3},
                        {4,5,6},
                        {7,8,9},
                        {10,11,12},
                        {13,14,15} };
```

Die inneren geschweiften Klammern ignoriert der Compiler – sie dienen lediglich dem Programmierer zur übersichtlichen Gruppierung der Zahlen.

Jeder Wert ist durch ein Komma zu trennen, unabhängig von eventuell vorhandenen Klammern. Die gesamte Initialisierungsliste ist in geschweifte Klammern einzuschließen und muss mit einem Semikolon enden.

Listing 15.3 erzeugt ein zweidimensionales Array. Die erste Dimension ist die Menge der Zahlen 0 bis 4. Im Unterschied zum vorherigen Beispiel werden aber die Werte so gesetzt, dass die zweite Dimension das Doppelte jedes Wertes der ersten Dimension enthält.

Listing 15.3: multidimarray.cpp – Ein mehrdimensionales Array erzeugen

```
 1: // Listing 15.3 - Ein mehrdimensionales Array erzeugen
 2: #include <iostream>
 3:
 4: int main()
 5: {
 6:     int SomeArray[5][2] = { {0,0}, {1,2}, {2,4}, {3,6}, {4,8}};
 7:     for (int i = 0; i<5; i++)
 8:         for (int j=0; j<2; j++)
 9:         {
10:             std::cout << "SomeArray[" << i << "][" << j << "]: ";
11:             std::cout << SomeArray[i][j]<< std::endl;
12:         }
13:     return 0;
14: }
```

```
SomeArray[0][0]: 0
SomeArray[0][1]: 0
SomeArray[1][0]: 1
SomeArray[1][1]: 2
SomeArray[2][0]: 2
SomeArray[2][1]: 4
SomeArray[3][0]: 3
SomeArray[3][1]: 6
SomeArray[4][0]: 4
SomeArray[4][1]: 8
```

Zeile 6 deklariert `SomeArray` als zweidimensionales Array. Die erste Dimension besteht aus fünf, die zweite Dimension aus zwei Ganzzahlen. Damit entsteht eine 5-mal-2-Matrix, wie sie Abbildung 15.4 zeigt.

Die Initialisierung des Arrays erfolgt mit Wertepaaren, obwohl man die Werte genauso gut auch berechnen könnte. Die Zeilen 7 und 8 bilden eine verschachtelte `for`-Schleife. Die äußere `for`-Schleife durchläuft alle Elemente der ersten Dimension. Für jedes dieser Elemente durchläuft die innere `for`-Schleife alle Elemente der zweiten Dimension. Dieser Ablauf dokumentiert sich in der Ausgabe: auf `SomeArray[0][0]` folgt `SomeArray[0][1]`. Jeweils nach dem Inkrementieren der zweiten Dimension erhöht die äußere `for`-Schleife den Index für die erste Dimension um 1. Dann beginnt die zweite Dimension wieder von vorn (bei 0).

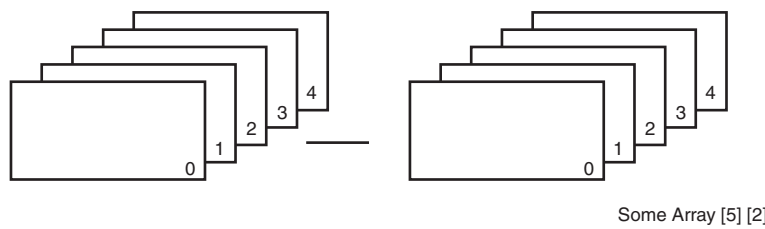


Abb. 15.4: Eine 5-mal-2-Matrix

15.8 Ein Wort zum Speicher

Mit der Deklaration eines Arrays teilt man dem Compiler die genaue Anzahl der im Array zu speichernden Objekte mit. Der Compiler reserviert Speicher für alle Objekte, auch wenn man sie überhaupt nicht verwendet. Bei Arrays, deren Größe von vornherein in etwa bekannt ist, stellt das kein Problem dar. Beispielsweise hat ein Schachbrett genau 64 Felder und Katzen (`CAT`-Objekte) bekommen zwischen 1 und 10 Kätzchen. Wenn man allerdings nur eine vage

Vorstellung von der Anzahl der benötigten Objekte hat, muss man auf ausgefeiltere Datenstrukturen ausweichen.

Dieses Buch geht auf Arrays von Zeigern, Arrays auf dem Heap und verschiedene andere Gruppierungen ein. Stunde 19 beschäftigt sich mit einer flexibleren Datenstruktur, die man als verkettete Liste bezeichnet. Komplexere Datenstrukturen, mit denen sich Probleme bei sehr umfangreichen Datenbeständen lösen lassen, gehören aber nicht zum Themenbereich dieses Buches – hierzu sei auf die Literatur zu Algorithmen und Datenstrukturen verwiesen. Bei der Programmierung zeigen sich vor allem zwei Dinge: Es gibt immer noch mehr zu lernen und es gibt immer noch andere Bücher, in denen man sich kundig machen kann.

15.9 Arrays von Zeigern

Die bisher behandelten Arrays speichern ihre Elemente auf dem Stack. Normalerweise ist der Stack-Speicher streng limitiert, während auf dem Heap wesentlich mehr Platz zur Verfügung steht. Es ist möglich, jedes Objekt auf dem Heap zu deklarieren und dann nur einen Zeiger auf das Objekt im Array abzulegen. Damit reduziert sich die benötigte Stack-Größe drastisch. Listing 15.4 ist eine Neuauflage von Listing 15.2, speichert aber nun alle Objekte auf dem Heap. Als Hinweis für den jetzt möglichen größeren Speicher ist das Array von 5 auf 500 Elemente erweitert und der Name von `Litter` (Körbchen) auf `Family` (Familie) geändert worden.

Listing 15.4: arrayonheap.cpp – Ein Array auf dem Heap speichern

```
 1: // Listing 15.4 - Ein Array von Zeigern auf Objekte
 2: #include <iostream>
 3:
 4: class CAT
 5: {
 6: public:
 7:     CAT() { itsAge = 1; itsWeight=5; } // Standardkonstruktor
 8:     ~CAT() {}                          // Destruktor
 9:     int GetAge() const { return itsAge; }
10:     int GetWeight() const { return itsWeight; }
11:     void SetAge(int age) { itsAge = age; }
12:
13: private:
14:     int itsAge;
15:     int itsWeight;
16: };
17:
18: int main()
19: {
20:     CAT * Family[500];
21:     int i;
```

```
22:     CAT * pCat;
23:     for (i = 0; i < 500; i++)
24:     {
25:         pCat = new CAT;
26:         pCat->SetAge(2*i +1);
27:         Family[i] = pCat;
28:     }
29:
30:     for (i = 0; i < 500; i++)
31:         std::cout << "Katze #" << i+1 << ": "
32:             << Family[i]->GetAge() << std::endl;
33:
34:     for (i = 0; i < 500; i++)
35:     {
36:         delete Family[i];
37:         Family[i] = NULL;
38:     }
39:
40:     return 0;
41: }
```

```
Katze #1: 1
Katze #2: 3
Katze #3: 5
...
Katze #499: 997
Katze #500: 999
```

Das in den Zeilen 4 bis 16 deklarierte `CAT`-Objekt ist mit dem in Listing 15.2 deklarierten `CAT`-Objekt identisch. Das in Zeile 20 deklarierte Array heißt jetzt aber `Family` und ist für die Aufnahme von 500 Zeigern auf `CAT`-Objekte ausgelegt.

Die Initialisierungsschleife (in den Zeilen 23 bis 28) erzeugt neue `CAT`-Objekte auf dem Heap und setzt die Werte für das Alter auf den zweifachen Indexwert plus 1. Das Alter im ersten `CAT`-Objekt enthält demnach den Wert `1`, das zweite den Wert `3`, das dritte den Wert `5` usw. Schließlich legt die Initialisierungsschleife den Zeiger im Array ab.

Da das Array für die Aufnahme von Zeigern deklariert ist, wird im Array der Zeiger – und nicht der dereferenzierte Wert des Zeigers – gespeichert.

Die zweite Schleife (Zeilen 30 bis 32) gibt alle Werte aus. Das Programm greift auf den Zeiger über den Index `Family[i]` zu. Mit der daraus erhaltenen Adresse findet der Zugriff auf die Methode `GetAge()` statt.

Dieses Beispiel speichert das Array Family und alle dazugehörenden Zeiger auf dem Stack, während die 500 erzeugten CAT-Objekte auf dem Heap abgelegt werden. Man ist nun in der Lage, bis zu 500 CAT-Objekte zu erzeugen, verbraucht aber nur so viel Speicher, wie für die tatsächlich angelegten Objekte erforderlich ist. Das Beispiel bestimmt in einer for-Schleife die Anzahl der zu reservierenden Objekte; es könnte aber auch die Anzahl der CAT-Objekte im Speicher lesen und die Speicherreservierung beenden, sobald genügend Platz zugewiesen ist. Verglichen mit der direkten Reservierung eines Arrays für 500 CAT-Objekte wäre der Speicherbedarf dann noch geringer.

Die dritte Schleife löscht die CAT-Objekte vom Heap und setzt alle Elemente von Family auf Null.

15.10 Arrays auf dem Heap

Man kann auch das gesamte Array auf dem Heap unterbringen. Dazu ruft man new auf und verwendet den Index-Operator. Das Ergebnis ist ein Zeiger auf einen Bereich im Heap, der das Array aufnimmt. Beispielsweise deklariert

```
CAT *Family = new CAT[500];
```

Family als Zeiger auf das erste Objekt in einem Array von 500 CAT-Objekten. Mit anderen Worten zeigt Family auf das Element Family[0] bzw. enthält dessen Adresse.

Auf diese Weise kann man über Family mittels Zeigerarithmetik auf jedes Element von Family zugreifen. Beispielsweise lässt sich schreiben:

```
CAT *Family = new CAT[500];
CAT *pCat = Family;              // pCat zeigt auf Family[0]
pCat->SetAge(10);                // setzt Family[0] auf 10
pCat++;                          // geht weiter zu Family[1]
pCat->SetAge(20);                // setzt Family[1] auf 20
```

Dieses Codefragment deklariert ein neues Array von 500 CAT-Objekten und einen Zeiger, der auf den Beginn des Arrays verweist. Über diesen Zeiger ruft man die Funktion SetAge() des ersten CAT-Objekts auf und übergibt den Wert 10. Die vierte Anweisung inkrementiert den Zeiger, der nun auf das nächste CAT-Objekt verweist. Die fünfte Anweisung ruft somit die Methode SetAge() auf dem zweiten CAT-Objekt auf.

15.11 Zeiger auf ein Array und ein Array von Zeigern

Sehen Sie sich die folgenden drei Deklarationen an:

```
1:  CAT   FamilyOne[500];
2:  CAT * FamilyTwo[500];
3:  CAT * FamilyThree = new CAT[500];
```

`FamilyOne` ist ein Array von 500 `CAT`-Objekten, `FamilyTwo` ein Array von 500 Zeigern auf `CAT`-Objekte und `FamilyThree` ein Zeiger auf ein Array mit 500 `CAT`-Objekten.

Die Unterschiede zwischen den drei Codezeilen haben entscheidenden Einfluss auf die Arbeitsweise der drei Arrays. Überraschend ist vielleicht, dass `FamilyThree` eine Variante von `FamilyOne` ist, sich aber grundsätzlich von `FamilyTwo` unterscheidet.

Damit stellt sich die heikle Frage, wie sich Zeiger zu Arrays verhalten. Im dritten Fall ist `FamilyThree` ein Zeiger auf ein Array. Das heißt, die Adresse im Zeiger `FamilyThree` ist die Adresse des ersten Elements in diesem Array. Genau das trifft auch auf `FamilyOne` zu; es ist die eigentliche Adresse für den Zeiger `FamilyThree`.

15.12 Zeiger und Array-Namen

In C++ ist ein Array-Name ein konstanter Zeiger auf das erste Element des Arrays. In der Deklaration

```
CAT Family[50];
```

ist `Family` demzufolge ein Zeiger auf das Element `&Family[0]` – das heißt, die erste Adresse des ersten Array-Elements von `Family`.

Es ist zulässig, Array-Namen als konstante Zeiger – und umgekehrt – zu verwenden. Demzufolge kann man mit `Family + 4` auf die Daten in `Family[4]` zugreifen.

Der Compiler führt die korrekten Berechnungen aus, wenn man Zeiger addiert, inkrementiert oder dekrementiert. Die Adresse für den Ausdruck `Family + 4` liegt nämlich nicht einfach 4 Bytes nach der Adresse von `Family`, sondern 4 Objekte danach. Hat jedes Objekt eine Länge von 4 Bytes, bedeutet `Family + 4` eine Adressverschiebung um 16 Bytes. Handelt es sich zum Beispiel um `CAT`-Objekte mit vier Elementvariablen vom Typ `long` (jeweils 4 Bytes) und zwei Elementvariablen vom Typ `short` (jeweils 2 Bytes), dann beträgt die Länge eines `CAT`-Objekts 20 Bytes und `Family + 4` liegt 80 Bytes nach dem Array-Anfang.

Listing 15.5 zeigt die Deklaration und Verwendung eines Arrays auf dem Heap.

Listing 15.5: newarray.cpp – Ein Array mit new erzeugen

```
// Listing 15.5 - Ein Array auf dem Heap
#include <iostream>

class CAT
{
public:
    CAT() { itsAge = 1; itsWeight=5; }     // Standardkonstruktor
    ~CAT();                                // Destruktor
    int GetAge() const { return itsAge; }
    int GetWeight() const { return itsWeight; }
    void SetAge(int age) { itsAge = age; }

private:
    int itsAge;
    int itsWeight;
};

CAT :: ~CAT()
{
//  std::cout << "Destruktor aufgerufen!\n";
}

int main()
{
    CAT * Family = new CAT[500];
    int i;
    CAT * pCat;
    for (i = 0; i < 500; i++)
    {
        pCat = new CAT;
        pCat->SetAge(2*i +1);
        Family[i] = *pCat;
        delete pCat;
    }

    for (i = 0; i < 500; i++)
        std::cout << "Katze #" << i+1 << ": "
            << Family[i].GetAge() << std::endl;

    delete [] Family;

    return 0;
}
```

```
Katze #1: 1
Katze #2: 3
Katze #3: 5
...
Katze #499: 997
Katze #500: 999
```

Zeile 25 deklariert das Array `Family` für 500 `CAT`-Objekte. Der Aufruf `new CAT[500]` erzeugt das gesamte Array auf dem Heap.

Hinter den Kulissen

Aus technischer Sicht deklariert Zeile 25 ein unbenanntes Array auf dem Heap und gibt die Adresse seines ersten Elements an den Zeiger `Family` zurück. Es ist üblich, diesen Zeiger mit dem Array gleichzusetzen. Wenn man nämlich ein Array (selbst auf dem Stack) deklariert, ist der Array-Name tatsächlich nichts weiter als ein Zeiger auf das erste Element.

Ebenfalls auf den Heap kommt jedes dem Array hinzugefügte `CAT`-Objekt (Zeile 30). Beachten Sie, dass dieses Mal nicht der Zeiger, sondern das Objekt selbst im Array gespeichert wird. Es handelt sich nicht um ein Array von Zeigern auf `CAT`-Objekte – es ist ein Array von `CAT`-Objekten.

Im Allgemeinen kann man `a []` durch `a *` (Indirektionsoperator) ersetzen. Dann verweist `Family[3]` auf dasselbe Objekt wie `*(Family + 3)`.

15.13 Arrays auf dem Heap löschen

`Family` ist ein Zeiger und zwar ein Zeiger auf das neue `CAT`-Array auf dem Heap. Wenn Zeile 32 von Listing 15.5 den Zeiger `pCat` dereferenziert, wird das `CAT`-Objekt selbst im Array gespeichert. (Warum auch nicht? Das Array befindet sich ja auf dem Heap.) Im nächsten Schleifendurchlauf kommt allerdings wieder `pCat` zum Einsatz. Besteht da nicht die Gefahr, dass es jetzt keinen Zeiger auf dieses `CAT`-Objekt mehr gibt und eine Speicherlücke entsteht?

Man hat es hier mit einem großen Problem zu tun, es sei denn, dass durch Löschen von `Family` der gesamte für das Array zugewiesene Speicher zurückgegeben wird. Der Compiler ist intelligent genug, um alle Objekte im Array abzubauen und dessen Speicher an den Heap zurückzugeben.

Davon können Sie sich selbst überzeugen. Ändern Sie in `newarray.cpp` (Listing 15.5) die Größe des Arrays in den Zeilen 25, 28 und 36 von 500 auf 10.

Entfernen Sie außerdem die Kommentarzeichen vor der `cout`-Anweisung in Zeile 20. Erreicht das Programm Zeile 40 und zerstört das Array, wird der Destruktor für jedes `CAT`-Objekt aufgerufen.

Erzeugt man mit `new` ein Element auf dem Heap, verwendet man immer `delete`, um dieses Element zu löschen und den zugeordneten Speicher freizugeben. Wenn man ein Array mit `new <class>[size]` erzeugt, schreibt man `delete[]`, um dieses Array zu löschen und dessen Speicher freizugeben. Die eckigen Klammern signalisieren dem Compiler, dass das gesamte Array zu löschen ist.

Wenn man die Klammern weglässt, wird nur das erste Objekt im Array gelöscht. Sie können das selbst nachprüfen, indem Sie die eckigen Klammern in Zeile 40 von `newarray.cpp` (Listing 15.5) entfernen. Zeile 22 sollte auch hier keine Kommentarzeichen enthalten, damit sich der Destruktor meldet. Beim Programmlauf stellen Sie dann fest, dass nur ein einziges `CAT`-Objekt zerstört wird. Herzlichen Glückwunsch! Gerade haben Sie eine Speicherlücke erzeugt.

Was Sie tun sollten

Denken Sie daran, dass ein Array mit *n* Elementen von 0 bis *n*-1 nummeriert ist.

Verwenden Sie Array-Indizierung mit Zeigern, die auf Arrays verweisen.

Achtung

Schreiben oder lesen Sie nicht über das Ende eines Arrays hinaus.

Verwechseln Sie nicht ein Array von Zeigern mit einem Zeiger auf ein Array.

15.14 char-Arrays

Ein *String* ist eine Folge einzelner Zeichen. Die bisher gezeigten Strings waren ausnahmslos unbenannte String-Konstanten in `cout`-Anweisungen wie zum Beispiel

```
cout << "hello world.\n";
```

Ein String in C++ ist ein Array mit Elementen vom Typ `char` und endet mit einem Null-Zeichen (als `'\0'` geschrieben, um es vom `NULL`-Zeiger zu unterscheiden). Man kann einen String genauso deklarieren und initialisieren wie jedes andere Array. Dazu folgendes Beispiel:

```
char Greeting[] = { 'H', 'e', 'l', 'l', 'o', ' ',
'W','o','r','l','d', '\0' };
```

Das letzte Zeichen, `'\0'`, ist das Null-Zeichen, das viele Funktionen in C++ als Abschlusszeichen eines Strings erkennen. Obwohl diese zeichenweise Lösung funktioniert, ist sie umständlich einzugeben und bietet viel Spielraum für Fehler. C++ erlaubt eine Kurzform für die obige Codezeile:

```
char Greeting[] = "Hello World";
```

Bei dieser Syntax sind zwei Punkte zu beachten:

- Statt der in Apostrophe eingeschlossenen, durch Komma getrennten und von geschweiften Klammern umgebenen Zeichen ist die gesamte Zeichenfolge in Anführungszeichen zu setzen und ohne Kommas und geschweifte Klammern zu schreiben.
- Das Null-Zeichen braucht man nicht hinzuzufügen, da der Compiler dies automatisch erledigt.

Der String `Hello World` hat eine Länge von 12 Bytes: `Hello` mit 5 Bytes, ein Leerzeichen mit 1 Byte, `World` mit 5 Bytes und das abschließende Null-Zeichen mit 1 Byte.

Man kann auch nicht initialisierte Zeichen-Arrays erzeugen. Wie bei allen Arrays darf man nur so viel Zeichen in den Puffer stellen, wie Platz dafür reserviert ist.

Listing 15.6 zeigt die Verwendung eines nicht initialisierten Puffers.

Listing 15.6: arrayfilled.cpp – Ein Array füllen

```
 1: // Listing 15.6 Puffer für Zeichen-Array
 2: #include <iostream>
 3:
 4: int main()
 5: {
 6:     char buffer[80];
 7:     std::cout << "Geben Sie den String ein: ";
 8:     std::cin >> buffer;
 9:     std::cout << "Inhalt des Puffers:  " << buffer << std::endl;
10:     return 0;
11: }
```

```
Geben Sie den String ein: Hello World
Inhalt des Puffers: Hello
```

Zeile 6 deklariert einen Puffer für 80 Zeichen. Diese Größe genügt, um eine Zeichenfolge mit 79 Zeichen und dem abschließenden Null-Zeichen aufzunehmen.

Die Anweisung in Zeile 7 fordert den Benutzer zur Eingabe einer Zeichenfolge auf, die das Programm in Zeile 8 in den Puffer übernimmt. Die Syntax von `cin` schreibt nach der eingegebenen Zeichenfolge ein abschließendes Null-Zeichen in den Puffer.

Das Programm in Listing 15.6 weist zwei Probleme auf. Erstens: Wenn der Benutzer mehr als 79 Zeichen eingibt, schreibt `cin` über das Ende des Puffers hinaus. Zweitens: Wenn der Benutzer ein Leerzeichen eingibt, nimmt `cin` das Ende des Strings an und beendet die Eingabe in den Puffer.

Um diese Probleme zu lösen, muss man die spezielle Methode `get()` auf `cin` aufrufen, die drei Parameter übernimmt:

- den zu füllenden Puffer
- die maximale Anzahl der zu übernehmenden Zeichen
- das Begrenzerzeichen, das die Eingabe abschließt

Der Standardbegrenzer ist `newline` (Zeichen für neue Zeile). Listing 15.7 zeigt dazu ein Beispiel.

Listing 15.7: arrayfilled2.cpp – Ein Array füllen

```
 1: //Listing 15.7 Arbeiten mit cin.get()
 2: #include <iostream>
 3:
 4: int main()
 5: {
 6:     char buffer[80];
 7:     std::cout << "Geben Sie den String ein: ";
 8:     std::cin.get(buffer, 79);       // Bis max. 79 Zeichen oder
bis newline einlesen
 9:     std::cout << "Inhalt des Puffers:  " << buffer << std::endl;
10:     return 0;
11: }
```

```
Geben Sie den String ein: Hello World
Inhalt des Puffers: Hello World
```

Zeile 8 ruft die Methode `get()`von `cin` auf und übergibt den in Zeile 6 deklarierten Puffer als erstes Argument. Das zweite Argument gibt die maximal zu lesenden Zeichen an. In diesem Fall muss dieser Wert gleich 79 sein, um Platz für das abschließende Null-Zeichen zu lassen. Auf die Angabe eines Zeichens für das Ende der Eingabe können wir verzichten, da der Standardwert `newline` unseren Ansprüchen genügt.

15.15 Die Funktionen strcpy() und strncpy()

C++ erbt von C eine Funktionsbibliothek für die Behandlung von Strings. Unter den zahlreichen Funktionen finden sich zwei für das Kopieren eines Strings in einen anderen: `strcpy()` und `strncpy()`. Die Funktion `strcpy()` kopiert den gesamten Inhalt eines Strings in den angegebenen Puffer. Listing 15.8 zeigt ein entsprechendes Beispiel.

Listing 15.8: usingstrcpy.cpp – Die Funktion strcpy()

```
1: // Listing 15.8 - Die Funktion strcpy()
2: #include <iostream>
3: #include <string.h>
4:
5: int main()
6: {
7:     char String1[] = "Keiner lebt fuer sich allein.";
8:     char String2[80];
9:
10:    strcpy(String2,String1);
11:
12:    std::cout << "String1: " << String1 << std::endl;
13:    std::cout << "String2: " << String2 << std::endl;
14:    return 0;
15: }
```

```
String1: Keiner lebt fuer sich allein.
String2: Keiner lebt fuer sich allein.
```

Zeile 3 bindet die Header-Datei `string.h` ein. Diese Datei enthält den Prototyp der Funktion `strcpy()`. Als Parameter übernimmt die Funktion `strcpy()` zwei Zeichen-Arrays – ein Ziel und anschließend eine Quelle. Wenn die Quelle größer als das Ziel ist, schreibt `strcpy()` über das Ende des Puffers hinaus.

Mit der Funktion `strncpy()` aus der Standardbibliothek kann man sich dagegen schützen. Diese Version übernimmt die Maximalzahl der zu kopierenden Zeichen. Die Funktion `strncopy()` kopiert bis zum ersten Null-Zeichen oder bis zu der als Argument übergebenen Maximalzahl von Zeichen in den Zielpuffer.

Listing 15.9 zeigt die Verwendung von `strncpy()`.

Listing 15.9: usingstrncpy.cpp – Die Funktion strncpy()

```
 1: // Listing 15.9 Die Funktion strncpy()
 2: #include <iostream>
 3: #include <string.h>
 4:
 5: int main()
 6: {
 7:     const int MaxLength = 80;
 8:     char String1[] = "Keiner lebt fuer sich allein.";
 9:     char String2[MaxLength+1];
10:
11:     strncpy(String2,String1,MaxLength);
12:     String2[strlen(String1)] = '\0'; // Null-Zeichen anfügen
13:     std::cout << "String1: " << String1 << std::endl;
14:     std::cout << "String2: " << String2 << std::endl;
15:     return 0;
16: }
```

```
String1: Keiner lebt fuer sich allein.
String2: Keiner lebt fuer sich allein.
```

In Zeile 11 wurde der Aufruf von `strcpy()` in `strncpy()` geändert. Hinzugekommen ist ein dritter Parameter: die maximale Anzahl der zu kopierenden Zeichen. Der Puffer `String2` ist für die Aufnahme von `MaxLength+1` Zeichen deklariert. Der zusätzliche Platz ist für das Null-Zeichen vorgesehen, das den String abschließen muss.

15.16 String-Klassen

Zu den meisten C++-Compilern, die dem ANSI/ISO-Standard entsprechen, gehört eine umfangreiche Bibliothek mit Klassen für die Manipulation von Daten. Als Standardkomponente einer Klassenbibliothek ist eine `String`-Klasse vorhanden.

C++ hat zwar den Null-terminierten String und die Funktionsbibliothek inklusive der Funktion `strcpy()` von C geerbt, allerdings fügen sich diese Funktionen nicht in das objektorientierte Konzept ein. Eine `String`-Klasse stellt einen gekapselten Satz von Daten und Funktionen für die Manipulation dieser Daten sowie Zugriffsfunktionen bereit, sodass die Daten selbst gegenüber den Klienten der `String`-Klasse verborgen sind.

Wenn zu Ihrem Compiler noch keine `String`-Klasse gehört (und vielleicht auch dann, wenn sie dazu gehört), sollten Sie sich eine eigene aufbauen.

Zumindest sollte die `String`-Klasse die grundlegenden Beschränkungen von Zeichen-Arrays aufheben. Wie alle Arrays sind auch Zeichen-Arrays statisch.

Damit ist von vornherein ihre Größe festgelegt und sie belegen immer den gleichen Platz im Speicher, auch wenn man ihn überhaupt nicht nutzt. Das Schreiben über das Ende des Arrays hinaus hat verheerende Folgen.

Eine gute `String`-Klasse reserviert nur so viel Speicher wie nötig. Dabei ist aber immer genug Speicher bereitzustellen, um für alle Eventualitäten gerüstet zu sein. Reicht der angeforderte Speicher nicht aus, sollte das Programm nicht einfach abstürzen, sondern eine ordnungsgemäße Fehlerbehandlung ermöglichen.

15.17 Fragen und Antworten

F: Was passiert, wenn ich in das Element 25 eines 24-elementigen Arrays schreibe?

A: Der Schreibvorgang findet in einem außerhalb des Arrays liegenden Speicherbereich statt und kann damit verheerende Folgen im Programm haben.

F: Was ist in einem nicht initialisierten Array-Element gespeichert?

A: Jeweils das, was sich gerade im Speicher befindet. Die Ergebnisse bei der Verwendung dieses Elements ohne vorherige Zuweisung eines Wertes sind nicht vorhersehbar.

F: Kann ich Arrays zusammenfassen?

A: Ja. Bei einfachen Arrays kann man sie mithilfe von Zeigern zu einem neuen, größeren Array kombinieren. Bei Strings kann man mit den integrierten Funktionen wie zum Beispiel `strcat()` arbeiten, um Zeichenfolgen zu verketten.

15.18 Workshop

Nachdem Sie mit Arrays vertraut sind, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu festigen.

15.18.1 Kontrollfragen

1. Was ist der kleinste und größte Array-Index für ein bestimmtes Array?
2. Was passiert, wenn man Daten jenseits der zulässigen Array-Grenzen speichert?

3. Welchen Vorteil bieten Arrays?
4. Wie kann ich mehr über die `String`-Klassen von C++ erfahren?

15.18.2 Übungen

1. Modifizieren Sie Listing 15.4 (`arrayonheap.cpp`), um die Verwendung von `pCat` zu vermeiden.
2. Definieren Sie im ursprünglichen Listing 15.4 (`arrayonheap.cpp`) eine Integer-Konstante `MAXSIZE`. Verwenden Sie diese Konstante an allen Stellen, an denen momentan der konstante Wert `500` steht. Gehen Sie auf einen niedrigeren Wert über und führen Sie das Programm erneut aus. Was passiert, wenn Sie das Schlüsselwort `const` entfernen? Die Array-Zuweisung ist in C++ nicht dynamisch.
3. Was passiert, wenn Sie in Übung 2 den Wert von `MAXSIZE` auf `200` setzen, aber nicht alle Stellen mit dem Wert `500` durch `MAXSIZE` ersetzt haben?
4. Ändern Sie in Zeile 11 von Listing 15.9 (`usingstrncpy.cpp`) die Variable `MaxLength` in den Wert `5`. Kompilieren und starten Sie das Programm. Vergleichen Sie die Ausgabe mit der ursprünglichen Version des Programms. Hier zeigt sich die wahre Leistung von `strncopy()` – die Funktion kann auch weniger als den kompletten String kopieren.

15.18.3 Antworten zu den Kontrollfragen

1. Die Zählung aller Array-Indizes beginnt bei Null. Der letzte Index (das letzte nutzbare Element) ist die Größe des Arrays minus 1. So hat das letzte Element eines mit `[50]` definierten Arrays den Index 49.
2. Das lässt sich nicht auf Anhieb sagen. Wenn Sie Glück haben, greifen Sie auf einen Speicherbereich zu, den das Betriebssystem für sich beansprucht und mit einer Fehlermeldung reagiert. Ungünstiger sieht es aus, wenn sich ein anderer Variablenwert ändert. Solche Fehler lassen sich nur schwer einkreisen und beseitigen.
3. Die Möglichkeit, auf eine Gruppe zusammengehörender Daten unter ein und demselben Variablennamen zuzugreifen.
4. Nutzen Sie die Hilfeeinrichtungen Ihres Compilers. Beim Borland-Compiler wählen Sie HILFE | HILFETHEMEN, erweitern auf der Registerkarte INHALT die Einträge *Referenz* und *C++ RTL* und klicken dann auf `<string>` (nicht auf `<string.h>`, das die C-String-Standardbibliothek verkörpert).

KAPITEL 16

jetzt lerne ich

Vererbung

In dieser Stunde lernen Sie,

- was Vererbung ist,
- wie man eine Klasse von einer anderen ableitet,
- wie man auf die Methoden der Basisklasse von der abgeleiteten Klasse zugreift,
- wie man die Methoden der Basisklasse überschreibt,
- was geschützter Zugriff ist und wie man ihn einsetzt.

16.1 Was ist Vererbung?

Zu den Grundzügen der menschlichen Intelligenz gehören das Auffinden, Erkennen und Herstellen von Beziehungen zwischen Begriffen. Wir konstruieren Hierarchien, Matrizen, Netzwerke und andere Zwischenverbindungen, um die Wechselwirkungen zwischen den Dingen zu erläutern und zu verstehen. C++ fängt diese Abstraktionsvorgänge in Vererbungshierarchien ein.

Was ist ein Hund? Wenn Sie sich Ihr Haustier ansehen, was stellen Sie fest? Ein Biologe sieht eine Vernetzung von miteinander in Wechselwirkung stehenden Organen, ein Physiker sieht Atome und wirkende Kräfte, ein Systematiker sieht einen Vertreter der Familie *Canis familiaris* und meine Mutter sieht vor allem Hundehaare.

Im Moment interessiert uns hier der Standpunkt des Systematikers. Für ihn ist ein Hund ein Säugetier, ein Säugetier ist eine Tierart usw. Systematiker gliedern die Lebewesen in Reich, Abteilung, Stamm, Klasse, Ordnung, Familie, Gattung und Art.

Die Hierarchie des Systematikers richtet eine »ist-ein«-Beziehung ein. Ein Hund ist ein Raubtier. Überall begegnen uns »ist-ein«-Beziehungen: ein Toyota ist ein Auto, dieses wiederum ist ein Fortbewegungsmittel. Ein Pudding ist ein Nachtisch, der wieder ein Nahrungsmittel ist.

Auf diese Weise schaffen wir systematische Kategorien, die von oben nach unten eine zunehmende Spezialisierung aufweisen. Zum Beispiel ist ein Auto eine spezielle Art von Fortbewegungsmittel. Sowohl Autos als auch Busse sind Fortbewegungsmittel. Man unterscheidet sie nach ihren spezifischen Merkmalen – Merkmale, die für Autos typisch sind, und Merkmale, die für Busse typisch sind. Auf der Spezialisierungsstufe »Fortbewegungsmittel« betrachtet man beide als identisch.

16.1.1 Vererbung und Ableitung

Der Begriff *Hund* erbt – das heißt, erhält automatisch – alle Merkmale eines Säugetiers. Von einem Säugetier ist bekannt, dass es sich bewegt und atmet – alle Säugetiere bewegen sich und atmen per Definition. Der Vorstellung vom Hund fügt man nun das Bellen, Schwanzwedeln usw. hinzu. Die Charakteristika eines Hundes sind spezialisiert, während die Charakteristika eines Säugetiers allen Säugetieren gemein sind.

Hunde lassen sich weiter einteilen in beispielsweise Jagdhunde und Terrier und wir können Terrier weiter untergliedern in Yorkshire Terrier, Dandie Dinmont Terrier usw.

Ein Yorkshire Terrier ist eine Art von Terrier, ist demnach eine Art von Hund, demzufolge eine Art von Säugetier, demzufolge eine Art von Tier und demzufolge eine Art der Lebewesen. Abbildung 16.1 stellt diese Hierarchie dar.

C++ versucht, diese Beziehungen durch die Definition von Klassen darzustellen, die sich von einer anderen Klasse ableiten. Die Ableitung ist eine Möglichkeit, eine »ist-ein«-Beziehung auszudrücken. Man leitet eine neue Klasse `Dog` (Hund) von der Klasse `Mammal` (Säugetier) ab. Dabei muss man nicht explizit feststellen, dass sich Hunde bewegen, da sie diese Eigenschaft von `Mammal` erben. Da eine `Dog`-Klasse von einer `Mammal`-Klasse erbt, bewegt sich `Dog` *automatisch*.

Eine Klasse, die eine existierende Klasse um neue Funktionalität erweitert, bezeichnet man als von dieser Originalklasse *abgeleitet*. Die Originalklasse heißt *Basisklasse* der neuen Klasse.

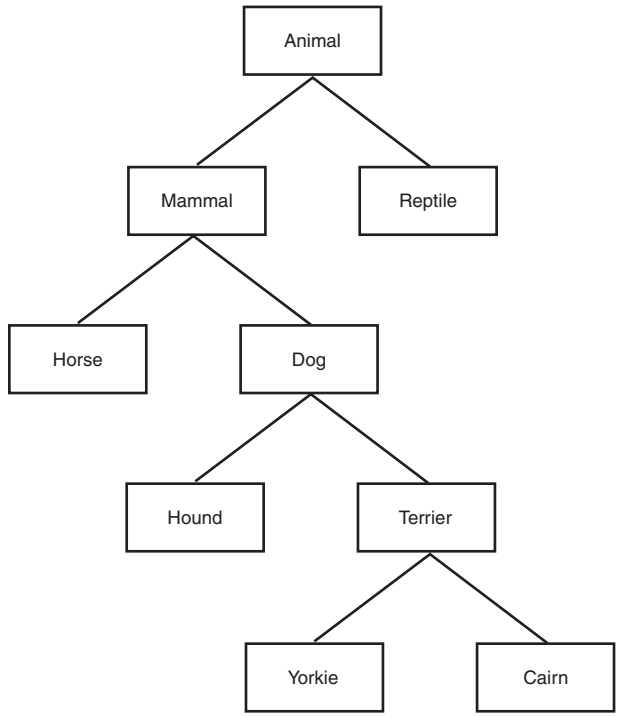


Abb. 16.1: Hierarchie von Tieren

Wenn man die Klasse `Dog` von der Klasse `Mammal` ableitet, dann ist `Mammal` die Basisklasse von `Dog`. Abgeleitete Klassen sind Obermengen ihrer Basisklassen. Genau wie ein Hund der Vorstellung von einem Säugetier bestimmte Merkmale hinzufügt, erweitert die Klasse `Dog` die Klasse `Mammal` um bestimmte Methoden oder Daten.

Normalerweise verfügt eine Basisklasse über mehrere abgeleitete Klassen. Da Hunde, Katzen und Pferde zu den Säugetieren gehören, leiten sich ihre Klassen von der Klasse `Mammal` ab.

16.1.2 Das Tierreich

Um die Behandlung von Ableitung und Vererbung einfacher zu gestalten, konzentriert sich dieses Kapitel auf die Beziehungen zwischen einer Reihe von Klassen, die Tiere darstellen. Nehmen wir an, dass die Simulation eines Bauernhofs als Kinderspiel zu entwickeln ist.

Mit der Zeit schafft man einen ganzen Satz von Tieren auf dem Bauernhof. Dazu gehören Pferde, Kühe, Hunde, Katzen und Schafe. Man erzeugt Methoden für diese Klassen, damit sie sich so verhalten, wie es ein Kind erwartet.

Momentan erzeugen wir aber nur Stub-Methoden mit einer einfachen Ausgabeanweisung.

Eine *Stub*-Routine (zu deutsch »Stummel«-Routine) enthält keinen ausführbaren Code, besteht gewöhnlich aus Kommentaren, die den endgültigen Verwendungszweck erläutern, und zeigt lediglich in der Art eines Platzhalters, wie die Funktion aufzurufen ist. Die Einzelheiten der Funktion schreibt man später, wenn mehr Zeit zur Verfügung steht. Manchmal liegen Stub-Funktionen jahrelang herum in der Hoffnung, dass sich der Programmierer endlich an die Realisierung macht.

Es hindert Sie keiner daran, den in diesem Kapitel vorgestellten Minimalcode so zu erweitern, dass sich die Tiere realistischer verhalten. Wenn man nur mehr Zeit hätte…

16.1.3 Die Syntax der Ableitung

Bei der Deklaration einer Klasse schreibt man nach dem Klassennamen einen Doppelpunkt, den Typ der Ableitung (zum Beispiel `public`) und den Namen der Basisklasse, von der sich die neue Klasse ableitet. Zum Beispiel:

```
class Dog : public Mammal
```

Auf den Typ der Ableitung gehen wir später in diesem Kapitel ein. Momentan verwenden wir immer `public`. Die Basisklasse muss bereits vorher deklariert worden sein, da man sonst einen Compiler-Fehler erhält. Listing 16.1 deklariert eine `Dog`-Klasse, die von einer `Mammal`-Klasse abgeleitet ist.

Listing 16.1: simple-inherit.cpp – Einfache Vererbung

```
// Listing 16.1 Einfache Vererbung
#include <iostream>

enum BREED { YORKIE, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };

class Mammal
{
public:
    // Konstruktoren
    Mammal();
    ~Mammal();

    // Zugriffsfunktionen
    int GetAge() const;
    void SetAge(int);
    int GetWeight() const;
    void SetWeight();

    // Andere Methoden
    void Speak();
    void Sleep();
```

```
22:
23:
24: protected:
25:     int itsAge;
26:     int itsWeight;
27: };
28:
29: class Dog : public Mammal
30: {
31: public:
32:     // Konstruktoren
33:     Dog();
34:     ~Dog();
35:
36:     // Zugriffsfunktionen
37:     BREED GetBreed() const;
38:     void SetBreed(BREED);
39:
40:     // Andere Methoden
41:     // WagTail();
42:     // BegForFood();
43:
44: protected:
45:     BREED itsBreed;
46: };
47:int main()
48:{
49:    return 0;
50:}
```

Dieses Programm liefert keine Ausgaben, da es sich nur um einen Satz von Deklarationen ohne die dazugehörenden Implementierungen handelt. Trotzdem enthält dieses Listing interessante Details und es lässt sich auch kompilieren.

Die Zeilen 6 bis 27 deklarieren die Klasse `Mammal`. In diesem Beispiel leitet sich `Mammal` von keiner anderen Klasse ab. Normalerweise wäre das aber der Fall – das heißt, Säugetiere gehören zur Klasse der Wirbeltiere. In einem C++-Programm kann man nur einen Bruchteil der Informationen darstellen, die man über ein gegebenes Objekt hat. Die Realität ist zu komplex, um sie vollständig wiederzugeben, sodass jede C++-Hierarchie eine etwas willkürliche Darstellung der verfügbaren Daten ist. Die Kunst eines guten Entwurfs besteht in einer einigermaßen wahrheitsgetreuen Widerspiegelung der Realität.

Die Hierarchie muss an irgendeiner Stelle beginnen. Im Beispielprogramm ist das die Klasse `Mammal`. Aufgrund dieser Entscheidung finden wir hier einige Elementvariablen, die vielleicht in einer höheren Basisklasse besser aufgehoben wären. Beispielsweise haben mit Sicherheit alle Tiere ein Alter und ein Gewicht, sodass man bei einer Ableitung der Klasse `Mammal` von `Animal` diese Attribute erben könnte. Im Beispiel erscheinen die Attribute jedoch in der Klasse `Mammal`.

Um das Programm relativ einfach und übersichtlich zu halten, wurden in die Klasse `Mammal` lediglich sechs Methoden aufgenommen – vier Zugriffsmethoden, `Speak` (Sprechen) und `Sleep` (Schlafen).

Aus der Syntax in Zeile 29 geht hervor, dass die Klasse `Dog` von `Mammal` erbt. Jedes `Dog`-Objekt verfügt über drei Elementvariablen: `itsAge`, `itsWeight` und `itsBreed`. Beachten Sie, dass in der Klassendeklaration von `Dog` die Elementvariablen `itsAge` und `itsWeight` nicht aufgeführt sind. `Dog`-Objekte erben diese Variablen sowie alle Methoden von der Klasse `Mammal`. Ausgenommen hiervon sind der Kopieroperator, die Konstruktoren und der Destruktor.

16.2 Private und Protected

Sicherlich haben Sie das neue Zugriffsschlüsselwort `protected` in den Zeilen 24 und 44 von Listing 16.1 bemerkt. Bisher haben wir Klassendaten als `private` deklariert. Für abgeleitete Klassen sind private Elemente allerdings nicht verfügbar. Man könnte zwar `itsAge` und `itsWeight` als `public` deklarieren, das ist aber nicht wünschenswert. Andere Klassen sollen nämlich nicht auf diese Datenelemente direkt zugreifen können.

Man braucht also eine Kennzeichnung, die Folgendes aussagt: »Mache dieses Element sichtbar zu dieser Klasse und zu Klassen, die sich von dieser Klasse ableiten.« Genau das bewirkt das Schlüsselwort `protected` (geschützt). Geschützte Datenelemente und Funktionen sind für abgeleitete Klassen vollständig sichtbar, sonst aber privat.

Im Sinne der Zugriffssteuerung liegt `protected` irgendwo zwischen `public` und `private`. Verglichen mit `private` ist die Steuerung etwas lockerer, jedoch nicht so sehr wie `public`.

Insgesamt gibt es drei Spezifizierer für den Zugriff: `public`, `protected` und `private`. Kommt in einer Funktion ein Objekt einer bestimmten Klasse vor, kann die Funktion auf alle öffentlichen (`public`) Datenelemente und Elementfunktionen dieser Klasse zugreifen. Die Elementfunktionen können wiederum auf alle privaten (`private`) Datenelemente und Funktionen ihrer eigenen Klasse und alle geschützten (`protected`) Datenelemente und Funktionen einer beliebigen Klasse, von der sie sich ableiten, zugreifen.

Demzufolge kann die Funktion `Dog::WagTail` auf die privaten Daten `itsBreed` und auf die geschützten Daten in der Klasse `Mammal` zugreifen.

Selbst wenn sich in der Hierarchie andere Klassen zwischen `Mammal` und `Dog` befinden (beispielsweise `DomesticAnimals`, Haustiere), kann die Klasse `Dog` weiterhin auf die geschützten Elemente von `Mammal` zugreifen. Das setzt allerdings voraus, dass die dazwischen liegenden Klassen mit öffentlicher Vererbung arbeiten.

Listing 16.2 demonstriert, wie man Objekte vom Typ `Dog` erzeugt und auf die Daten und Funktionen dieses Typs zugreift.

Listing 16.2: derived-object.cpp – Ein abgeleitetes Objekt

```
 1: // Listing 16.2 Ein abgeleitetes Objekt verwenden
 2: #include <iostream>
 3:
 4: enum BREED { YORKIE, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
 5:
 6: class Mammal
 7: {
 8: public:
 9:     // Konstruktoren
10:     Mammal():itsAge(2), itsWeight(5){}
11:     ~Mammal(){}
12:
13:     // Zugriffsfunktionen
14:     int GetAge()const { return itsAge; }
15:     void SetAge(int age) { itsAge = age; }
16:     int GetWeight() const { return itsWeight; }
17:     void SetWeight(int weight) { itsWeight = weight; }
18:
19:     // Andere Methoden
20:     void Speak()const { std::cout << "Mammal gibt Laut!\n"; }
21:     void Sleep()const { std::cout << "Psst. Ich schlafe.\n"; }
22:
23:
24: protected:
25:     int itsAge;
26:     int itsWeight;
27: };
28:
29: class Dog : public Mammal
30: {
31: public:
32:     // Konstruktoren
33:     Dog():itsBreed(YORKIE){}
34:     ~Dog(){}
35:
36:     // Zugriffsfunktionen
```

```
 37:     BREED GetBreed() const { return itsBreed; }
 38:     void SetBreed(BREED breed) { itsBreed = breed; }
 39:
 40:     // Andere Methoden
 41:     void WagTail() { std::cout << "Schwanzwedeln...\n"; }
 42:     void BegForFood() { std::cout << "Um Futter betteln...\n"; }
 43:
 44: private:
 45:     BREED itsBreed;
 46: };
 47:
 48: int main()
 49: {
 50:     Dog fido;
 51:     fido.Speak();
 52:     fido.WagTail();
 53:     std::cout << "Fido ist " << fido.GetAge() << " Jahre
alt.\n";
 54:     return 0;
 55: }
```

```
Mammal gib Laut!
Schwanzwedeln...
Fido ist 2 Jahre alt.
```

Die Zeilen 6 bis 27 deklarieren die Klasse `Mammal` (zur Platzersparnis sind alle Funktionen inline definiert). In den Zeilen 29 bis 46 wird die Klasse `Dog` als abgeleitete Klasse von `Mammal` deklariert. Aufgrund dieser Deklarationen verfügen alle `Dog`-Objekte über ein Alter (`age`), ein Gewicht (`weight`) und eine Rasse (`breed`).

Zeile 50 deklariert das `Dog`-Objekt `Fido`, das sowohl alle Attribute eines `Mammal`-Objekts als auch alle Attribute eines `Dog`-Objekts erbt. Daher weiß `Fido`, wie man mit dem Schwanz wedelt (`WagTail`), aber auch, wie man spricht (`Speak`) und schläft (`Sleep`).

16.3 Konstruktoren und Destruktoren

`Dog`-Objekte sind `Mammal`-Objekte. Das ist das Wesen einer »ist-ein«-Beziehung. Beim Erzeugen von `Fido` wird zuerst dessen Konstruktor aufgerufen, der ein `Mammal`-Objekt erzeugt. Dann folgt der Aufruf des `Dog`-Konstruktors, der die Konstruktion des `Dog`-Objekts vervollständigt. Da `Fido` keine Parameter übernimmt, wird in diesem Fall der Standardkonstruktor aufgerufen. `Fido` existiert erst, nachdem dieses Objekt vollständig erzeugt wurde, das heißt, sowohl der

Mammal-Teil als auch der Dog-Teil aufgebaut ist. Daher ist der Aufruf beider Konstruktoren erforderlich.

Beim Abbau des Fido-Objekts wird zuerst der Dog-Destruktor und dann der Destruktor für den Mammal-Teil von Fido aufgerufen. Jeder Destruktor hat die Möglichkeit, seinen eigenen Teil von Fido aufzuräumen. Denken Sie daran, hinter Ihrem Hund sauberzumachen! Listing 16.3 demonstriert die Abläufe.

Listing 16.3: conand-dest.cpp – Aufgerufene Konstruktoren und Destruktoren

```
// Listing 16.3 Aufgerufene Konstruktoren und Destruktoren
#include <iostream>

enum BREED { YORKIE, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };

class Mammal
{
public:
    // Konstruktoren
    Mammal();
    ~Mammal();

    // Zugriffsfunktionen
    int GetAge() const { return itsAge; }
    void SetAge(int age) { itsAge = age; }
    int GetWeight() const { return itsWeight; }
    void SetWeight(int weight) { itsWeight = weight; }

    // Andere Methoden
    void Speak() const { std::cout << "Mammal gib Laut!\n"; }
    void Sleep() const { std::cout << "Psst. Ich schlafe.\n"; }

protected:
    int itsAge;
    int itsWeight;
};

class Dog : public Mammal
{
public:
    // Konstruktoren
    Dog();
    ~Dog();

    // Zugriffsfunktionen
    BREED GetBreed() const { return itsBreed; }
    void SetBreed(BREED breed) { itsBreed = breed; }

    // Andere Methoden
    void WagTail() { std::cout << "Schwanzwedeln...\n"; }
    void BegForFood() { std::cout << "Um Futter betteln...\n"; }
```

```
 42:
 43: private:
 44:     BREED itsBreed;
 45: };
 46:
 47: Mammal::Mammal():
 48: itsAge(1),
 49: itsWeight(5)
 50: {
 51:     std::cout << "Mammal-Konstruktor...\n";
 52: }
 53:
 54: Mammal::~Mammal()
 55: {
 56:     std::cout << "Mammal-Destruktor...\n";
 57: }
 58:
 59: Dog::Dog():
 60: itsBreed(YORKIE)
 61: {
 62:     std::cout << "Dog-Konstruktor...\n";
 63: }
 64:
 65: Dog::~Dog()
 66: {
 67:     std::cout << "Dog-Destruktor...\n";
 68: }
 69:
 70: int main()
 71: {
 72:     Dog fido;   // Einen Hund erzeugen
 73:     fido.Speak();
 74:     fido.WagTail();
 75:     std::cout << "Fido ist " << fido.GetAge() << " Jahre
alt.\n";
 76:     return 0;
 77: }
```

```
Mammal-Konstruktor...
Dog-Konstruktor...
Mammal gib Laut!
Schwanzwedeln...
Fido ist 1 Jahre alt.
Dog-Destruktor...
Mammal-Destruktor...
```

Listing 16.3 entspricht etwa Listing 16.2, enthält jetzt aber die Ausgabe von Meldungen für die Aufrufe von Konstruktoren und Destruktoren. Als Erstes erfolgt der Aufruf des Konstruktors von `Mammal`. Daran schließt sich der Aufruf des Konstruktors von `Dog` an. Damit existiert das `Dog`-Objekt vollständig und man kann dessen Methoden aufrufen. Verliert das `Fido`-Objekt seinen Gültigkeitsbereich, wird der Destruktor von `Dog` und daran anschließend der Destruktor von `Mammal` aufgerufen.

16.3.1 Argumente an Konstruktoren der Basisklasse übergeben

Den Konstruktor von `Mammal` möchte man vielleicht überladen, um ein bestimmtes Alter zu übernehmen, und den Konstruktor von `Dog` überlädt man gegebenenfalls, um eine bestimmte Rasse zu übernehmen. Wie lassen sich die Parameter für Alter und Gewicht an die richtigen Konstruktoren in `Mammal` übergeben? Wie lässt sich erreichen, dass man nur bei `Dog`-Objekten das Gewicht initialisiert, bei `Mammal`-Objekten jedoch nicht?

Die Initialisierung der Basisklasse kann man während der Klasseninitialisierung vornehmen, indem man den Namen der Basisklasse und anschließend die von der Basisklasse erwarteten Parameter angibt. Listing 16.4 zeigt dazu ein Beispiel.

Listing 16.4: overload-derive.cpp – Konstruktoren in abgeleiteten Klassen überladen

```
// Listing 16.4 Konstruktoren in abgeleiteten Klassen überladen
#include <iostream>

enum BREED { YORKIE, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };

class Mammal
{
public:
    // Konstruktoren
    Mammal();
    Mammal(int age);
    ~Mammal();

    // Zugriffsfunktionen
    int GetAge() const { return itsAge; }
    void SetAge(int age) { itsAge = age; }
    int GetWeight() const { return itsWeight; }
    void SetWeight(int weight) { itsWeight = weight; }

    // Andere Methoden
    void Speak() const { std::cout << "Mammal gib Laut!\n"; }
    void Sleep() const { std::cout << "Psst. Ich schlafe.\n"; }

```

```
24: protected:
25:     int itsAge;
26:     int itsWeight;
27: };
28:
29: class Dog : public Mammal
30: {
31: public:
32:     // Konstruktoren
33:     Dog();
34:     Dog(int age);
35:     Dog(int age, int weight);
36:     Dog(int age, BREED breed);
37:     Dog(int age, int weight, BREED breed);
38:     ~Dog();
39:
40:     // Zugriffsmethoden
41:     BREED GetBreed() const { return itsBreed; }
42:     void SetBreed(BREED breed) { itsBreed = breed; }
43:
44:     // Andere Methoden
45:     void WagTail() { std::cout << "Schwanzwedeln...\n"; }
46:     void BegForFood() { std::cout << "Um Futter betteln...\n"; }
47:
48: private:
49:     BREED itsBreed;
50: };
51:
52: Mammal::Mammal():
53: itsAge(1),
54: itsWeight(5)
55: {
56:     std::cout << "Mammal-Konstruktor...\n";
57: }
58:
59: Mammal::Mammal(int age):
60: itsAge(age),
61: itsWeight(5)
62: {
63:     std::cout << "Mammal(int)-Konstruktor...\n";
64: }
65:
66: Mammal::~Mammal()
67: {
68:     std::cout << "Mammal-Destruktor...\n";
69: }
70:
```

```
 71: Dog::Dog():
 72: Mammal(),
 73: itsBreed(YORKIE)
 74: {
 75:     std::cout << "Dog-Konstruktor...\n";
 76: }
 77:
 78: Dog::Dog(int age):
 79: Mammal(age),
 80: itsBreed(YORKIE)
 81: {
 82:     std::cout << "Dog(int)-Konstruktor...\n";
 83: }
 84:
 85: Dog::Dog(int age, int weight):
 86: Mammal(age),
 87: itsBreed(YORKIE)
 88: {
 89:     itsWeight = weight;
 90:     std::cout << "Dog(int, int)-Konstruktor...\n";
 91: }
 92:
 93: Dog::Dog(int age, int weight, BREED breed):
 94: Mammal(age),
 95: itsBreed(breed)
 96: {
 97:     itsWeight = weight;
 98:     std::cout << "Dog(int, int, BREED)-Konstruktor...\n";
 99: }
100:
101: Dog::Dog(int age, BREED breed):
102: Mammal(age),
103: itsBreed(breed)
104: {
105:     std::cout << "Dog(int, BREED)-Konstruktor...\n";
106: }
107:
108: Dog::~Dog()
109: {
110:     std::cout << "Dog-Destruktor...\n";
111: }
112:
113: int main()
114: {
115:     Dog fido;
116:     Dog rover(5);
117:     Dog buster(6,8);
```

```
118:     Dog yorkie (3,YORKIE);
119:     Dog dobbie (4,20,DOBERMAN);
120:     fido.Speak();
121:     rover.WagTail();
122:     std::cout << "Yorkie ist "
123:         << yorkie.GetAge() << " Jahre alt.\n";
124:     std::cout << "Dobbie wiegt "
125:         << dobbie.GetWeight() << " Pfund.\n";
126:     return 0;
127:}
```

Zeilennummern in der Ausgabe

Die hier angegebenen Zeilennummern erscheinen nicht in der Ausgabe, sondern dienen nur als Hilfsmittel für die Analyse.

```
1:  Mammal-Konstruktor...
2:  Dog-Konstruktor...
3:  Mammal(int)-Konstruktor...
4:  Dog(int)-Konstruktor...
5:  Mammal(int)-Konstruktor...
6:  Dog(int, int)-Konstruktor...
7:  Mammal(int)-Konstruktor...
8:  Dog(int, BREED)-Konstruktor...
9:  Mammal(int)-Konstruktor...
10: Dog(int, int, BREED)-Konstruktor...
11: Mammal gib Laut!
12: Schwanzwedeln...
13: Yorkie ist 3 Jahre alt.
14: Dobbie wiegt 20 Pfund.
15: Dog-Destruktor...
16: Mammal-Destruktor...
17: Dog-Destruktor...
18: Mammal-Destruktor...
19: Dog-Destruktor...
20: Mammal-Destruktor...
21: Dog-Destruktor...
22: Mammal-Destruktor...
23: Dog-Destruktor...
24: Mammal-Destruktor...
```

Die Zeile 11 überlädt den Konstruktor von `Mammal`, um das Alter des Säugetiers als ganze Zahl zu übernehmen. Die Implementierung in den Zeilen 59 bis 64 initialisiert `itsAge` mit dem an den Konstruktor übergebenen Wert sowie `itsWeight` mit dem Wert `5`.

In den Zeilen 33 bis 37 stehen fünf überladene Konstruktoren von `Dog`. Der erste ist der Standardkonstruktor. Der zweite übernimmt das Alter, wobei es sich um den gleichen Parameter handelt, den auch der Konstruktor von `Mammal` übernimmt. Der dritte Konstruktor übernimmt sowohl Alter als auch Gewicht, der vierte Alter und Rasse und im fünften Konstruktor finden wir die Parameter Alter, Gewicht und Rasse.

Beachten Sie, dass in Zeile 72 der Standardkonstruktor von `Dog` den Standardkonstruktor von `Mammal` aufruft. Obwohl das nicht zwingend erforderlich ist, dient es der Dokumentation, dass man den Konstruktor der Basisklasse aufrufen möchte, der keine Parameter übernimmt. Der Konstruktor der Basisklasse wird in jedem Fall aufgerufen, das hier gezeigte Verfahren verdeutlicht aber Ihre Absichten explizit.

In den Zeilen 78 bis 83 steht die Implementierung des `Dog`-Konstruktors, der eine ganze Zahl übernimmt. In dessen Initialisierungsphase (Zeilen 79 und 80) initialisiert `Dog` seine Basisklasse mit Übergabe des Parameters und initialisiert dann seine Rasse (`itsBreed`).

Ein weiterer `Dog`-Konstruktor ist in den Zeilen 85 bis 91 zu finden. Dieser übernimmt zwei Parameter. Wieder erfolgt die Initialisierung der Basisklasse durch Aufruf des passenden Konstruktors. Dieses Mal findet aber auch die Zuweisung von `weight` an die Variable `itsWeight` der Basisklasse von `Dog` statt. Beachten Sie, dass die Zuweisung an eine Variable der Basisklasse in der Initialisierungsphase nicht möglich ist. Der Code

```
Dog::Dog(int age, int weight);
Mammal(age),
itsBreed(YORKIE),
itsWeight(weight)           // Fehler!
{
   std::cout << "Dog(int, int) Konstruktor...\n";
}
```

ist also fehlerhaft, weil man einen Wert in der Basisklasse nicht initialisieren kann. Ebenso verbietet sich die Konstruktion

```
Dog::Dog(int age, int weight);
Mammal(age, weight),        // Fehler!
itsBreed(YORKIE)
{
   std::cout << "Dog(int, int) Konstruktor...\n";
}
```

da `Mammal` keinen Konstruktor hat, der den `weight`-Parameter übernimmt; man muss die Zuweisung innerhalb des Rumpfes des Konstruktors von `Dog` erledigen.

```
Dog::Dog(int age, int weight);
Mammal(age),                // Konstruktor der Basisklasse
itsBreed(YORKIE)            // Initialisierung
{
   itsWeight = weight;      // Zuweisung
}
```

Sehen Sie sich die verbleibenden Konstruktoren an, um sich mit deren Arbeitsweise vertraut zu machen. Achten Sie darauf, was man initialisieren kann und was noch im Rumpf des Konstruktors zuzuweisen ist.

Die hier angegebenen Zeilennummern erscheinen nicht in der Ausgabe, sondern dienen nur als Hilfsmittel für die Analyse. Die beiden ersten Zeilen der Ausgabe repräsentieren die Instanzierung von `Fido` mithilfe des Standardkonstruktors.

Die Ausgabezeilen 3 und 4 dokumentieren das Erstellen von `rover`, die Zeilen 5 und 6 von `buster`. Beachten Sie, dass derjenige `Mammal`-Konstruktor aufgerufen wird, der einen Integer-Wert als Parameter übernimmt, während es sich bei `Dog` um den Konstruktor handelt, der zwei Integer-Werte übernimmt.

Nach dem Erstellen aller Objekte werden diese verwendet und verlieren anschließend ihren Gültigkeitsbereich. Beim Abbau der einzelnen Objekte wird zuerst der `Dog`-Destruktor und danach der `Mammal`-Destruktor aufgerufen. Insgesamt sind es von jedem fünf Aufrufe.

16.4 Funktionen redefinieren

Ein `Dog`-Objekt hat Zugriff auf alle Elementfunktionen in der Klasse `Mammal` sowie alle Elementfunktionen (wie `WagTail()`), die die Deklaration der `Dog`-Klasse gegebenenfalls hinzufügt. Ein `Dog`-Objekt kann auch eine Funktion der Basisklasse redefinieren, das heißt, die Implementierung einer Funktion der Basisklasse in einer abgeleiteten Klasse ändern. Wenn man ein Objekt der abgeleiteten Klasse erstellt, wird die korrekte Funktion aufgerufen.

Erzeugt eine abgeleitete Klasse eine Funktion mit demselben Rückgabetyp und derselben Signatur wie eine Elementfunktion in der Basisklasse, aber mit einer neuen Implementierung, spricht man vom *Redefinieren* oder *Überschreiben* dieser Methode.

Die redefinierte Funktion muss bezüglich Rückgabetyp und Signatur mit der Funktion in der Basisklasse übereinstimmen. Als *Signatur* bezeichnet man alles, was zu einem Funktionsprototyp außer dem Rückgabetyp gehört: Name, Parameterliste und – falls verwendet – das Schlüsselwort `const`.

Listing 16.5 zeigt die Abläufe, wenn die Dog-Klasse die Methode Speak() in Mammal redefiniert. Um Platz zu sparen, wurde auf die Zugriffsfunktionen in diesen Klassen verzichtet.

Listing 16.5: baseoverride.cpp – Eine Methode der Basisklasse in einer abgeleiteten Klasse redefinieren

```
// Listing 16.5 Eine Methode der Basisklasse in einer abgeleiteten Klasse redefinieren
#include <iostream>

enum BREED { YORKIE, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };

class Mammal
{
public:
    // Konstruktoren
    Mammal() { std::cout << "Mammal-Konstruktor...\n"; }
    ~Mammal() { std::cout << "Mammal-Destruktor...\n"; }

    //Other methods
    void Speak()const { std::cout << "Mammal gib Laut!\n"; }
    void Sleep()const { std::cout << "Psst. Ich schlafe.\n"; }

protected:
    int itsAge;
    int itsWeight;
};

class Dog : public Mammal
{
public:
    // Konstruktoren
    Dog(){ std::cout << "Dog-Konstruktor...\n"; }
    ~Dog(){ std::cout << "Dog-Destruktor...\n"; }

    // Andere Methoden
    void WagTail() { std::cout << "Schwanzwedeln...\n"; }
    void BegForFood() { std::cout << "Um Futter betteln...\n"; }
    void Speak()const { std::cout << "Wuff!\n"; }

private:
    BREED itsBreed;
};

int main()
{
    Mammal bigAnimal;
    Dog fido;
    bigAnimal.Speak();
```

```
43:     fido.Speak();
44:     return 0;
45: }
```

```
Mammal-Konstruktor...
Mammal-Konstruktor...
Dog-Konstruktor...
Mammal gib Laut!
Wuff!
Dog-Destruktor...
Mammal-Destruktor...
Mammal-Destruktor...
```

In Zeile 32 redefiniert die `Dog`-Klasse die Methode `Speak()`. Dadurch sagt das `Dog`-Objekt »Wuff!«, wenn man die Methode `Speak()` aufruft. Zeile 40 erzeugt das `Mammal`-Objekt `bigAnimal`, das die erste Ausgabezeile produziert, wenn der `Mammal`-Konstruktor aufgerufen wird. Zeile 41 erzeugt das `Dog`-Objekt `Fido`, das die beiden nächsten Ausgabezeilen beim Aufruf der `Mammal`- und `Dog`-Konstruktoren auf den Bildschirm bringt.

Das `Mammal`-Objekt ruft in Zeile 42 seine Methode `Speak()` auf. In Zeile 43 ruft dann das `Dog`-Objekt seine `Speak()`-Methode auf. Die Ausgabe zeigt, dass die korrekten Methoden aufgerufen wurden. Schließlich verlieren die beiden Objekte ihren Gültigkeitsbereich und die Destruktoren werden aufgerufen.

16.4.1 Überladen und Überschreiben

Die Begriffe sind ähnlich und beide Verfahren führen ähnliche Aufgaben aus. Wenn man eine Methode überlädt, erzeugt man mehrere Methoden mit dem gleichen Namen, aber unterschiedlichen Signaturen. Redefiniert (überschreibt) man eine Methode, erzeugt man eine Methode in einer abgeleiteten Klasse mit dem gleichen Namen wie eine Methode in der Basisklasse und mit der gleichen Signatur.

16.4.2 Die Methode der Basisklasse verbergen

Im vorhergehenden Listing verbirgt die Methode `Speak()` der Klasse `Dog` die Methode der Basisklasse. Dies ist zwar beabsichtigt, kann aber zu unerwarteten Ergebnissen führen. Wenn `Mammal` über eine überladene Methode `Move()` verfügt und `Dog` diese Methode redefiniert, verbirgt die `Dog`-Methode alle `Mammal`-Methoden mit diesem Namen.

Wenn Mammal die Methode Move() mit drei Methoden überlädt – eine ohne Parameter, eine mit einem Integer-Parameter und eine mit einer Ganzzahl und einer Richtung – und Dog einfach die Methode Move() ohne Parameter redefiniert, kann man mit einem Dog-Objekt nicht mehr ohne weiteres auf die beiden anderen Methoden zugreifen. Listing 16.6 verdeutlicht dieses Problem.

Listing 16.6: hiding-base.cpp – Methoden verbergen

```
 1: // Listing 16.6 Methoden verbergen
 2:
 3: #include <iostream>
 4:
 5: class Mammal
 6: {
 7: public:
 8:     void Move() const { std::cout << "Mammal geht einen
Schritt.\n"; }
 9:     void Move(int distance) const
10:         { std::cout << "Mammal geht " << distance <<"
Schritte.\n"; }
11: protected:
12:     int itsAge;
13:     int itsWeight;
14: };
15:
16: class Dog : public Mammal
17: {
18: public:
19:     void Move() const { std::cout << "Dog geht 5 Schritte.\n"; }
20: }; // Hier kann eine Warnung erscheinen, dass eine Funktion
verborgen wird!
21:
22: int main()
23: {
24:     Mammal bigAnimal;
25:     Dog fido;
26:     bigAnimal.Move();
27:     bigAnimal.Move(2);
28:     fido.Move();
29:     // fido.Move(10);
30:     return 0;
31: }
```

```
Mammal geht einen Schritt.
Mammal geht 2 Schritte.
Hund geht 5 Schritte.
```

Diese Klassen verzichten auf alle zusätzlichen Methoden und Daten. In den Zeilen 8 und 9 deklariert die `Mammal`-Klasse die überladenen `Move()`-Methoden. In Zeile 19 redefiniert `Dog` die parameterlose Version von `Move()`. Diese Methoden werden in den Zeilen 26 bis 28 aufgerufen und die Ausgabe spiegelt ihre Ausführung wider.

Zeile 29 ist allerdings auskommentiert, da sie einen Compiler-Fehler hervorruft. Die `Dog`-Klasse könnte normalerweise die Methode `Move(int)` aufrufen, wenn diese Klasse nicht die parameterlose Version von `Move()` redefiniert hätte. Nun ist aber diese Methode redefiniert. Möchte man beide Methoden verwenden, muss man auch beide Methoden redefinieren. Das erinnert an die Regel, dass der Compiler keinen Standardkonstruktor bereitstellt, wenn man irgendeinen Konstruktor erzeugt.

Es ist ein häufiger Fehler, eine Methode der Basisklasse zu verbergen, wenn man sie redefinieren will und dabei das Schlüsselwort `const` vergisst. Das Schlüsselwort `const` gehört zur Signatur. Lässt man es weg, ändert man die Signatur und verbirgt damit die Methode, anstatt sie – wie eigentlich beabsichtigt – zu redefinieren.

Manche Compiler geben eine Warnung für den Codeabschnitt in den Zeilen 16 bis 20 aus, der Borland-Compiler nicht. Man darf zwar Methoden der Basisklasse verbergen, doch geschieht das oftmals ungewollt. Deshalb geben manche Compiler eine Warnung aus, wenn sie eine solche Konstruktion erkennen.

16.4.3 Die Basismethode aufrufen

Eine redefinierte Basismethode kann man weiterhin aufrufen, wenn man den vollständig qualifizierten Namen der Methode angibt. Dazu schreibt man den Basisnamen, zwei Doppelpunkte und dann den Methodennamen wie im folgenden Beispiel:

```
Mammal::Move()
```

In Listing 16.6 kann man Zeile 29 wie folgt schreiben, sodass der Compiler keinen Fehler mehr produziert:

```
29:     fido.Mammal::Move(10);
```

Diese Anweisung ruft explizit die `Mammal`-Methode auf. Listing 16.7 zeigt dieses Konzept in seiner Gesamtheit.

Listing 16.7: calling-base.cpp – Die Basismethode von der redefinierten Methode aufrufen

```
 1: // Listing 16.7 Basismethode von redefinierter Methode aufrufen
 2: #include <iostream>
 3:
 4: class Mammal
 5: {
 6: public:
 7:     void Move() const { std::cout << "Mammal geht einen Schritt.\n"; }
 8:     void Move(int distance) const
 9:         { std::cout << "Mammal geht "
10:               << distance << " Schritte.\n"; }
11: protected:
12:     int itsAge;
13:     int itsWeight;
14: };
15:
16: class Dog : public Mammal
17: {
18: public:
19:     void Move()const;
20: };
21:
22: void Dog::Move() const
23: {
24:     std::cout << "In Move von Dog...\n";
25:     Mammal::Move(3);
26: }
27:
28: int main()
29: {
30:     Mammal bigAnimal;
31:     Dog fido;
32:     bigAnimal.Move(2);
33:     fido.Mammal::Move(6);
34:     return 0;
35: }
```

```
Mammal geht 2 Schritte.
Mammal geht 6 Schritte.
```

Zeile 30 erzeugt das `Mammal`-Objekt `bigAnimal`. Die Zeile 31 erzeugt das `Dog`-Objekt `Fido`. Die Anweisung in Zeile 32 ruft die Methode `Move()` des `Mammal`-Objekts mit Übergabe eines `int`-Wertes auf.

Der Programmierer möchte `Move(int)` auf dem `Dog`-Objekt aufrufen, hat aber ein Problem. `Dog` redefiniert die Methode `Move()`, überlädt sie aber nicht und stellt keine Version bereit, die einen `int` übernimmt. Das lässt sich durch den expliziten Aufruf der Methode `Move(int)` der Basisklasse in Zeile 33 lösen.

Was Sie tun sollten

Erweitern Sie die Funktionalität von ausgetesteten Klassen durch Ableiten.

Ändern Sie das Verhalten von bestimmten Funktionen in der abgeleiteten Klasse durch Redefinieren (Überschreiben) der Methoden der Basisklasse.

Achtung

Verbergen Sie keine Funktion der Basisklasse, indem Sie die Signatur der Funktion ändern.

16.5 Fragen und Antworten

F: Werden vererbte Elemente und Funktionen an nachfolgende Generationen weitergereicht? Wenn sich `Dog` von `Mammal` ableitet und `Mammal` von `Animal`, erbt dann `Dog` die Funktionen und Daten von `Animal`?

A: Ja. Setzt man die Ableitung fort, erben die abgeleiteten Klassen die Gesamtheit aller Funktionen und Daten aller darüber liegenden Basisklassen.

F: Kann man in einer abgeleiteten Klasse eine öffentliche Basisfunktion als privat deklarieren?

A: Ja. Diese Funktion bleibt dann aber für alle nachfolgenden Ableitungen ebenfalls privat.

16.6 Workshop

Nachdem Sie sich mit Vererbung vertraut gemacht haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu festigen.

16.6.1 Kontrollfragen

1. Welchen Zweck erfüllen Stubs?
2. Warum leitet man eine Klasse ab?
3. Die Listings 16.1 bis 16.5 enthalten jeweils eine `enum`-Anweisung. Was bewirkt diese?
4. Warum verbirgt man eine Methode der Basisklasse in einer abgeleiteten Klasse (wie zum Beispiel in Listing 16.6)?

16.6.2 Übungen

1. Entfernen Sie die Kommentarzeichen in Zeile 29 von Listing 16.6 (`hidingbase.cpp`). Was passiert? Was müssen Sie tun, damit das Programm funktioniert?
2. Modifizieren Sie Listing 16.2 (`derivedobject.cpp`), um einen Text-String für die Rasse (`breed`) anstelle eines Aufzählungstyps zu verwenden.
3. In Listing 16.5 (`baseoverride.cpp`) existiert die Methode `Sleep()` nur in der Klasse `Mammal`. Ändern Sie das Programm, um diese Methode für `bigAnimal` und `fido` aufzurufen.

16.6.3 Antworten zu den Kontrollfragen

1. Mit Stubs kann man eine erforderliche Methode (Funktion) erzeugen, ohne sich um die Details kümmern zu müssen, wie diese Methode ihre Aufgabe realisiert. Zu einem späteren Zeitpunkt in der Programmentwicklung ergänzt man dann die entsprechenden Anweisungen.
2. Das Ableiten einer Klasse ist einfacher als die Klasse von Grund auf neu zu schreiben. Denn warum sollte man bereits realisierten Code noch einmal von vorn entwickeln?
3. Mit der `enum`-Anweisung lassen sich symbolische Namen (wie zum Beispiel `YORKIE`) anstelle von willkürlichen und schwer verständlichen Konstantenwerten (wie etwa `42`) verwenden.
4. Manchmal unterscheidet sich das Verhalten der abgeleiteten Klasse so weit von der Basisklasse, dass bestimmte Methoden der Basisklasse nicht mehr geeignet sind. Da es nicht immer möglich ist, die Basisklasse zu modifizieren (wenn beispielsweise der Quellcode nicht verfügbar ist), kann man auf diesen Mechanismus zurückgreifen.

KAPITEL 17

jetzt lerne ich

Polymorphie und abgeleitete Klassen

In dieser Stunde lernen Sie,

- was virtuelle Methoden sind,
- wie man virtuelle Destruktoren und Kopierkonstruktoren verwendet,
- wie man Basisklassen mithilfe virtueller Methoden polymorph nutzen kann,
- wo die Gefahren liegen und zu welchen Lasten virtuelle Methoden gehen.

17.1 Polymorphie mit virtuellen Methoden

Im vorherigen Kapitel wurde die Tatsache betont, dass ein `Dog`-Objekt ein `Mammal`-Objekt ist. Bisher hat das nur bedeutet, dass das `Dog`-Objekt die Attribute (Daten) und Fähigkeiten (Methoden) seiner Basisklasse geerbt hat. In C++ geht die »ist-ein«-Beziehung allerdings noch weiter.

Durch Polymorphie lassen sich abgeleitete Objekte wie Basisobjekte behandeln. Nehmen wir zum Beispiel an, Sie erzeugen mehrere spezialisierte `Mammal`-Typen: `Dog`, `Cat`, `Horse` usw. Diese leiten sich alle von `Mammal` ab und `Mammal` hat eine Reihe von Methoden, die sich aus den abgeleiteten Klassen ausklammern lassen. Eine solche Methode ist `Speak()` – alle Säugetiere können Lärm machen.

Jedem abgeleiteten Typ möchte man beibringen, wie im konkreten Fall zu sprechen ist: Ein Hund sagt »Wuff«, eine Katze »Miau« usw. Jede Klasse muss in der Lage sein, die Methode `Speak()` zu redefinieren.

Hat man gleichzeitig eine Sammlung von `Mammal`-Objekten (beispielsweise eine `Farm` mit `Dog`-, `Cat`-, `Horse`- und `Cow`-Objekten), muss die `Farm`-Simulation in der Lage sein, die `Speak()`-Methode der jeweiligen Objekte aufzurufen, ohne sich mit den Details befassen zu müssen, wie die Objekte die Methode `Speak()` implementieren. Betrachtet man diese Objekte sämtlich als `Mammal`-Objekte und ruft die Methode `Mammal.Speak()` auf, behandelt man die Objekte polymorph. Das aus dem Griechischen stammende Wort *polymorph* bedeutet vielgestaltig – man hat es mit `Mammal` in seinen vielen Formen zu tun.

So kann man beispielsweise einen Zeiger auf `Mammal` deklarieren und ihm die Adresse eines `Dog`-Objekts zuweisen, das man auf Heap erstellt. Dog »ist ein« `Mammal` und somit ist die folgende Anweisung korrekt:

```
Mammal* pMammal = new Dog;
```

Über diesen Zeiger lässt sich jede Methode in `Mammal` aufrufen. Für die in `Dog` redefinierten Methoden soll natürlich die korrekte Methode aufgerufen werden. Virtuelle Elementfunktionen erlauben genau das. Wenn Sie diese Objekte polymorph behandeln, rufen Sie die Methode auf dem `Mammal`-Zeiger auf; dabei wissen Sie nicht und müssen sich auch nicht darum kümmern, was das tatsächliche Objekt (in diesem Fall `Dog`) ist oder wie es seine Methoden implementiert.

Listing 17.1 zeigt, wie man Polymorphie mit virtuellen Methoden implementiert.

Listing 17.1: virtmethod.cpp – Virtuelle Methoden

```
 1: // Listing 17.1 Einsatz virtueller Methoden
 2: #include <iostream>
 3:
 4: class Mammal
 5: {
 6: public:
 7:     Mammal():itsAge(1) { std::cout << "Mammal-Konstruktor...\n"; }
 8:     ~Mammal() { std::cout << "Mammal-Destruktor...\n"; }
 9:     void Move() const { std::cout << "Mammal geht einen Schritt.\n"; }
10:     virtual void Speak() const { std::cout << "Mammal spricht!\n"; }
11:
12: protected:
13:     int itsAge;
14: };
15:
16: class Dog : public Mammal
17: {
```

```
18: public:
19:     Dog() { std::cout << "Dog-Konstruktor...\n"; }
20:     ~Dog() { std::cout << "Dog-Destruktor...\n"; }
21:     void WagTail() { std::cout << "Schwanzwedeln...\n"; }
22:     void Speak()const { std::cout << "Wuff!\n"; }
23:     void Move()const { std::cout << "Dog geht 5 Schritte...\n"; }
24: };
25:
26: int main()
27: {
28:     Mammal *pDog = new Dog;
29:     pDog->Move();
30:     pDog->Speak();
31:     return 0;
32: }
```

```
Mammal-Konstruktor...
Dog-Konstruktor...
Mammal geht einen Schritt.
Wuff!
```

Zeile 10 stellt für `Mammal` eine virtuelle Methode – `Speak()` – bereit. Der Designer dieser Klasse gibt dabei zu erkennen, dass diese Klasse letztendlich als Basistyp für eine andere Klasse dienen soll. Die abgeleitete Klasse wird wahrscheinlich diese Funktion redefinieren.

Zeile 28 erzeugt einen Zeiger `pDog` auf `Mammal`, weist ihm aber die Adresse eines neuen `Dog`-Objekts zu. Da ein `Dog`-Objekt von `Mammal` abgeleitet ist, stellt dies eine zulässige Zuweisung dar. Über diesen Zeiger ruft man dann die Funktion `Move()` auf. Da der Compiler `pDog` nur als `Mammal`-Objekt kennt, sucht er beim `Mammal`-Objekt nach der Methode `Move()`.

In Zeile 30 verweist der Zeiger auf die Methode `Speak()`. Da `Speak()` virtuell ist, wird die in `Dog` redefinierte `Speak()`-Methode aufgerufen.

Das ist schon fast Zauberei. Der aufrufenden Funktion ist eigentlich nur bekannt, dass sie einen `Mammal`-Zeiger hat. Hier wird aber eine Methode von `Dog` aufgerufen. In der Tat ließe sich ein Array von Zeigern auf `Mammal` einrichten, in dem jeder Zeiger auf eine Unterklasse von `Mammal` verweist. Führt man nacheinander Aufrufe mit diesen Zeigern aus, aktiviert das Programm immer die korrekte Funktion. Listing 17.2 verdeutlicht dieses Konzept.

Listing 17.2: multivirtual.cpp – Mehrere virtuelle Elementfunktionen der Reihe nach aufrufen

```
// Listing 17.2 Mehrere virtuelle Elementfunktionen der Reihe nach aufrufen
#include <iostream>

class Mammal
{
public:
    Mammal():itsAge(1) {  }
    ~Mammal() { }
    virtual void Speak() const { std::cout << "Mammal gib Laut!\n"; }
protected:
    int itsAge;
};

class Dog : public Mammal
{
public:
    void Speak()const { std::cout << "Wuff!\n"; }
};


class Cat : public Mammal
{
public:
    void Speak()const { std::cout << "Miau!\n"; }
};


class Horse : public Mammal
{
public:
    void Speak()const { std::cout << "Wieher!\n"; }
};

class Pig : public Mammal
{
public:
    void Speak()const { std::cout << "Grunz!\n"; }
};

int main()
{
    Mammal* theArray[5];
    Mammal* ptr;
    int choice, i;
    for ( i = 0; i<5; i++)
    {
```

```
47:         std::cout << "(1)Hund (2)Katze (3)Pferd (4)Schwein: ";
48:         std::cin >> choice;
49:         switch (choice)
50:         {
51:         case 1:
52:             ptr = new Dog;
53:             break;
54:         case 2:
55:             ptr = new Cat;
56:             break;
57:         case 3:
58:             ptr = new Horse;
59:             break;
60:         case 4:
61:             ptr = new Pig;
62:             break;
63:         default:
64:             ptr = new Mammal;
65:             break;
66:         }
67:         theArray[i] = ptr;
68:     }
69:     for (i=0;i<5;i++)
70:         theArray[i]->Speak();
71:     return 0;
72: }
```

```
(1)Hund (2)Katze (3)Pferd (4)Schwein: 1
(1)Hund (2)Katze (3)Pferd (4)Schwein: 2
(1)Hund (2)Katze (3)Pferd (4)Schwein: 3
(1)Hund (2)Katze (3)Pferd (4)Schwein: 4
(1)Hund (2)Katze (3)Pferd (4)Schwein: 5
Wuff!
Miau!
Wieher!
Grunz!
Mammal gib Laut!
```

Dieses abgespeckte Programm, das lediglich die grundlegende Funktionalität für jede Klasse bereitstellt, zeigt virtuelle Elementfunktionen in ihrer reinsten Form. Die vier deklarierten Klassen – `Dog`, `Cat`, `Horse` und `Pig` (Hund, Katze, Pferd und Schwein) – sind alle von `Mammal` abgeleitet.

Zeile 9 deklariert die Funktion `Speak()` von `Mammal` als virtuell. In den Zeilen 17, 24, 31 und 37 redefinieren die vier abgeleiteten Klassen die Implementierung von `Speak()`.

Der Benutzer soll auswählen, welche Objekte zu erzeugen sind. Dementsprechend werden die Zeiger in den Zeilen 45 bis 66 in das Array aufgenommen.

Spätes Binden

Beachten Sie, dass zur Kompilierzeit noch gar nicht bekannt sein kann, welches Objekt erzeugt und welche der `Speak()`-Methoden demzufolge aufgerufen wird. Der Zeiger `ptr` wird an sein Objekt zur Laufzeit gebunden. Man bezeichnet das als *spätes Binden* oder *Binden zur Laufzeit* im Gegensatz zum statischen Binden oder Binden zur Kompilierzeit.

17.1.1 Arbeitsweise virtueller Elementfunktionen

Beim Erzeugen eines abgeleiteten Objekts, etwa eines `Dog`-Objekts, wird zuerst der Konstruktor für die Basisklasse und dann der Konstruktor für die abgeleitete Klasse aufgerufen. Abbildung 17.1 zeigt, wie sich das `Dog`-Objekt nach dessen Erzeugung darstellt. Beachten Sie, dass der `Mammal`-Teil des Objekts unmittelbar zusammen mit dem `Dog`-Teil im Speicher abgelegt ist.

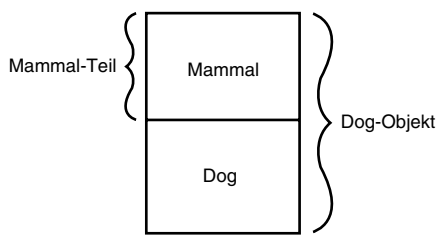


Abb. 17.1: Das Dog-Objekt nach dessen Erzeugung

Wenn eine virtuelle Funktion in einem Objekt erzeugt wird, muss das Objekt die Spur dieser Funktion verfolgen. Viele Compiler bauen eine so genannte *V-Tabelle* für virtuelle Funktionen auf. Jeder Typ erhält eine eigene Tabelle und jedes Objekt dieses Typs verwaltet einen virtuellen Tabellenzeiger (einen so genannten `vptr` oder *V-Zeiger*) auf diese Tabelle.

Trotz verschiedenartiger Implementierungen müssen alle Compiler die gleiche Aufgabe umsetzen, sodass diese Beschreibung zumindest im Kern zutrifft.

Jeder `vptr` eines Objekts zeigt auf die V-Tabelle, die wiederum Zeiger auf alle virtuellen Elementfunktionen enthält. Beim Erzeugen des `Mammal`-Teils von `Dog` wird der Zeiger `vptr` so initialisiert, dass er auf die virtuellen Methoden für `Mammal` zeigt. Abbildung 17.2 verdeutlicht diesen Sachverhalt.

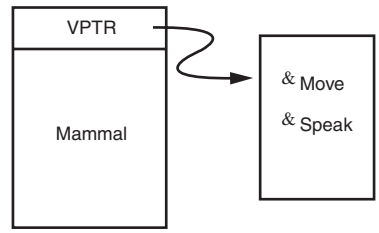


Abb. 17.2: Die V-Tabelle eines Mammal-Objekts

Wenn beim Aufruf des `Dog`-Konstruktors der `Dog`-Teil dieses Objekts hinzukommt, wird `vptr` angepasst und zeigt nun auf die redefinierten virtuellen Funktionen (falls vorhanden) im `Dog`-Objekt, wie es Abbildung 17.3 zeigt.

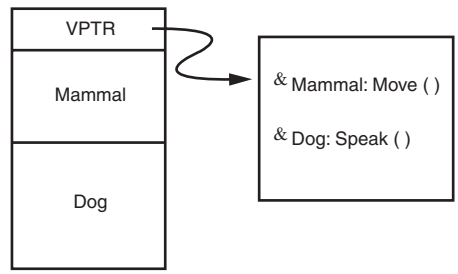


Abb. 17.3: Die V-Tabelle eines Dog-Objekts

Wenn man einen Zeiger auf ein `Mammal` verwendet, weist der `vptr` weiterhin – je nach tatsächlichem Typ des Objekts – auf die richtige Funktion. Ruft man `Speak()` auf, aktiviert das Programm folglich die betreffende Funktion.

17.1.2 Verbotene Pfade

Verfügt das `Dog`-Objekt über eine »Schwanzwedel«-Methode `WagTail()`, die im `Mammal`-Objekt nicht vorhanden ist, kann man mit dem Zeiger auf `Mammal` nicht auf diese Methode zugreifen (sofern man keine Typumwandlung in einen Zeiger auf `Dog` vornimmt). Da `WagTail()` keine virtuelle Funktion ist und sich diese Methode nicht in einem `Mammal`-Objekt befindet, lässt sich dieser Weg nur über ein `Dog`-Objekt oder einen `Dog`-Zeiger beschreiten.

Obwohl man den `Mammal`-Zeiger in einen `Dog`-Zeiger umwandeln kann, gibt es wesentlich bessere und sicherere Wege, um die Methode `WagTail()` aufzurufen. In C++ sollte man auf explizite Typumwandlungen verzichten, da sie fehleranfällig sind. Auf dieses Thema geht Stunde 18 im Zusammenhang mit der Mehrfachvererbung sowie Stunde 24 im Detail ein.

17.1.3 Slicing (Datenteilung)

Der Zauber der virtuellen Funktionen wirkt nur bei Zeigern und Referenzen. Die Übergabe eines Objekts als Wert lässt den Aufruf virtueller Elementfunktionen nicht zu. Listing 17.3 verdeutlicht dieses Problem.

Listing 17.3: slicing.cpp – Datenteilung bei Übergabe als Wert

```
// Listing 17.3 Datenteilung bei Übergabe als Wert
#include <iostream>

class Mammal
{
public:
    Mammal():itsAge(1) {  }
    ~Mammal() { }
    virtual void Speak() const { std::cout << "Mammal gib
Laut!\n"; }
protected:
    int itsAge;
};

class Dog : public Mammal
{
public:
    void Speak()const { std::cout << "Wuff!\n"; }
};

class Cat : public Mammal
{
public:
    void Speak()const { std::cout << "Miau!\n"; }
};

void ValueFunction (Mammal);
void PtrFunction   (Mammal*);
void RefFunction   (Mammal&);

int main()
{
    Mammal* ptr=0;
    int choice;
    while (1)
    {
        bool fQuit = false;
        std::cout << "(1)Hund (2)Katze (0)Beenden: ";
        std::cin >> choice;
        switch (choice)
        {
```

```
41:         case 0:
42:             fQuit = true;
43:             break;
44:         case 1:
45:             ptr = new Dog;
46:             break;
47:         case 2:
48:             ptr = new Cat;
49:             break;
50:         default:
51:             ptr = new Mammal;
52:             break;
53:         }
54:         if (fQuit)
55:             break;
56:         PtrFunction(ptr);
57:         RefFunction(*ptr);
58:         ValueFunction(*ptr);
59:     }
60:     return 0;
61: }
62:
63: void ValueFunction (Mammal MammalValue) // Diese Funktion wird
zuletzt aufgerufen
64: {
65:     MammalValue.Speak();
66: }
67:
68: void PtrFunction (Mammal * pMammal)
69: {
70:     pMammal->Speak();
71: }
72:
73: void RefFunction (Mammal & rMammal)
74: {
75:     rMammal.Speak();
76: }
```

```
(1)Hund (2)Katze (0)Beenden: 1
Wuff!
Wuff!
Mammal gib Laut!
(1)Hund (2)Katze (0)Beenden: 2
Miau!
Miau!
Mammal gib Laut!
(1)Hund (2)Katze (0)Beenden: 0
```

Die Zeilen 4 bis 24 deklarieren abgespeckte Versionen der Klassen `Mammal`, `Dog` und `Cat`. Die drei deklarierten Funktionen `PtrFunction()`, `RefFunction()` und `ValueFunction()` übernehmen einen Zeiger auf ein `Mammal`, eine `Mammal`-Referenz bzw. ein `Mammal`-Objekt. Alle drei Funktionen bewirken dann das Gleiche – sie rufen die Methode `Speak()` auf.

Der Benutzer wird aufgefordert, einen Hund (`Dog`-Objekt) oder eine Katze (`Cat`-Objekt) zu wählen. Gemäß der Auswahl erzeugen die Zeilen 39 bis 53 einen Zeiger auf den richtigen Typ.

In der ersten Ausgabezeile hat der Benutzer einen Hund gewählt. Zeile 45 erzeugt das `Dog`-Objekt auf dem Heap. Dann erfolgt die Übergabe des `Dog`-Objekts an drei Funktionen als Zeiger, als Referenz und als Wert. Der Zeiger und die Referenzen rufen virtuelle Elementfunktionen auf und es wird die Elementfunktion `Dog->Speak()` aktiviert. Das zeigt sich in den beiden ersten Ausgabezeilen nach der Benutzerauswahl.

Die Übergabe des dereferenzierten Zeigers erfolgt allerdings als Wert. Die Funktion erwartet ein `Mammal`-Objekt (wie in der Argumentliste spezifiziert), sodass der Compiler das `Dog`-Objekt genau bis zum `Mammal`-Teil auftrennt – das so genannte Slicing. An diesem Punkt erfolgt der Aufruf der `Mammal`-Methode `Speak()`, wie es die dritte Ausgabezeile nach der Benutzerauswahl dokumentiert.

Dieses Experiment wiederholt sich mit entsprechenden Ergebnissen für das `Cat`-Objekt.

17.1.4 Virtuelle Destruktoren

Es ist zulässig und üblich, einen Zeiger auf ein abgeleitetes Objekt zu übergeben, wenn ein Zeiger auf ein Basisobjekt erwartet wird. Was passiert, wenn dieser Zeiger auf ein abgeleitetes Objekt gelöscht wird? Ist der Destruktor virtuell (wie es sein sollte), geht alles in Ordnung – der Destruktor der abgeleiteten Klasse wird aufgerufen. Da der Destruktor der abgeleiteten Klasse automatisch den Destruktor der Basisklasse aufruft, wird das gesamte Objekt geeignet abgebaut.

Als Faustregel für diesen Fall gilt: Wenn irgendeine Funktion in der Klasse virtuell ist, sollte der Destruktor ebenfalls virtuell sein.

17.1.5 Virtuelle Kopierkonstruktoren

Wie bereits erwähnt können Konstruktoren nicht virtuell sein. Trotzdem kann es vorkommen, dass ein Programm unbedingt einen Zeiger auf ein Basisobjekt übergeben muss und eine Kopie des erzeugten und korrekt abgeleiteten Objekts braucht. Eine übliche Lösung für dieses Problem besteht in der Erzeu-

gung einer Clone-Methode in der Basisklasse, wobei man diese Methode virtuell deklariert. Eine *Clone-Methode* erzeugt eine neue Kopie des aktuellen Objekts und gibt dieses Objekt zurück.

Da jede abgeleitete Klasse die Clone-Methode überschreibt, wird eine Kopie der abgeleiteten Klasse erzeugt. Listing 17.4 zeigt ein entsprechendes Beispiel.

Listing 17.4: virtual-copy.cpp – Virtueller Kopierkonstruktor

```
 1: // Listing 17.4 Virtueller Kopierkonstruktor
 2: #include <iostream>
 3:
 4: class Mammal
 5: {
 6: public:
 7:     Mammal():itsAge(1) { std::cout << "Mammal-Konstruktor...\n"; }
 8:     virtual ~Mammal() { std::cout << "Mammal-Destruktor...\n"; }
 9:     Mammal (const Mammal & rhs);
10:     virtual void Speak() const { std::cout << "Mammal gib Laut!\n"; }
11:     virtual Mammal* Clone() { return new Mammal(*this); }
12:     int GetAge()const { return itsAge; }
13:
14: protected:
15:     int itsAge;
16: };
17:
18: Mammal::Mammal (const Mammal & rhs):itsAge(rhs.GetAge())
19: {
20:     std::cout << "Mammal-Kopierkonstruktor...\n";
21: }
22:
23: class Dog : public Mammal
24: {
25: public:
26:     Dog() { std::cout << "Dog-Konstruktor...\n"; }
27:     virtual ~Dog() { std::cout << "Dog-Destruktor...\n"; }
28:     Dog (const Dog & rhs);
29:     void Speak()const { std::cout << "Wuff!\n"; }
30:     virtual Mammal* Clone() { return new Dog(*this); }
31: };
32:
33: Dog::Dog(const Dog & rhs):
34: Mammal(rhs)
35: {
36:     std::cout << "Dog-Kopierkonstruktor...\n";
37: }
38:
39: class Cat : public Mammal
40: {
```

```
41: public:
42:     Cat() { std::cout << "Cat-Konstruktor...\n"; }
43:     virtual ~Cat() { std::cout << "Cat-Destruktor...\n"; }
44:     Cat (const Cat &);
45:     void Speak()const { std::cout << "Miau!\n"; }
46:     virtual Mammal* Clone() { return new Cat(*this); }
47: };
48:
49: Cat::Cat(const Cat & rhs):
50: Mammal(rhs)
51: {
52:     std::cout << "Cat-Kopierkonstruktor...\n";
53: }
54:
55: enum ANIMALS { MAMMAL, DOG, CAT};
56: const int NumAnimalTypes = 3;
57: int main()
58: {
59:     Mammal *theArray[NumAnimalTypes];
60:     Mammal* ptr;
61:     int choice,i;
62:     for (i = 0; i<NumAnimalTypes; i++)
63:     {
64:         std::cout << "(1)Hund (2)Katze (3)Säugetier: ";
65:         std::cin >> choice;
66:         switch (choice)
67:         {
68:         case DOG:
69:             ptr = new Dog;
70:             break;
71:         case CAT:
72:             ptr = new Cat;
73:             break;
74:         default:
75:             ptr = new Mammal;
76:             break;
77:         }
78:         theArray[i] = ptr;
79:     }
80:     Mammal *OtherArray[NumAnimalTypes];
81:     for (i=0;i<NumAnimalTypes;i++)
82:     {
83:         theArray[i]->Speak();
84:         OtherArray[i] = theArray[i]->Clone();
85:     }
86:     for (i=0;i<NumAnimalTypes;i++)
87:         OtherArray[i]->Speak();
88:     return 0;
89: }
```

```
1:  (1)Hund (2)Katze (3)Säugetier: 1
2:  Mammal-Konstruktor...
3:  Dog-Konstruktor...
4:  (1)Hund (2)Katze (3)Säugetier: 2
5:  Mammal-Konstruktor...
6:  Cat-Konstruktor...
7:  (1)Hund (2)Katze (3)Säugetier: 3
8:  Mammal-Konstruktor...
9:  Wuff!
10: Mammal-Kopierkonstruktor...
11: Dog-Kopierkonstruktor...
12: Miau!
13: Mammal-Kopierkonstruktor...
14: Cat-Kopierkonstruktor...
15: Mammal gib Laut!
16: Mammal-Kopierkonstruktor...
17: Wuff!
18: Miau!
19: Mammal gib Laut!
```

Listing 17.4 ist den beiden vorherigen Listings sehr ähnlich. Lediglich die `Mammal`-Klasse hat eine neue virtuelle Methode erhalten: `Clone()`. Diese Methode gibt einen Zeiger auf ein neues `Mammal`-Objekt durch Aufruf des Kopierkonstruktors zurück und übergibt sich dabei selbst (`*this`) als konstante Referenz.

`Dog` und `Cat` überschreiben die `Clone()`-Methode, wobei sie deren Daten initialisieren und Kopien von sich selbst an ihre eigenen Kopierkonstruktoren übergeben. Da `Clone()` eine virtuelle Methode ist, erzeugt dies praktisch einen virtuellen Kopierkonstruktor, wie es aus Zeile 84 hervorgeht.

Der Benutzer wählt hier Hunde, Katzen oder Säugetiere, deren Erzeugung in den Zeilen 66 bis 77 erfolgt. Zeile 78 speichert einen Zeiger auf jede Auswahl in einem Array.

Beim Durchlauf durch das Array lässt jedes Objekt in den Zeilen 83 und 84 der Reihe nach seine Methoden `Speak()` und `Clone()` aufrufen. Der Aufruf von `Clone()` liefert einen Zeiger auf eine Kopie des Objekts, das Zeile 84 dann in einem zweiten Array speichert.

Ausgabezeile 1 zeigt die Benutzeraufforderung und die eingegebene `1` für die Auswahl eines Hundes. Damit werden die `Mammal`- und `Dog`-Konstruktoren aufgerufen. Das Ganze wiederholt sich für die Konstruktoren von `Cat` und `Mammal` in den Ausgabezeilen 4 bis 8.

Zeile 9 der Ausgabe gibt den Aufruf von `Speak()` auf dem ersten Objekt, dem `Dog`-Objekt, von Zeile 83 wieder. Das Programm ruft die virtuelle Methode `Speak()` auf und aktiviert die korrekte Version. Es schließt sich der Aufruf der Methode `Clone()` an. Da sie ebenfalls virtuell ist, folgt der Aufruf der Methode `Clone()` des `Dog`-Objekts, was zum Aufruf des `Mammal`-Konstruktors und des `Dog`-Kopierkonstruktors führt.

Das Gleiche wiederholt sich für `Cat` in den Ausgabezeilen 12 bis 14 und für `Mammal` in den Ausgabezeilen 15 und 16. Schließlich wird das neue Array durchlaufen (Ausgabezeilen 17 bis 19, Codezeilen 86 und 87), wobei jedes neue Objekt die Methode `Speak()` aufruft.

Bei diesem Verfahren müssen Sie als Programmierer im Unterschied zur Verwendung des Kopierkonstruktors die Methode `Clone()` explizit aufrufen. Der Kopierkonstruktor wird automatisch aufgerufen, wenn ein Objekt kopiert wird. Denken Sie daran, dass Sie die Kopiermethode in einer abgeleiteten Klasse immer überschreiben können. Allerdings sind Sie dann nicht mehr so flexibel.

17.1.6 Der Preis der virtuellen Methoden

Da Objekte mit virtuellen Methoden über eine V-Tabelle arbeiten, fallen zusätzliche Verwaltungsaufgaben an. Bei einer kleinen Klasse, von der man voraussichtlich keine andere Klasse ableitet, gibt es für virtuelle Methoden überhaupt keine Berechtigung.

Hat man irgendwelche Methoden als virtuell deklariert, sind die »Anschaffungskosten« für die V-Tabelle zum größten Teil bereits bezahlt (auch wenn jeder Eintrag etwas zusätzliche Speicherverwaltung mit sich bringt). Jetzt kommt der Zeitpunkt, wo der Destruktor auch virtuell sein soll, und man schließt messerscharf, dass alle anderen Methoden wahrscheinlich ebenfalls virtuell werden. Sehen Sie sich eingehend alle nicht virtuellen Methoden an, bis Ihnen klar ist, warum diese nicht virtuell sind.

Was Sie tun sollten

Verwenden Sie virtuelle Methoden, wenn Sie höchstwahrscheinlich von einer Klasse ableiten.

Verwenden Sie einen virtuellen Destruktor, wenn irgendeine Methode virtuell ist.

Achtung

Kennzeichnen Sie den Konstruktor nicht als virtuell.

17.2 Fragen und Antworten

F: Warum deklariert man nicht alle Funktionen einer Klasse als virtuell?

A: Bei der ersten virtuellen Funktion entsteht zusätzlicher Verwaltungsaufwand durch das Anlegen einer V-Tabelle. Danach lässt sich der Overhead vernachlässigen. Viele C++-Programmierer gehen einfach davon aus, dass alle Funktionen virtuell sein müssen, wenn mindestens eine Funktion als virtuell deklariert ist. Andere Programmierer stimmen damit nicht überein, sondern fragen immer nach einem vernünftigen Grund für eine virtuelle Funktion.

F: Angenommen sei eine virtuelle Funktion `SomeFunc()` in einer Basisklasse. Diese Funktion wird überladen und übernimmt entweder einen oder zwei Integer-Wert(e). Die abgeleitete Klasse redefiniert nun die Version mit *einem* Integer-Wert. Welche Funktion wird dann aktiviert, wenn ein Zeiger auf ein abgeleitetes Objekt die Version mit *zwei* Integer-Argumenten aufruft?

A: Das Redefinieren der Version mit einem Integer-Argument verbirgt die gesamte Funktion der Basisklasse. Demzufolge erhält man einen Compiler-Fehler mit dem Hinweis, dass diese Funktion nur einen Integer-Wert erfordert.

17.3 Workshop

Nachdem Sie sich mit Polymorphie und abgeleiteten Klassen beschäftigt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu vertiefen.

17.3.1 Kontrollfragen

1. Ist es möglich, Listing 17.3 (`slicing.cpp`) zu ändern, um die Methode `Speak()` der abgeleiteten Klasse aufzurufen, indem man `ValueFunction()` überschreibt?
2. Woher weiß das Programm, welche virtuelle Funktion aufzurufen ist, wenn die Objekte in einer Variablen vom Typ der Basisklasse gespeichert sind?

3. Welcher Methodentyp kann nicht virtuell sein?
4. Was ist Polymorphie?

17.3.2 Übungen

1. Was passiert, wenn eine abgeleitete Klasse eine virtuelle Methode der Basisklasse nicht überschreibt? Kommentieren Sie in Listing 17.1 (`virtmethod.cpp`) die Zeile 22 aus: die Methode `Speak()` innerhalb von `Dog`. Können Sie sich Beispiele vorstellen, wo so etwas sinnvoll wäre?
2. Was passiert, wenn Sie in Listing 17.3 (`slicing.cpp`) das Schlüsselwort `virtual` in Zeile 9 (bei der Definition von `Speak()` in der Basisklasse) entfernen? Können Sie erkennen, warum die überschriebenen Funktionen niemals aufgerufen werden. (Hinweis: Sehen Sie sich Kontrollfrage 1 an.)
3. Probieren Sie alle Codebeispiele aus und verwenden Sie andere als die im Buch angegebenen Werte, um ein besseres Gefühl dafür zu bekommen, wie der Code funktioniert.

17.3.3 Antworten zu den Kontrollfragen

1. Nein, nicht direkt. Obwohl sich eine `Dog`-Version dieser Funktion erzeugen lässt, wird sie nie aufgerufen, weil die zugrunde liegende Variable zur Basisklasse (`Mammal`) gehört. Nur die Version von `ValueFunction()`, die `Mammal` als Parameter hat, wird aufgerufen.
2. Die V-Tabelle verfolgt diese Informationen automatisch. Der mit dieser Tabelle verbundene Overhead ist es, der den Aufruf virtueller Funktionen etwas teurer macht als bei regulären Methoden.
3. Der Konstruktor (einschließlich des Kopierkonstruktors).
4. Polymorphie behandelt die Mitglieder von abgeleiteten Klassen, als würden sie zur Basisklasse gehören. *Poly* bedeutet viel; *morph* heißt Form.

KAPITEL 18

jetzt lerne ich

Polymorphie – weiterführende Themen

In dieser Stunde lernen Sie,

- was man unter »absteigender Typumwandlung« versteht und warum man sie einsetzt,
- was abstrakte Datentypen sind,
- was abstrakte Funktionen sind.

18.1 Probleme mit einfacher Vererbung

Die letzten Kapitel haben die polymorphe Behandlung abgeleiteter Objekte bezüglich ihrer Basisklassen erläutert. Wenn zum Beispiel die Methode `Speak()` der Basisklasse in der abgeleiteten Klasse redefiniert wird, führt ein Zeiger auf ein Basisobjekt, dem ein abgeleitetes Objekt zugewiesen wurde, »das Richtige« aus. Listing 18.1 verdeutlicht dieses Konzept.

Listing 18.1: virtualmethods.cpp – Virtuelle Methoden

```
 1: // Listing 18.1 - Virtuelle Methoden
 2: #include <iostream>
 3:
 4: class Mammal
 5: {
 6: public:
 7:     Mammal():itsAge(1) { std::cout << "Mammal-Konstruktor...\n"; }
 8:     virtual ~Mammal() { std::cout << "Mammal-Destruktor...\n"; }
 9:     virtual void Speak() const { std::cout << "Mammal gib Laut!\n"; }
10: protected:
11:     int itsAge;
```

```
12: };
13:
14: class Cat: public Mammal
15: {
16: public:
17:     Cat() { std::cout << "Cat-Konstruktor...\n"; }
18:     ~Cat() { std::cout << "Cat-Destruktor...\n"; }
19:     void Speak()const { std::cout << "Miau!\n"; }
20: };
21:
22: int main()
23: {
24:     Mammal *pCat = new Cat;
25:     pCat->Speak();
26:     return 0;
27: }
```

```
Mammal-Konstruktor...
Cat-Konstruktor...
Miau!
```

Zeile 9 deklariert `Speak()` als virtuelle Methode. Zeile 19 redefiniert diese Methode und Zeile 25 ruft sie auf. Beachten Sie wieder, dass `pCat` als Zeiger auf `Mammal` deklariert ist, aber die Adresse von `Cat` zugewiesen bekommt. Es handelt sich hier um das in Stunde 17 behandelte Wesen der Polymorphie.

Was passiert aber, wenn man eine Methode in `Cat` hinzufügen möchte, die für `Mammal` nicht passend ist? Nehmen wir zum Beispiel an, es sei eine »Schnurren«-Methode namens `Purr()` hinzuzufügen. Katzen schnurren, andere Säugetiere aber nicht. Die Klasse definiert man dann etwa wie folgt:

```
class Cat: public Mammal
{
public:
   Cat() { std::cout << "Cat-Konstruktor...\n"; }
   ~Cat() { std::cout << "Cat-Destruktor...\n"; }
  void Speak()const { std::cout << "Miau\n"; }
  void Purr()const { std::cout << "rrrrrrrrrrrrrrr\n"; }
};
```

Das Problem hierbei: Wenn man `Purr()` mithilfe des Zeigers auf `Mammal` aufruft, erhält man einen Compiler-Fehler. Der Borland-Compiler reagiert darauf wie folgt:

```
"virtualmethods.cpp": E2316 'Purr' ist kein Element von 'Mammal' in
Funktion main() in Zeile 27
```

Andere Compiler liefern ähnliche Fehlermeldungen.

Wenn der Compiler versucht, `Purr()` in seiner virtuellen `Mammal`-Tabelle aufzulösen, gibt es keinen Eintrag. Nun bieten sich mehrere Kompromisslösungen an.

Man kann diese Methode nach oben in die Basisklasse verschieben, was zwar funktioniert, aber einem guten Programmierstil widerspricht. Wenn man die Basisklasse mit Methoden aufbläht, die zu abgeleiteten Klassen spezifisch sind, produziert man schwer zu wartenden Code.

In der Tat spiegelt das gesamte Problem einen schlechten Entwurf wider. Hat man nämlich einen Zeiger auf eine Basisklasse, dem ein abgeleitetes Klassenobjekt zugewiesen ist, möchte man im Allgemeinen dieses Objekt polymorph verwenden. In diesem Fall sollte man nicht einmal versuchen, auf Methoden zuzugreifen, die spezifisch zur abgeleiteten Klasse sind.

Um es deutlicher zu sagen: Das Problem ist nicht, dass man solche spezifischen Methoden hat, sondern dass man versucht, auf sie über den Basisklassenzeiger zuzugreifen. Verfügt man über einen derartigen Zeiger, würde man im Idealfall gar nicht versuchen, an derartige Methoden heranzukommen.

Der Idealfall tritt aber selten ein und irgendwann hat man eine ganze Sammlung von Basisobjekten – beispielsweise einen Zoo voller Säugetiere. Nun kann es vorkommen, dass man ein `Cat`-Objekt erzeugt hat und das verflixte Ding zum Schnurren bringen will. Dann bleibt einem nichts weiter übrig, als etwas zu mogeln.

Und das funktioniert folgendermaßen: Man wandelt den Zeiger der Basisklasse in den abgeleiteten Typ um. Man sagt zum Compiler: »Schau an, mein Freund, ich weiß zufällig, dass es wirklich eine Katze ist, mach also genau das, was ich dir sage.« Freundliche Worte allein genügen aber nicht, denn man presst praktisch ein Katzen-Verhalten aus einem `Mammal`-Zeiger heraus.

Zu diesem Zweck steht der Operator `dynamic_cast` zur Verfügung. Dieser Operator gewährleistet eine sichere Typumwandlung. Darüber hinaus lassen sich durch diesen Operator schnell diejenigen Stellen im Code finden, wo man dieses Verfahren einsetzt, und man kann die entsprechenden Abschnitte schnell umarbeiten, sobald man eine bessere Lösung gefunden hat.

Wenn man einen Zeiger auf eine Basisklasse wie `Mammal` hat und ihm einen Zeiger auf eine abgeleitete Klasse wie `Cat` zuweist, kann man den `Mammal`-Zeiger polymorph verwenden, um auf virtuelle Funktionen zuzugreifen. Muss man dann an das `Cat`-Objekt herankommen, um zum Beispiel die `Purr()`-Methode aufzurufen, erzeugt man einen `Cat`-Zeiger und nimmt die Umwandlung mit dem Operator `dynamic_cast` vor.

Zur Laufzeit wird der Basiszeiger untersucht. Geht die Umwandlung in Ordnung, erhält man einen einwandfreien neuen `Cat`-Zeiger. Wenn überhaupt

kein Cat-Objekt existiert und die Umwandlung scheitert, bekommt man als neuen Zeiger NULL zurück. Listing 18.2 verdeutlicht dies.

Listing 18.2: dynamic-cast.cpp – Dynamische Typumwandlung

```
// Listing 18.2 - Dynamische Typumwandlung
#include <iostream>
using std::cout; // Diese Datei verwendet std::cout

class Mammal
{
public:
    Mammal():itsAge(1) { cout << "Mammal-Konstruktor...\n"; }
    virtual ~Mammal() { cout << "Mammal-Destruktor...\n"; }
    virtual void Speak() const { cout << "Mammal gib Laut!\n"; }
protected:
    int itsAge;
};

class Cat: public Mammal
{
public:
    Cat() { cout << "Cat-Konstruktor...\n"; }
    ~Cat() { cout << "Cat-Destruktor...\n"; }
    void Speak()const { cout << "Miau\n"; }
    void Purr() const { cout << "rrrrrrrrrr\n"; }
};

class Dog: public Mammal
{
public:
    Dog() { cout << "Dog-Konstruktor...\n"; }
    ~Dog() { cout << "Dog-Destruktor...\n"; }
    void Speak()const { cout << "Wuff!\n"; }
};


int main()
{
    const int NumberMammals = 3;
    Mammal* Zoo[NumberMammals];
    Mammal* pMammal;
    int choice,i;
    for (i=0; i<NumberMammals; i++)
    {
        cout << "(1)Hund (2)Katze: ";
        std::cin >> choice;
        if (choice == 1)
            pMammal = new Dog;
        else
            pMammal = new Cat;
```

```
47:
48:         Zoo[i] = pMammal;
49:     }
50:
51:     cout << "\n";
52:
53:     for (i=0; i<NumberMammals; i++)
54:     {
55:         Zoo[i]->Speak();
56:
57:         Cat *pRealCat =  dynamic_cast<Cat *> (Zoo[i]);
58:
59:         if (pRealCat)
60:             pRealCat->Purr();
61:         else
62:             cout << "Oh nein, keine Katze!\n";
63:
64:         delete Zoo[i];
65:         cout << "\n";
66:     }
67:
68:     return 0;
69: }
```

```
(1)Hund (2)Katze: 1
Mammal-Konstruktor...
Dog-Konstruktor...
(1)Hund (2)Katze: 2
Mammal-Konstruktor...
Cat-Konstruktor...
(1)Hund (2)Katze: 1
Mammal-Konstruktor...
Dog-Konstruktor...

Wuff!
Oh nein, keine Katze!
Mammal-Destruktor...

Miau
rrrrrrrrrr
Mammal-Destruktor...

Wuff!
Oh nein, keine Katze!
Mammal-Destruktor...
```

Die Zeilen 39 bis 49 fragen den Benutzer, ob er entweder ein `Cat`- oder ein `Dog`-Objekt in das Array der `Mammal`-Zeiger aufnehmen möchte. Zeile 53 geht das Array durch und in Zeile 55 wird die virtuelle Methode `Speak()` eines jeden Objekts aufgerufen. Diese Methoden reagieren polymorph: Katzen miauen und Hunde sagen Wuff!

In Zeile 60 sollen die `Cat`-Objekte schnurren. Diese Methode soll aber nicht auf `Dog`-Objekten aufgerufen werden. In Zeile 57 stellt daher der Operator `dynamic_cast` sicher, dass der Aufruf von `Purr()` auf einem `Cat`-Objekt erfolgt. Ist dieses vorhanden, wird der Zeiger nicht `NULL` und besteht den Test in Zeile 59.

18.2 Abstrakte Datentypen

Häufig bildet man eine geschlossene Hierarchie von Klassen. Beispielsweise erzeugt man eine Klasse `Shape` (Form) und leitet davon `Rectangle` (Rechteck) und `Circle` (Kreis) ab. Als Spezialfall des Rechtecks lässt sich von `Rectangle` die Klasse `Square` (Quadrat) ableiten.

Alle abgeleiteten Klassen redefinieren die Methode `Draw()` (Zeichnen), die Methode `GetArea()` (Fläche ermitteln) usw. Listing 18.3 zeigt das Grundgerüst einer Implementierung der `Shape`-Klasse und die davon abgeleiteten Klassen `Circle` und `Rectangle`.

Listing 18.3: shapeclass.cpp – Shape-Klassen

```
 1: // Listing 18.3. Shape-Klassen
 2: #include <iostream>
 3:
 4: class Shape
 5: {
 6: public:
 7:     Shape(){}
 8:     virtual ~Shape(){}
 9:     virtual long GetArea() { return -1; } // Fehler
10:     virtual long GetPerim() { return -1; }
11:     virtual void Draw() {}
12: };
13:
14: class Circle : public Shape
15: {
16: public:
17:     Circle(int radius):itsRadius(radius){}
18:     ~Circle(){}
19:     long GetArea() { return 3 * itsRadius * itsRadius; }
20:     long GetPerim() { return 9 * itsRadius; }
21:     void Draw();
```

```
22: private:
23:     int itsRadius;
24:     int itsCircumference;
25: };
26:
27: void Circle::Draw()
28: {
29:     std::cout << "Routine zum Zeichnen eines Kreises \n";
30: }
31:
32:
33: class Rectangle : public Shape
34: {
35: public:
36:     Rectangle(int len, int width):
37:         itsLength(len), itsWidth(width){}
38:     virtual ~Rectangle(){}
39:     virtual long GetArea() { return itsLength * itsWidth; }
40:     virtual long GetPerim() {return 2*itsLength + 2*itsWidth; }
41:     virtual int GetLength() { return itsLength; }
42:     virtual int GetWidth() { return itsWidth; }
43:     virtual void Draw();
44: private:
45:     int itsWidth;
46:     int itsLength;
47: };
48:
49: void Rectangle::Draw()
50: {
51:     for (int i = 0; i<itsLength; i++)
52:     {
53:         for (int j = 0; j<itsWidth; j++)
54:             std::cout << "x ";
55:
56:         std::cout << "\n";
57:     }
58: }
59:
60: class Square : public Rectangle
61: {
62: public:
63:     Square(int len);
64:     Square(int len, int width);
65:     ~Square(){}
66:     long GetPerim() {return 4 * GetLength();}
67: };
68:
```

```
Square::Square(int len):
    Rectangle(len,len)
{}

Square::Square(int len, int width):
    Rectangle(len,width)
{
    if (GetLength() != GetWidth())
        std::cout << "Fehler, kein Quadrat... ein Rechteck??\n";
}

int main()
{
    int choice;
    bool fQuit = false;
    Shape * sp;

    while (1)
    {
        std::cout << "(1)Kreis (2)Rechteck (3)Quadrat (0)Ende: ";
        std::cin >> choice;

        switch (choice)
        {
        case 1:
            sp = new Circle(5);
            break;
        case 2:
             sp = new Rectangle(4,6);
            break;
        case 3:
            sp = new Square(5);
            break;
        default:
            fQuit = true;
            break;
        }
        if (fQuit)
            break;

        sp->Draw();
        std::cout << "\n";
    }
    return 0;
}
```

```
(1)Kreis (2)Rechteck (3)Quadrat (0)Ende: 2
x x x x x
x x x x x
x x x x x
x x x x x
(1)Kreis (2)Rechteck (3)Quadrat (0)Ende: 3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x
(1)Kreis (2)Rechteck (3)Quadrat (0)Ende: 0
```

Die Zeilen 4 bis 12 deklarieren die Klasse `Shape`. Die Methoden `GetArea()` und `GetPerim()` liefern einen Fehlerwert zurück und `Draw()` führt keine Aktionen aus. Was bedeutet es schließlich, eine Form zu zeichnen? Nur konkrete Formen (Kreise, Rechtecke usw.) lassen sich zeichnen, Formen als Abstraktion sind nicht darstellbar.

`Circle` leitet sich von `Shape` ab und redefiniert drei virtuelle Methoden. Beachten Sie, dass es keinen Grund gibt, das Schlüsselwort `virtual` hinzuzufügen, da es Bestandteil der Vererbung virtueller Methoden ist. Allerdings schadet es auch nichts, wenn man es trotzdem angibt, wie es in der Klasse `Rectangle` auf den Zeilen 39 bis 43 zu sehen ist. Es empfiehlt sich, den Begriff `virtual` als Erinnerung aufzunehmen – eine Art Dokumentation.

`Square` leitet sich von `Rectangle` ab, redefiniert ebenfalls die Methode `GetPerim()` und erbt die restlichen Methoden, die in `Rectangle` definiert sind.

Da es Probleme gibt, wenn ein Klient ein `Shape`-Objekt instanziert, sollte man dies verhindern. Die Klasse `Shape` existiert nur, um eine Schnittstelle für die davon abgeleiteten Klassen bereitzustellen. Als solches handelt es sich um einen *abstrakten Datentyp* (ADT).

Ein abstrakter Datentyp repräsentiert ein Konzept (wie `Shape`) und nicht ein Objekt (wie `Circle`). In C++ ist ein ADT immer die Basisklasse für andere Klassen und es ist nicht erlaubt, Instanzen eines ADT zu erzeugen. Wenn Sie also `Shape` zu einem abstrakten Datentyp machen, lässt sich kein `Shape`-Objekt instanzieren.

18.2.1 Abstrakte Funktionen

C++ unterstützt das Erstellen von abstrakten Datentypen mit abstrakten oder »rein virtuellen« Funktionen. Eine abstrakte Funktion ist eine virtuelle Funktion, die in der abgeleiteten Klasse überschrieben werden *muss*. Eine abstrakte Funktion erhält man, wenn man eine virtuelle Funktion mit `0` initialisiert:

```
virtual void Draw() = 0;
```

Jede Klasse mit einer oder mehreren abstrakten Funktionen ist ein abstrakter Datentyp. Es ist verboten, ein Objekt von einer als ADT fungierenden Klasse zu instanzieren. Der Versuch ist bereits strafbar und führt zu einem Compiler-Fehler. Bringt man eine abstrakte Funktion in einer Klasse unter, signalisiert man den Klienten der Klasse zwei Dinge:

- Erzeuge kein Objekt dieser Klasse, sondern leite von dieser Klasse ab.
- Redefiniere auf jeden Fall die abstrakte Funktion.

Von einem ADT abgeleitete Klassen erben die abstrakte Funktion in ihrer »reinen« Form. Aus diesem Grund muss man jede abstrakte Funktion redefinieren, wenn man Objekte instanzieren möchte. Wenn also `Rectangle` von `Shape` erbt und `Shape` über drei abstrakte Funktionen verfügt, muss `Rectangle` alle drei redefinieren – oder `Rectangle` bleibt ebenfalls ein ADT. In Listing 18.4 liegt die Klasse `Shape` in einer neuen Fassung als abstrakter Datentyp vor. Ersetzen Sie die Deklaration von `Shape` in den Zeilen 4 bis 12 von Listing 18.3 durch die Deklaration von `Shape` in Listing 18.4 (was Listing 18.5 ergibt) und starten Sie das Programm erneut.

Listing 18.4: adt.cpp – Abstrakte Datentypen

```
 1: class Shape
 2: {
 3: public:
 4:      Shape(){}
 5:      virtual ~Shape(){}
 6:      virtual long GetArea() = 0;
 7:      virtual long GetPerim()= 0;
 8:      virtual void Draw() = 0;
 9: private:
10: };
```

adt.cpp lässt sich nicht kompilieren!

Mühen Sie sich nicht damit ab, Listing 18.4 kompilieren zu wollen – es ist hier nur angegeben, um den geänderten Code zu zeigen. Listing 18.5 (das Listing 18.3 mit eingebautem Listing 18.4 entspricht) ist kompilierbar; probieren Sie dieses Programm aus.

Listing 18.5: adtshape-class.cpp – Die Klasse Shape mit abstrakten Datentypen

```
// Listing 18.5. Shape-Klassen mit abstrakten Datentypen
#include <iostream>

class Shape
{
public:
    Shape(){}
    virtual ~Shape(){}
    virtual long GetArea() = 0;
    virtual long GetPerim() = 0;
    virtual void Draw() = 0;
};

class Circle : public Shape
{
public:
    Circle(int radius):itsRadius(radius){}
    ~Circle(){}
    long GetArea() { return 3 * itsRadius * itsRadius; }
    long GetPerim() { return 9 * itsRadius; }
    void Draw();
private:
    int itsRadius;
    int itsCircumference;
};

void Circle::Draw()
{
    std::cout << "Zeichenroutine fuer Kreis an dieser Stelle!\n";
}


class Rectangle : public Shape
{
public:
    Rectangle(int len, int width):
        itsLength(len), itsWidth(width){}
    virtual ~Rectangle(){}
    virtual long GetArea() { return itsLength * itsWidth; }
    virtual long GetPerim() {return 2*itsLength + 2*itsWidth; }
    virtual int GetLength() { return itsLength; }
    virtual int GetWidth() { return itsWidth; }
    virtual void Draw();
private:
    int itsWidth;
    int itsLength;
};

```

```
49: void Rectangle::Draw()
50: {
51:     for (int i = 0; i<itsLength; i++)
52:     {
53:         for (int j = 0; j<itsWidth; j++)
54:             std::cout << "x ";
55:
56:         std::cout << "\n";
57:     }
58: }
59:
60: class Square : public Rectangle
61: {
62: public:
63:     Square(int len);
64:     Square(int len, int width);
65:     ~Square(){}
66:     long GetPerim() {return 4 * GetLength();}
67: };
68:
69: Square::Square(int len):
70:     Rectangle(len,len)
71: {}
72:
73: Square::Square(int len, int width):
74:     Rectangle(len,width)
75: {
76:     if (GetLength() != GetWidth())
77:         std::cout << "Fehler, kein Quadrat... ein Rechteck??\n";
78: }
79:
80: int main()
81: {
82:     int choice;
83:     bool fQuit = false;
84:     Shape * sp;
85:
86:     while (1)
87:     {
88:         std::cout << "(1)Kreis (2)Rechteck (3)Quadrat (0)Beenden: ";
89:         std::cin >> choice;
90:
91:         switch (choice)
92:         {
93:         case 1:
94:             sp = new Circle(5);
95:             break;
96:         case 2:
97:              sp = new Rectangle(4,6);
```

```
 98:             break;
 99:         case 3:
100:             sp = new Square(5);
101:             break;
102:         default:
103:             fQuit = true;
104:             break;
105:         }
106:         if (fQuit)
107:             break;
108:
109:         sp->Draw();
110:         std::cout << "\n";
111:     }
112:     return 0;
113: }
```

```
(1)Kreis (2)Rechteck (3)Quadrat (0)Beenden: 2
x x x x x
x x x x x
x x x x x
x x x x x
(1)Kreis (2)Rechteck (3)Quadrat (0)Beenden: 3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x
(1)Kreis (2)Rechteck (3)Quadrat (0)Beenden: 0
```

Wie Sie feststellen, hat sich die Arbeitsweise des Programms überhaupt nicht geändert. Der einzige Unterschied ist, dass man nun keine Objekte der Klasse `Shape` erzeugen kann.

Abstrakte Datentypen

Eine Klasse deklariert man als abstrakten Datentyp, indem man eine oder mehrere abstrakte oder »rein virtuelle« Funktionen in die Klassendeklaration aufnimmt. Eine abstrakte Funktion deklarieren Sie, indem Sie `= 0` hinter die Funktionsdeklaration schreiben.

Dazu ein Beispiel:

```
class Shape
{
    virtual void Draw() = 0;    // abstrakt
};
```

Abstrakte Funktionen implementieren

Normalerweise implementiert man abstrakte Funktionen in einer abstrakten Basisklasse überhaupt nicht. Da niemals Objekte dieses Typs erzeugt werden, gibt es keinen Anlass, Implementierungen bereitzustellen. Der ADT arbeitet damit ausschließlich als Definition einer Schnittstelle auf Objekte, die sich von ihm ableiten.

Es ist allerdings möglich, eine Implementierung für eine abstrakte Funktion anzugeben. Die Funktion lässt sich dann von Objekten aufrufen, die vom ADT abgeleitet sind, etwa um allen redefinierten Funktionen eine gemeinsame Funktionalität zu verleihen. Listing 18.6 bringt eine Neuauflage von Listing 18.3, realisiert `Shape` aber jetzt als ADT und mit einer Implementierung für die abstrakte Funktion `Draw()`. Die Klasse `Circle` redefiniert `Draw()` wie erforderlich, holt sich dann aber zusätzliche Funktionalität über die Funktion der Basisklasse.

In diesem Beispiel besteht die zusätzliche Funktionalität einfach in einer weiteren Meldung. Die Basisklasse könnte aber auch einen gemeinsam genutzten Zeichenmechanismus bereitstellen, der zum Beispiel ein Fenster einrichtet, mit dem alle abgeleiteten Klassen arbeiten.

Listing 18.6: implement.cpp – Abstrakte Funktionen implementieren

```
// Listing 18.6. Abstrakte Funktionen implementieren
#include <iostream>

class Shape
{
public:
    Shape(){}
    virtual ~Shape(){}
    virtual long GetArea() = 0;
     virtual long GetPerim()= 0;
    virtual void Draw() = 0;
private:
};

void Shape::Draw()
{
    std::cout << "Abstrakter Zeichenmechanismus!\n";
}

class Circle : public Shape
{
public:
    Circle(int radius):itsRadius(radius){}
    ~Circle(){}
    long GetArea() { return 3 * itsRadius * itsRadius; }
    long GetPerim() { return 9 * itsRadius; }
```

```
27:     void Draw();
28: private:
29:     int itsRadius;
30:     int itsCircumference;
31: };
32:
33: void Circle::Draw()
34: {
35:     std::cout << "Zeichenroutine fuer Kreis an dieser Stelle!\n";
36:     Shape::Draw();
37: }
38:
39: class Rectangle : public Shape
40: {
41: public:
42:     Rectangle(int len, int width):
43:         itsLength(len), itsWidth(width){}
44:     virtual ~Rectangle(){}
45:     long GetArea() { return itsLength * itsWidth; }
46:     long GetPerim() {return 2*itsLength + 2*itsWidth; }
47:     virtual int GetLength() { return itsLength; }
48:     virtual int GetWidth() { return itsWidth; }
49:     void Draw();
50: private:
51:     int itsWidth;
52:     int itsLength;
53: };
54:
55: void Rectangle::Draw()
56: {
57:     for (int i = 0; i<itsLength; i++)
58:     {
59:         for (int j = 0; j<itsWidth; j++)
60:             std::cout << "x ";
61:          std::cout << "\n";
62:     }
63:     Shape::Draw();
64: }
65:
66: class Square : public Rectangle
67: {
68: public:
69:     Square(int len);
70:     Square(int len, int width);
71:     ~Square(){}
72:     long GetPerim() {return 4 * GetLength();}
73: };
74:
```

```
 75: Square::Square(int len):
 76:     Rectangle(len,len)
 77: {}
 78:
 79: Square::Square(int len, int width):
 80:     Rectangle(len,width)
 81: {
 82:     if (GetLength() != GetWidth())
 83:         std::cout << "Fehler, kein Quadrat... ein Rechteck??\n";
 84: }
 85:
 86: int main()
 87: {
 88:     int choice;
 89:     bool fQuit = false;
 90:     Shape * sp;
 91:
 92:     while (1)
 93:     {
 94:         std::cout << "(1)Kreis (2)Rechteck (3)Quadrat (0)Beenden: ";
 95:         std::cin >> choice;
 96:
 97:         switch (choice)
 98:         {
 99:         case 1:
100:             sp = new Circle(5);
101:             break;
102:         case 2:
103:             sp = new Rectangle(4,6);
104:             break;
105:         case 3:
106:             sp = new Square (5);
107:             break;
108:         default:
109:             fQuit = true;
110:             break;
111:         }
112:         if (fQuit)
113:             break;
114:         sp->Draw();
115:         std::cout << "\n";
116:     }
117:     return 0;
118: }
```

```
(1)Kreis (2)Rechteck (3)Quadrat (0)Beenden: 2
x x x x x
x x x x x
x x x x x
x x x x x
Abstrakter Zeichenmechanismus!

(1)Kreis (2)Rechteck (3)Quadrat (0)Beenden: 3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x
Abstrakter Zeichenmechanismus!

(1)Kreis (2)Rechteck (3)Quadrat (0)Beenden: 0
```

Die Zeilen 4 bis 13 deklarieren den abstrakten Datentyp `Shape`. Hier sind alle drei Zugriffsmethoden als abstrakt deklariert – eine hätte auch eine genügt. Wenn irgendeine Methode als abstrakt deklariert ist, wird die gesamte Klasse zu einem ADT.

Die Methoden `GetArea()` und `GetPerim()` sind nicht implementiert, `Draw()` hat dagegen eine Implementierung. `Circle` und `Rectangle` redefinieren `Draw()`; beide Klassen rufen die Basismethode auf, um gemeinsam genutzte Funktionalität aus der Basisklasse zu übernehmen.

18.2.2 Komplexe Abstraktionshierarchien

Manchmal leitet man ADTs von anderen ADTs ab, etwa um einige der abgeleiteten Funktionen »nicht rein« zu machen und andere »rein« zu belassen.

Wenn man die Klasse `Animal` erzeugt, kann man `Eat()` (Essen), `Sleep()` (Schlafen), `Move()` (Bewegen) und `Reproduce()` (Fortpflanzen) als abstrakte Funktionen deklarieren. Von `Animal` leitet man vielleicht `Mammal` und `Fish` ab.

Da sich die Säugetiere in der gleichen Weise fortpflanzen, deklarieren Sie `Mammal::Reproduce()` als »nicht rein«, belassen aber `Eat()`, `Sleep()` und `Move()` als abstrakte Funktionen.

Von `Mammal` leiten Sie `Dog` ab und `Dog` muss die drei restlichen abstrakten Funktionen redefinieren und implementieren, damit sich Objekte vom Typ `Dog` erzeugen lassen.

Als Klassenentwickler haben Sie damit gesagt, dass sich keine `Animals` oder `Mammals` instanzieren lassen, dass aber alle `Mammals` die bereitgestellte Methode `Reproduce()` erben können, ohne sie zu redefinieren.

Listing 18.7 zeigt diese Technik mit einer gerüstartigen Implementierung dieser Klassen.

Listing 18.7: deriving-adt.cpp – ADTs von anderen ADTs ableiten

```
// Listing 18.7
// ADTs von anderen ADTs ableiten
#include <iostream>

enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown } ;

class Animal  // Gemeinsame Basisklasse sowohl für Horse als
auch Fish
{
public:
    Animal(int);
    virtual ~Animal() { std::cout << "Animal-Destruktor...\n"; }
    virtual int GetAge() const { return itsAge; }
    virtual void SetAge(int age) { itsAge = age; }
    virtual void Sleep() const = 0;
    virtual void Eat() const = 0;
    virtual void Reproduce() const = 0;
    virtual void Move() const = 0;
    virtual void Speak() const = 0;
private:
    int itsAge;
};

Animal::Animal(int age):
itsAge(age)
{
    std::cout << "Animal-Konstruktor...\n";
}

class Mammal : public Animal
{
public:
    Mammal(int age):Animal(age)
        { std::cout << "Mammal-Konstruktor...\n";}
    virtual ~Mammal() { std::cout << "Mammal-Destruktor...\n";}
    virtual void Reproduce() const
        { std::cout << "Mammal Fortpflanzung...\n"; }
};

class Fish : public Animal
{
public:
```

```
42:     Fish(int age):Animal(age)
43:         { std::cout << "Fish-Konstruktor...\n";}
44:     virtual ~Fish()
45:         { std::cout << "Fish-Destruktor...\n";  }
46:     virtual void Sleep() const
47:         { std::cout << "Fisch schlummert...\n"; }
48:     virtual void Eat() const
49:         { std::cout << "Fisch frisst...\n"; }
50:     virtual void Reproduce() const
51:         { std::cout << "Fisch legt Eier...\n"; }
52:     virtual void Move() const
53:         { std::cout << "Fisch schwimmt...\n";   }
54:     virtual void Speak() const { }
55: };
56:
57: class Horse : public Mammal
58: {
59: public:
60:     Horse(int age, COLOR color ):
61:         Mammal(age), itsColor(color)
62:         { std::cout << "Horse-Konstruktor...\n"; }
63:     virtual ~Horse()
64:         { std::cout << "Horse-Destruktor...\n"; }
65:     virtual void Speak()const
66:         { std::cout << "Wieher!... \n"; }
67:     virtual COLOR GetItsColor() const
68:         { return itsColor; }
69:     virtual void Sleep() const
70:         { std::cout << "Pferd schlaeft...\n"; }
71:     virtual void Eat() const
72:         { std::cout << "Pferd frisst...\n"; }
73:     virtual void Move() const
74:         { std::cout << "Pferd laeuft...\n";}
75:
76: protected:
77:     COLOR itsColor;
78: };
79:
80: class Dog : public Mammal
81: {
82: public:
83:     Dog(int age, COLOR color ):
84:         Mammal(age), itsColor(color)
85:         { std::cout << "Dog-Konstruktor...\n"; }
86:     virtual ~Dog()
87:         { std::cout << "Dog-Destruktor...\n"; }
88:     virtual void Speak()const
89:         { std::cout << "Wuff!... \n"; }
```

```
 90:     virtual void Sleep() const
 91:         { std::cout << "Hund schlaeft...\n"; }
 92:     virtual void Eat() const
 93:         { std::cout << "Hund frisst...\n"; }
 94:     virtual void Move() const
 95:         { std::cout << "Hund laeuft...\n"; }
 96:     virtual void Reproduce() const
 97:         { std::cout << "Hunde pflanzen sich fort...\n"; }
 98:
 99: protected:
100:     COLOR itsColor;
101: };
102:
103: int main()
104: {
105:     Animal *pAnimal=0;
106:     int choice;
107:     bool fQuit = false;
108:
109:     while (1)
110:     {
111:         std::cout << "(1)Hund (2)Pferd (3)Fisch (0)Beenden: ";
112:         std::cin >> choice;
113:
114:         switch (choice)
115:         {
116:         case 1:
117:             pAnimal = new Dog(5,Brown);
118:             break;
119:         case 2:
120:             pAnimal = new Horse(4,Black);
121:             break;
122:         case 3:
123:             pAnimal = new Fish (5);
124:             break;
125:         default:
126:             fQuit = true;
127:             break;
128:         }
129:         if (fQuit)
130:             break;
131:
132:         pAnimal->Speak();
133:         pAnimal->Eat();
134:         pAnimal->Reproduce();
135:         pAnimal->Move();
136:         pAnimal->Sleep();
137:         delete pAnimal;
```

```
138:         std::cout << "\n";
139:     }
140:     return 0;
141: }
```

```
(1)Hund (2)Pferd (3)Fisch (0)Beenden: 1
Animal-Konstruktor...
Mammal-Konstruktor...
Dog-Konstruktor...
Wuff!...
Hund frisst...
Hunde pflanzen sich fort....
Hund laeuft...
Hund schlaeft...
Dog-Destruktor...
Mammal-Destruktor...
Animal-Destruktor...

(1)Hund (2)Pferd (3)Fisch (0)Beenden: 0
```

Die Zeilen 7 bis 21 deklarieren den abstrakten Datentyp `Animal`. Die virtuellen Zugriffsfunktionen für `itsAge` sind nicht abstrakt. Alle `Animal`-Objekte nutzen diese Funktionen gemeinsam. Weiterhin gibt es die fünf abstrakten Funktionen `Sleep()`, `Eat()`, `Reproduce()`, `Move()` und `Speak()`.

Die Deklaration der von `Animal` abgeleiteten Klasse `Mammal` steht in den Zeilen 29 bis 37. Die Klasse fügt keine neuen Daten hinzu. Allerdings redefiniert `Mammal` die Funktion `Reproduce()`, um eine allgemeine Form der Fortpflanzung für alle Säugetiere bereitzustellen. `Fish` muss `Reproduce()` redefinieren, da sich `Fish` direkt von `Animal` ableitet und nicht die Fortpflanzung von Säugetieren nutzen kann. (Logisch, oder?)

`Mammal`-Klassen brauchen nun nicht mehr die Funktion `Reproduce()` zu redefinieren, können es aber bei Bedarf tun, wie zum Beispiel `Dog` in den Zeilen 96 und 97. `Fish`, `Horse` und `Dog` redefinieren die restlichen abstrakten Funktionen, sodass sich Objekte des jeweiligen Typs instanzieren lassen.

Der Rumpf des Programms verwendet einen `Animal`-Zeiger, um auf die verschiedenartigen abgeleiteten Objekte der Reihe nach zu verweisen. Beim Aufruf der virtuellen Methoden wird die jeweils richtige Methode in der abgeleiteten Klasse gemäß der Laufzeitbindung des Zeigers aufgerufen.

Versucht man, ein `Animal` oder ein `Mammal` zu instanzieren, erhält man einen Compiler-Fehler, da beides abstrakte Datentypen sind.

18.2.3 Welche Typen sind abstrakt?

In einem Programm ist die Klasse `Animal` abstrakt, in einem anderen nicht. Woraus ergibt sich, ob eine Klasse abstrakt zu machen ist?

Diese Frage lässt sich nicht global beantworten. Es hängt immer davon ab, was in dem jeweiligen Programm sinnvoll ist. In einem Programm für einen Bauernhof oder einen Zoo deklariert man `Animal` zum Beispiel als abstrakten Datentyp und `Dog` als Klasse, von der man Objekte instanzieren kann.

Wenn man andererseits einen Hundezwinger erstellt, kann man `Dog` als abstrakten Datentyp deklarieren und nur Hunderassen instanzieren: Spürhunde, Terrier usw. Die Abstraktionsebene ist davon abhängig, wie fein man die Typen unterscheiden muss.

Was Sie tun sollten

Verwenden Sie abstrakte Datentypen, um gemeinsame Funktionalität für mehrere verwandte Klassen bereitzustellen.

Redefinieren Sie alle abstrakten Funktionen.

Deklarieren Sie alle Funktionen als abstrakt, die zu redefinieren sind.

Achtung

Versuchen Sie nicht, ein Objekt eines abstrakten Datentyps zu instanzieren.

18.3 Fragen und Antworten

F: Was bedeutet das Delegieren der Funktionalität nach oben?

A: Es handelt sich um eine Verlagerung der Funktionalität in eine darüber liegende, gemeinsame Basisklasse. Wenn mehrere Klassen eine Funktion gemeinsam nutzen, ist es sinnvoll, eine gemeinsame Basisklasse zu suchen, in der sich diese Funktion unterbringen lässt.

F: Ist das Delegieren nach oben immer empfehlenswert?

A: Ja, wenn man gemeinsam genutzte Funktionalität nach oben verschiebt. Nein, wenn man lediglich die Schnittstelle nach oben bringt. Das heißt, wenn nicht alle abgeleiteten Klassen die Methode verwenden können, ist es ein Missgriff, diese Funktion in eine gemeinsam genutzte Basisklasse zu verlagern. Wenn man es doch tut, muss man den Laufzeittyp des Objekts aktivieren, bevor man entscheidet, ob man die Funktion aufrufen kann.

F: Warum ist dynamische Typumwandlung nicht zu empfehlen?

A: Das Grundanliegen der virtuellen Funktionen besteht darin, die virtuelle Tabelle und nicht den Programmierer über den Laufzeittyp des Objekts entscheiden zu lassen.

F: Warum soll man sich mit abstrakten Datentypen herumschlagen? Man könnte doch die nicht abstrakte Form beibehalten und einfach das Erzeugen von Objekten dieses Typs vermeiden.

A: Der Zweck vieler Konventionen in C++ besteht darin, den Compiler bei der Fehlersuche heranzuziehen, um Laufzeitfehler aus dem Code (den man schließlich verkaufen möchte) zu verbannen. Wenn man eine Klasse mit abstrakten Funktionen ausstattet und damit abstrakt macht, weist der Compiler mit einer entsprechenden Fehlermeldung auf alle Objekte hin, die man aus diesem abstrakten Typ erzeugen möchte.

18.4 Workshop

Nachdem Sie sich weiter gehend mit Polymorphie auseinander gesetzt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu festigen.

18.4.1 Kontrollfragen

1. Worin liegt der Unterschied zwischen einer virtuellen Methode und einer überschriebenen Methode in der Basisklasse?
2. Worin liegt der Unterschied zwischen einer »reinen« und einer »nicht reinen« virtuellen Methode?
3. Weshalb sind abstrakte Datentypen »abstrakt«?
4. Was ist falsch daran, in der Basisklasse Methoden bereitzustellen, die nur für bestimmte abgeleitete Klassen relevant sind?

18.4.2 Übungen

1. Ändern Sie Listing 18.7 (`derivingadt.cpp`), um ein Objekt des Typs `Animal` oder `Mammal` zu instanzieren. Was sagt Ihnen der Compiler und warum?
2. Stellen Sie sich eine Geschäftsanwendung vor und versuchen Sie, abstrakte Datentypen dafür herauszuarbeiten. *Hinweis:* Bankkonten sind sehr allgemein gehalten und haben gemeinsame Verhaltensweisen, es gibt aber unterschiedliche Arten von Konten.

3. Was passiert, wenn Sie in Listing 18.2 (`dynamiccast.cpp`) den `if`-Test in Zeile 59 (zusammen mit den Zeilen 61 und 62) entfernen? Welche Objekte arbeiten korrekt und welche Objekte scheitern?

18.4.3 Antworten zu den Kontrollfragen

1. Eine virtuelle Funktion ist in den abgeleiteten Klassen zu überschreiben.
2. Eine Klasse mit »nicht reinen« virtuellen Methoden lässt sich instanzieren. Abstrakte (»rein virtuelle«) Methoden enthalten keinen ausführbaren Code – ein Compiler-Fehler ist das Ergebnis, wenn Sie versuchen, eine derartige Klasse zu instanzieren. Eine abstrakte Funktion muss überschrieben werden.
3. Diese Datentypen sind deshalb abstrakt, weil sie reine virtuelle Methoden enthalten und selbst nicht instanziert werden können. Von dieser Klasse lässt sich ein konkretes Beispiel erzeugen – man muss eine Klasse von der abstrakten Klasse mit den richtigen Methoden ableiten und eine Instanz von der abgeleiteten Klasse erzeugen.
4. Offen gesagt zeigt sich hier ein schlechtes Design. Theoretisch sollten die Methoden der Basisklasse für die meisten – wenn nicht sogar alle – abgeleiteten Klassen relevant sein. Man leitet von der Basisklasse ab, um sie zu erweitern, und nicht, um irgendwelche Teile von ihr vollkommen unter den Tisch fallen zu lassen.

KAPITEL 19

jetzt lerne ich

Verkettete Listen

In dieser Stunde lernen Sie,

- was eine verkettete Liste ist,
- wie man eine verkettete Liste erzeugt,
- wie man die Funktionalität durch Vererbung kapselt.

19.1 Verkettete Listen und andere Strukturen

Arrays lassen sich in vielerlei Hinsicht mit Tupperware vergleichen. Es sind hervorragende Behälter, sie besitzen aber eine feste Größe. Wählt man einen zu großen Behälter, verschwendet man Platz im Kühlschrank. Nimmt man einen zu kleinen Behälter, quillt der Inhalt über.

Als Problemlösung bietet sich eine *verkettete Liste* an. Dabei handelt es sich um eine Datenstruktur aus kleinen Behältern, die aneinander »gekoppelt« werden. Behälter in diesem Sinne sind Klassen, die die Objekte enthalten, die in der Liste zu speichern sind. Der Grundgedanke besteht darin, eine Klasse zu schreiben, die genau ein Objekt der Daten enthält – etwa ein `CAT`- oder ein `Rectangle`-Objekt – und auf den nächsten Behälter in der Liste zeigen kann. Man erzeugt einen Behälter für jedes zu speichernde Objekt und verkettet die Behälter nach Bedarf.

Diese kleinen Behälter heißen *Knoten*. Den ersten Knoten in der Liste bezeichnet man als *Kopf*.

Man unterscheidet bei Listen drei grundlegende Formen. Von der einfachsten zur komplexesten sind das

- einfach verkettete Listen,
- doppelt verkettete Listen,
- Bäume.

In einer *einfach verketteten Liste* zeigt jeder Knoten auf den nächsten, jedoch nicht auf den vorhergehenden. Um einen bestimmten Knoten zu finden, beginnt man am Anfang der Liste und tastet sich wie bei einer Schnitzeljagd (»der nächste Knoten liegt unter dem Sofa«) von Knoten zu Knoten. Eine *doppelt verkettete Liste* kann man sowohl in Vorwärts- als auch in Rückwärtsrichtung durchlaufen. Ein *Baum* ist eine komplexe Struktur, die sich aus Knoten aufbaut. Jeder Knoten kann in zwei oder drei Richtungen zeigen. Abbildung 19.1 zeigt diese drei fundamentalen Strukturen.

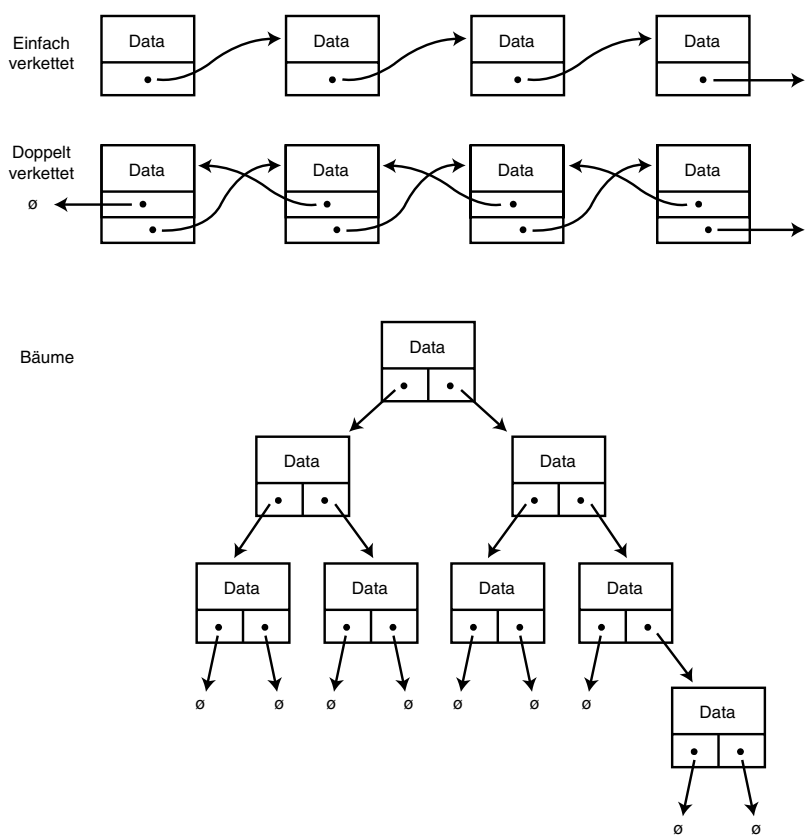


Abb. 19.1: Verkettete Listen

Informatiker haben noch komplexere und raffiniertere Datenstrukturen entwickelt, die nahezu alle auf untereinander verbundenen Knoten beruhen.

19.2 Eine Fallstudie

In diesem Abschnitt untersuchen wir anhand einer Fallstudie eine verkettete Liste im Detail. Zum einen geht es darum, wie man komplexe Strukturen erzeugt, zum anderen, wie man große Projekte mit Vererbung, Polymorphie und Kapselung verwaltet.

19.2.1 Delegierung von Verantwortlichkeit

Eine grundlegende Prämisse der objektorientierten Programmierung besteht darin, dass jedes Objekt genau *eine* Aufgabe erledigt und alles, was nicht zu seinem »Kerngeschäft« gehört, an andere Objekte delegiert.

Ein Auto ist ein gutes Beispiel für dieses Konzept: Die Aufgabe des Motors besteht in der Leistungserzeugung. Die Verteilung dieser Leistung gehört nicht zu den Aufgaben des Motors, sondern kommt den Einrichtungen der Kraftübertragung zu. Die Rollbewegung realisieren weder der Motor noch die Kraftübertragung, diese Aufgabe wird an die Räder weitergereicht.

Eine gut konzipierte Maschine besteht aus einer Menge kleiner Teile mit genau umrissenen Aufgaben, die in ihrer Gesamtheit die vorgesehene Funktionalität realisieren. Das Gleiche gilt für ein gut konzipiertes Programm: jede Klasse arbeitet still vor sich hin, zusammengenommen bilden sie die »eierlegende Wollmilchsau«.

19.3 Die Einzelkomponenten

Die verkettete Liste besteht aus Knoten. Die Knotenklasse selbst ist abstrakt. Die Arbeit realisieren wir mit drei Untertypen. Es gibt einen Kopfknoten, der den Kopf der Liste verwaltet, einen Endeknoten und null oder mehr interne Knoten. Die internen Knoten verfolgen die eigentlich in die Liste aufzunehmenden Daten.

Beachten Sie, dass die Daten und die Liste zwei verschiedene Paar Schuhe sind. Man kann theoretisch jeden beliebigen Datentyp in einer Liste speichern. Nicht die Daten sind verkettet, sondern die Knoten, die die Daten aufnehmen.

Das Rahmenprogramm bekommt von den Knoten nichts mit. Es arbeitet nur mit der Liste. Die Liste führt allerdings kaum Aufgaben aus, sondern delegiert sie einfach an die Knoten.

Listing 19.1 zeigt den Code, den wir eingehend untersuchen werden.

Listing 19.1: linkedlist.cpp – Eine verkettete Liste

```
// ***********************************************
//           Listing 19.1
//
//        Demonstriert verkettete Liste
//
//
//
//
//
// Zeigt objektorientierte Lösung für verkettete Listen.
// Die Liste delegiert die eigentliche Arbeit an die Knoten
// Die Knoten sind abstrakte Datentypen. Es gibt drei
// Knotentypen: Kopfknoten, Endeknoten und interne Knoten.
// Nur die internen Knoten nehmen Daten auf.
//
// Die Klasse Data dient als Objekt, das in der
// verketteten Liste gespeichert wird.
//
// ***********************************************
#include <iostream>

enum { kIsSmaller, kIsLarger, kIsSame};

// Data-Klasse für die Speicherung in der Liste. Jede
// Klasse in dieser verketteten Liste muss zwei Methoden
// unterstützen: Show (zeigt den Wert an) und Compare
// (gibt die relative Position zurück).
class Data
{
public:
    Data(int val):myValue(val){}
    ~Data(){}
    int Compare(const Data &);
    void Show() { std::cout << myValue << std::endl; }
private:
    int myValue;
};

// Compare entscheidet, wohin ein bestimmtes Objekt
// in der Liste gehört.
int Data::Compare(const Data & theOtherData)
{
    if (myValue < theOtherData.myValue)
        return kIsSmaller;
    if (myValue > theOtherData.myValue)
        return kIsLarger;
```

```
47:     else
48:         return kIsSame;
49: }
50:
51: // Vorwärtsdeklarationen
52: class Node;
53: class HeadNode;
54: class TailNode;
55: class InternalNode;
56:
57: // ADT, der das Knotenobjekt in der Liste darstellt. Alle
58: // abgeleiteten Klassen müssen Insert und Show redefinieren.
59: class Node
60: {
61: public:
62:     Node(){}
63:     virtual ~Node(){}
64:     virtual Node * Insert(Data * theData)=0;
65:     virtual void Show() = 0;
66: private:
67: };
68:
69: // Dieser Knoten nimmt das eigentliche Objekt auf.
70: // Hier hat das Objekt den Typ Data.
71: // Bei der Behandlung von Templates wird eine
72: // Verallgemeinerung vorgestellt.
73: class InternalNode: public Node
74: {
75: public:
76:     InternalNode(Data * theData, Node * next);
77:     virtual ~InternalNode(){ delete myNext; delete myData; }
78:     virtual Node * Insert(Data * theData);
79:     virtual void Show()
80:         { myData->Show(); myNext->Show(); } // delegieren!
81:
82: private:
83:     Data * myData;    // die eigentlichen Daten
84:     Node * myNext;    // zeigt auf nächsten Knoten in der Liste
85: };
86:
87: // Der Konstruktor führt nur Initialisierungen aus.
88: InternalNode::InternalNode(Data * theData, Node * next):
89: myData(theData),myNext(next)
90: {
91: }
92:
93: // Der Kern der Listenkonstruktion. Stellt man ein
94: // neues Objekt in die Liste, wird es an den Knoten
```

```
// weitergereicht, der ermittelt, wohin das Objekt
// gehört, und es in die Liste einfügt.
Node * InternalNode::Insert(Data * theData)
{

    // Ist das neue Objekt größer oder kleiner als ich?
    int result = myData->Compare(*theData);


    switch(result)
    {
    // Ist das neue Objekt gleich groß, kommt es vor das aktuelle.
    case kIsSame:      // zum nächsten case-Zweig "durchfallen"
    case kIsLarger:    // neue Daten vor mir einordnen
        {
            InternalNode * dataNode =
                new InternalNode(theData, this);
            return dataNode;
        }

        // Größer als ich, also an den nächsten Knoten
        // weiterreichen. ER soll sich darum kümmern.
    case kIsSmaller:
        myNext = myNext->Insert(theData);
        return this;
    }
    return this;  // Tribut an Compiler
}


// Endeknoten ist einfach eine Markierung
class TailNode : public Node
{
public:
    TailNode(){}
    virtual ~TailNode(){}
    virtual Node * Insert(Data * theData);
    virtual void Show() { }
private:
};

// Wenn Daten zu mir kommen, müssen sie vor mir eingefügt werden,
// da ich der Endeknoten bin und NICHTS nach mir kommt.
Node * TailNode::Insert(Data * theData)
{
    InternalNode * dataNode = new InternalNode(theData, this);
    return dataNode;
}

```

```
144: // Kopfknoten hat keine Daten, sondern zeigt einfach
145: // auf den Beginn der Liste.
146: class HeadNode : public Node
147: {
148: public:
149:     HeadNode();
150:     virtual ~HeadNode() { delete myNext; }
151:     virtual Node * Insert(Data * theData);
152:     virtual void Show() { myNext->Show(); }
153: private:
154:     Node * myNext;
155: };
156:
157: // Nach Erzeugen des Kopfknotens erzeugt dieser
158: // sofort den Endeknoten.
159: HeadNode::HeadNode()
160: {
161:     myNext = new TailNode;
162: }
163:
164: // Vor dem Kopf kommt nichts, also die Daten einfach
165: // an den nächsten Knoten weiterreichen
166: Node * HeadNode::Insert(Data * theData)
167: {
168:     myNext = myNext->Insert(theData);
169:     return this;
170: }
171:
172: // Ich stehe im Mittelpunkt, mache aber gar nichts.
173: class LinkedList
174: {
175: public:
176:     LinkedList();
177:     ~LinkedList() { delete myHead; }
178:     void Insert(Data * theData);
179:     void ShowAll() { myHead->Show(); }
180: private:
181:     HeadNode * myHead;
182: };
183:
184: // Bei Geburt erzeuge ich den Kopfknoten.
185: // Er erzeugt den Endeknoten.
186: // Eine leere Liste zeigt damit auf den Kopf, dieser
187: // zeigt auf das Ende, dazwischen ist nichts.
188: LinkedList::LinkedList()
189: {
190:     myHead = new HeadNode;
191: }
```

```
192:
193: // Delegieren, delegieren, delegieren
194: void LinkedList::Insert(Data * pData)
195: {
196:     myHead->Insert(pData);
197: }
198:
199: // Rahmenprogramm zum Testen
200: int main()
201: {
202:     Data * pData;
203:     int val;
204:     LinkedList ll;
205:
206:     // Benutzer zum Erzeugen von Werten auffordern.
207:     // Diese Werte in die Liste stellen.
208:     for (;;)
209:     {
210:         std::cout << "Welcher Wert? (0 zum Beenden): ";
211:         std::cin >> val;
212:         if (!val)
213:             break;
214:         pData = new Data(val);
215:         ll.Insert(pData);
216:     }
217:
218:     // Jetzt Liste durchlaufen und Daten anzeigen.
219:     ll.ShowAll();
220:     return 0;  // ll verliert Gültigkeitsbereich und wird abgebaut!
221: }
```

```
Welcher Wert? (0 zum Beenden): 5
Welcher Wert? (0 zum Beenden): 8
Welcher Wert? (0 zum Beenden): 3
Welcher Wert? (0 zum Beenden): 9
Welcher Wert? (0 zum Beenden): 2
Welcher Wert? (0 zum Beenden): 10
Welcher Wert? (0 zum Beenden): 0
2
3
5
8
9
10
```

Als Erstes fällt eine Aufzählungskonstante auf, die drei konstante Werte bereitstellt: `kIsSmaller`, `kIsLarger` und `kIsSame` (k ist kleiner, größer bzw. gleich). Jedes Objekt, das sich in dieser verketteten Liste speichern lässt, muss eine `Compare()`-Methode (Vergleichen) unterstützen. Diese Konstanten repräsentieren den von der Methode `Compare()` zurückgegebenen Wert.

Die Zeilen 28 bis 37 erzeugen zu Demonstrationszwecken die Klasse `Data`. Die Implementierung der Methode `Compare` steht in den Zeilen 41 bis 49. Ein `Data`-Objekt nimmt einen Wert auf und kann sich selbst mit anderen `Data`-Objekten vergleichen. Außerdem unterstützt es eine Methode `Show()`, um den Wert des `Data`-Objekts anzuzeigen.

Am besten lässt sich die Arbeitsweise der verketteten Liste anhand eines Beispiels verstehen. Zeile 200 deklariert ein Rahmenprogramm, Zeile 201 einen Zeiger auf ein `Data`-Objekt und Zeile 203 definiert eine lokale verkettete Liste.

Beim Erzeugen der verketteten Liste wird der Konstruktor in Zeile 188 aufgerufen. Der Konstruktor hat einzig die Aufgabe, ein `HeadNode`-Objekt (Kopfknoten) zu reservieren und die Adresse dieses Objekts dem in der verketteten Liste gespeicherten Zeiger in Zeile 181 zuzuweisen.

Diese Reservierung eines `HeadNode`-Objekts ruft den in Zeile 159 dargestellten `HeadNode`-Konstruktor auf. Dieser Aufruf wiederum reserviert ein `TailNode`-Objekt (Endeknoten) und weist dessen Adresse dem Zeiger `myNext` (Zeiger auf nächsten Knoten) des Kopfknotens zu. Die Erzeugung des `TailNode`-Objekts ruft den `TailNode`-Konstruktor auf (Zeile 129), der inline deklariert ist und keine Aktionen ausführt.

Durch einfaches Reservieren einer verketteten Liste auf dem Stack werden somit die Liste erzeugt, die Kopf- und Endeknoten erstellt sowie deren Beziehungen eingerichtet. Die sich daraus ergebende Struktur zeigt Abbildung 19.2.

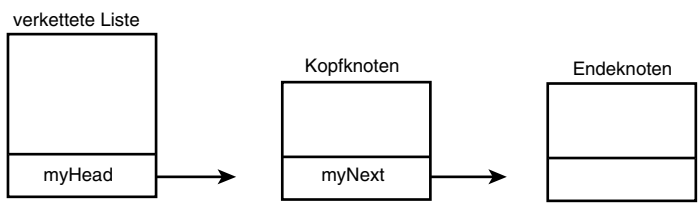


Abb. 19.2: Die verkettete Liste nach der Erzeugung

Zeile 208 leitet eine Endlosschleife ein. Der Benutzer gibt jetzt Werte ein, die in die verkettete Liste aufgenommen werden. Er kann beliebig viele Werte eingeben, mit einer `0` zeigt er das Ende der Eingabe an. Der Code in Zeile 212 wertet die eingegebenen Daten aus. Bei einer `0` verlässt das Programm die Schleife.

Ist der Wert ungleich `0`, erzeugt Zeile 213 ein neues `Data`-Objekt , das die Anweisung auf Zeile 215 in die Liste einfügt. Nehmen wir zu Demonstrationszwecken an, dass der Benutzer den Wert `15` eingibt. Das bewirkt den Aufruf der Methode `Insert()`in Zeile 194.

Die verkettete Liste delegiert sofort die Verantwortlichkeit für das Einfügen des Objekts an ihren Kopfknoten. Das führt zum Aufruf der Methode `Insert` in Zeile 168. Der Kopfknoten übergibt die Verantwortlichkeit sofort an den Knoten, auf den `myNext` momentan zeigt. In diesem (ersten) Fall zeigt er auf den Endeknoten. (Erinnern Sie sich: Beim Erzeugen des Kopfknotens hat dieser einen Verweis auf einen Endeknoten angelegt.) Das zieht den Aufruf der Methode `Insert()` in Zeile 138 nach sich.

Die Methode `TailNode::Insert()` weiß, dass das übernommene Objekt unmittelbar vor sich selbst einzufügen ist – das heißt, das neue Objekt kommt direkt vor dem Endeknoten in die Liste. Daher erzeugt die Methode in Zeile 140 ein neues `InternalNode`-Objekt und übergibt die Daten und den Zeiger auf sich selbst. Das führt zum Aufruf des Konstruktors für das `InternalNode`-Objekt, wie Zeile 88 zeigt.

Der `InternalNode`-Konstruktor initialisiert lediglich seinen `Data`-Zeiger mit der Adresse des übergebenen `Data`-Objekts und seinen `myNext`-Zeiger mit der Adresse des übergebenen Knotens. In diesem Fall zeigt der Knoten auf den Endeknoten. (Der Endeknoten hat seinen eigenen `this`-Zeiger übergeben.)

Nachdem nun der `InternalNode`-Knoten erstellt ist, wird die Adresse dieses internen Knotens in Zeile 140 an den Zeiger `dataNode` zugewiesen und diese Adresse wird wiederum von der Methode `TailNode::Insert()` zurückgegeben. Das bringt uns zu `HeadNode::Insert()` zurück, wo die Adresse des `InternalNode`-Knotens dem Zeiger `myNext` von `HeadNode` (in Zeile 168) zugewiesen wird. Schließlich wird die Adresse von `HeadNode` an die verkettete Liste zurückgegeben und (in Zeile 196) verworfen. (Das Programm stellt nichts damit an, da die verkettete Liste bereits die Adresse des Kopfknotens kennt.)

Warum schlägt man sich mit der Rückgabe der Adresse herum, wenn man sie überhaupt nicht verwendet? Die Methode `Insert()` ist in der Basisklasse – `Node` – deklariert. Den Rückgabewert benötigen die anderen Implementierungen. Wenn man den Rückgabewert von `HeadNode::Insert()` ändert, erhält man einen Compiler-Fehler. Es ist somit einfacher, den `HeadNode` zurückzugeben und die Adresse von der verketteten Liste verwerfen zu lassen.

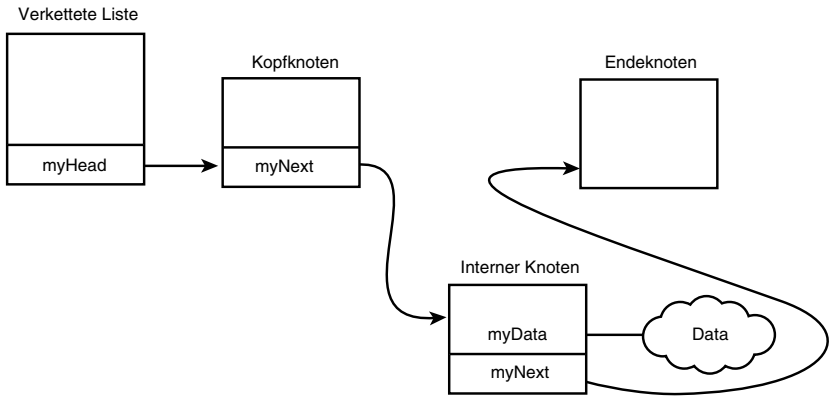


Abb. 19.3: Die verkettete Liste nach Einfügen des ersten Knotens

Was ist bisher geschehen? Die Daten wurden in die Liste eingefügt. Die Liste hat sie an den Kopf übergeben. Der Kopf hat diese Daten blindlings dorthin weitergereicht, wohin der Kopf gerade verweist. In diesem (ersten) Fall hat der Kopf auf das Ende gezeigt. Der Endeknoten hat sofort einen neuen internen Knoten erzeugt und den neuen Knoten mit einem Verweis auf das Ende initialisiert. Dann hat der Endeknoten die Adresse des neuen Knotens an den Kopf zurückgegeben und der Kopf hat diesen Zeiger auf den neuen Knoten in seinen Zeiger `myNext` eingetragen. Die Daten in der Liste befinden sich nun am richtigen Platz, wie es Abbildung 19.3 zeigt.

Nach dem Einfügen des ersten Knotens setzt das Programm mit Zeile 210 fort, nimmt einen Wert entgegen und testet ihn. Wir nehmen nun an, dass der Benutzer den Wert `3` eingegeben hat. Daraufhin erzeugt Zeile 214 ein neues `Data`-Objekt, das Zeile 215 in die Liste einfügt.

Auch hier übergibt die Liste in Zeile 196 die Daten an ihren Kopfknoten (`HeadNode`). Die Methode `HeadNode::Insert()` übergibt ihrerseits den neuen Wert an den Knoten, auf den `myNext` momentan zeigt. Wie wir wissen, verweist dieser Zeiger jetzt auf den Knoten mit dem `Data`-Objekt, dessen Wert gleich `15` ist. Es schließt sich der Aufruf der Methode `InternalNode::Insert` in Zeile 97 an.

In Zeile 101 veranlasst `InternalNode` über den eigenen Zeiger `myData`, dass das gespeicherte `Data`-Objekt (mit dem Wert `15`) seine Methode `Compare()` aufruft und dabei das neue `Data`-Objekt (mit dem Wert `3`) übergibt. Das führt zum Aufruf der in Zeile 41 dargestellten Methode `Compare()`.

Die Methode `Compare()` vergleicht beide Werte. Da `myValue` gleich `15` und der Wert `theOtherData.myValue` gleich `3` ist, gibt die Methode den Wert `kIsLarger` zurück. Daraufhin verzweigt das Programm zu Zeile 108.

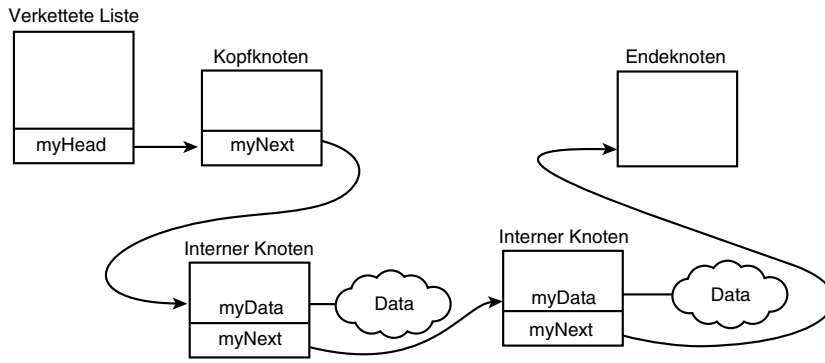


Abb. 19.4: Die verkettete Liste nach Einfügen des zweiten Knotens

Für das neue `Data`-Objekt wird ein neuer `InternalNode` erzeugt. Der neue Knoten zeigt auf das aktuelle `InternalNode`-Objekt und die Methode `InternalNode::Insert()` gibt die Adresse des neuen `InternalNode` an `HeadNode` zurück. Dementsprechend wird der neue Knoten, dessen Objektwert kleiner als der Objektwert des aktuellen Knotens ist, in die Liste eingefügt. Die Liste entspricht nun Abbildung 19.4.

Für den dritten Schleifendurchlauf nehmen wir an, dass der Benutzer den Wert `8` eingegeben hat. Dieser Wert ist größer als `3`, aber kleiner als `15` und sollte demnach zwischen die beiden vorhandenen Knoten eingefügt werden. Der Ablauf entspricht zunächst genau dem vorherigen Beispiel. Der Vergleich liefert jetzt aber `kIsSmaller` statt `kIsLarger` (weil das Objekt mit dem Wert `3` kleiner als das neue Objekt mit dem Wert `8` ist).

Daraufhin verzweigt die Methode `InternalNode::Insert()` zu Zeile 117. Anstatt einen neuen Knoten zu erzeugen und einzufügen, übergibt `InternalNode` einfach die neuen Daten an die Methode `Insert()` desjenigen Objekts, auf das der Zeiger `myNext` gerade zeigt. In diesem Fall erfolgt der Aufruf von `InsertNode` auf dem `InternalNode`, dessen `Data`-Objektwert gleich `15` ist.

Nach einem erneuten Vergleich wird ein neuer `InternalNode` erzeugt. Dieser zeigt auf den `InternalNode`, dessen `Data`-Objektwert gleich `15` ist. Die Adresse des eingefügten internen Knotens reicht das Programm an den `InternalNode` weiter, dessen `Data`-Objektwert `3` ist (Zeile 119).

Unterm Strich wird der neue Knoten an der richtigen Stelle in die Liste eingefügt.

Am Ende haben Sie eine sortierte Liste von Datenelementen – unabhängig davon, in welcher Reihenfolge Sie die Werte eingegeben haben.

19.4 Was haben wir gelernt?

Das hier vorgestellte Konzept unterscheidet sich grundlegend von der prozeduralen Programmierung. Bei einem herkömmlichen prozeduralen Programm würde eine steuernde Funktion die Daten untersuchen und die betreffenden Funktionen aufrufen.

In diesem objektorientierten Ansatz erhält jedes Objekt einen genau festgelegten Satz von Verantwortlichkeiten. Die verkettete Liste ist für die Verwaltung des Kopfknotens zuständig. Der Kopfknoten übergibt sofort die neuen Daten dorthin, wohin er gerade zeigt, ohne sich darum zu kümmern, wozu die Daten gut sind.

Sobald der Endeknoten Daten erhält, erzeugt er einen neuen Knoten und fügt ihn ein. Der Endeknoten handelt nach der Devise: »Wenn diese Daten zu mir gelangen, muss ich sie direkt vor mir einfügen.«

Interne Knoten sind kaum komplizierter. Sie fordern das gespeicherte Objekt auf, sich selbst mit dem neuen Objekt zu vergleichen. Je nach Ergebnis fügen sie entweder das neue Objekt ein oder geben es einfach weiter.

Beachten Sie, dass der interne Knoten *keine Ahnung* hat, wie der Vergleich auszuführen ist. Die Realisierung des Vergleichs bleibt dem Objekt selbst überlassen. Der interne Knoten fordert die Objekte lediglich zum Vergleich auf und erwartet eine von drei möglichen Antworten. Bei einer bestimmten Antwort wird das Objekt eingefügt; andernfalls reicht es der Knoten weiter und weiß nicht, wohin es schließlich gelangt.

Wer ist also verantwortlich? In einem gut konzipierten objektorientierten Programm ist *niemand* verantwortlich. Jedes Objekt werkelt still vor sich hin und im Endeffekt hat man eine gut funktionierende Maschine vor sich.

Das Schöne an einer verketteten Liste ist, dass man jeden beliebigen Datentyp in das Datenobjekt dieser Klasse einfügen kann. Im vorliegenden Beispiel enthält es eine Ganzzahl. Es sind aber auch mehrere vordefinierte Datentypen oder selbst andere Objekte (einschließlich anderer verketteter Listen) möglich.

Wenn man mit dynamischer Speicherzuordnung arbeitet, bleibt der Speicherbedarf bei kleinen verketteten Listen gering und nimmt bei großen Listen entsprechend zu. Vor allem aber verwendet eine verkettete Liste immer nur so viel Speicher, das für die momentan gespeicherten Daten Platz ist. Arrays dagegen reservieren von vornherein eine festgelegte Speichermenge, was verschwenderisch und einschränkend zugleich ist.

Warum verwendet man nicht immer verkettete Listen?

Angesichts der Leistungsfähigkeit von verketteten Listen taucht natürlich die Frage auf, warum man sie nicht immer verwendet. Die Leistungsfähigkeit hat ihren Preis. Verkettete Listen arbeiten sequenziell, d.h. man muss an dem einen Ende der Liste beginnen und die Liste bis zum anderen Ende durcharbeiten, sofern man das gewünschte Element nicht schon vorher gefunden hat. Dieses Zugriffsprinzip ist nicht gerade das schnellste.

Stellen Sie sich vor, Sie müssten das gesamte Telefonbuch durchlesen, um nach einer einzigen Nummer zu suchen! In der Regel geht man nach einer Art Teilungsmethode vor, wobei man in der Mitte beginnt oder an einer Stelle, wo vermutlich der gesuchte Buchstabe steht, den restlichen Teil des Buches in der entsprechenden Richtung teilt, falls der Buchstabe nicht vorhanden war, und das Ganze so lange wiederholt, bis man die Nummer gefunden hat. In der Informatik bezeichnet man das als binäre Suche.

19.5 Fragen und Antworten

F: Warum soll ich eine verkettete Liste erzeugen, wenn auch ein Array funktioniert?

A: Ein Array muss eine feste Größe haben, während sich eine verkettete Liste dynamisch zur Laufzeit in der Größe ändern kann.

F: Warum trennt man das Datenobjekt vom Knoten?

A: Nachdem die Knotenobjekte einmal in der richtigen Weise funktionieren, kann man diesen Code für alle möglichen Objekte wiederverwenden, die man in einer Liste speichern will.

F: Wenn ich andere Objekte in die Liste aufnehmen möchte, muss ich dann einen neuen Listentyp und einen neuen Knotentyp erzeugen?

A: Momentan ja. Dieses Problem lösen wir in Stunde 23 bei der Behandlung von Templates (Vorlagen).

19.6 Workshop

In dieser Stunde haben Sie sich mit einer komplexen Datenstruktur beschäftigt: der verketteten Liste. Nun können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu diesem Thema zu festigen.

19.6.1 Kontrollfragen

1. Was sind die drei Typen von Listen?
2. Welchen Zweck erfüllt der Kopfzeiger?
3. Welchen Zweck erfüllt der Zeiger auf den nächsten Knoten?
4. Was ist für eine doppelt verkettete Liste erforderlich?

19.6.2 Übungen

1. Führen Sie das Programm in Listing 19.1 (`linkedlist.cpp`) im Debug-Modus aus und fügen Sie eine Reihe von Knoten im Schrittbetrieb ein. Dabei lässt sich beobachten, wie diese Methoden einander aufrufen und die Zeiger geeignet angepasst werden.
2. Modifizieren Sie das Programm in Listing 19.1 (`linkedlist.cpp`), sodass die `Data`-Klasse den Datentyp `double` (anstelle von `int`) speichert. Was müssen Sie alles ändern?
3. Modifizieren Sie das Programm in Listing 19.1 (`linkedlist.cpp`) weiter, sodass die Klasse `Data` aus mehreren vordefinierten Datentypen (statt nur eines einzelnen `int` oder `double`) besteht. Was müssen Sie ändern? Hinweis: Es ist eine Art Schlüssel erforderlich, um den Vergleich ausführen zu können.

19.6.3 Antworten zu den Kontrollfragen

1. Einfach verkettete Listen, doppelt verkettete Listen und Bäume.
2. Wie aus dem Namen hervorgeht, enthält er den Kopf (oder den Beginn) der verketteten Liste. Man muss unbedingt wissen, wo die Liste beginnt, damit man darauf zugreifen kann.
3. Der Zeiger auf den nächsten Knoten verweist auf das nächste Element der verketteten Liste. Man kann sich diesen Knoten auch als Kopf für den Rest der Liste vorstellen. Der Zeiger stellt die eigentliche Verkettung in der verketteten Liste her.
4. Eine doppelt verkettete Liste hat sowohl Kopf- als auch Endezeiger und sowohl Zeiger auf den nächsten als auch auf den vorhergehenden Knoten, sodass man an einem beliebigen Ende der Liste beginnen und sich zum entgegengesetzten Ende der Liste durcharbeiten kann.

KAPITEL 20

jetzt lerne ich

Spezielle Klassen, Funktionen und Zeiger

In dieser Stunde lernen Sie,

- was statische Elementvariablen und statische Elementfunktionen sind,
- wie man statische Elementvariablen und statische Elementfunktionen verwendet,
- was Friend-Funktionen und Friend-Klassen sind,
- wie man mit Friend-Funktionen spezielle Probleme löst,
- wie man Zeiger auf Elementfunktionen verwendet.

20.1 Statische Datenelemente

Bis jetzt haben Sie sich wahrscheinlich die Daten in jedem Objekt als einzigartig zu diesem Objekt und nicht als gemeinsam genutzte Daten zwischen mehreren Objekten einer Klasse vorgestellt. Wenn man zum Beispiel fünf `Cat`-Objekte erzeugt hat, verfügt jedes Objekt über eigene Variablen für Alter, Gewicht und andere Daten. Das Alter des einen Objekts beeinflusst nicht das Alter eines anderen Objekts.

Manchmal möchte man aber Daten verfolgen, die mehrere Objekte einer Klasse gemeinsam nutzen – beispielsweise, wie viele Katzen bisher geboren wurden und wie viele es noch gibt.

Im Gegensatz zu normalen Elementvariablen werden *statische* Elementvariablen von allen Instanzen einer Klasse gemeinsam genutzt. Sie bilden einen

Kompromiss zwischen globalen Daten, die allen Teilen eines Programms zur Verfügung stehen, und Datenelementen, die normalerweise nur dem einzelnen Objekt verfügbar sind.

Man kann sich das so vorstellen, dass ein statisches Element zur Klasse und nicht zum Objekt gehört. Normale Datenelemente sind je einmal pro Objekt vorhanden, während statische Elemente nur einmal pro Klasse auftreten. Listing 20.1 deklariert ein `Cat`-Objekt mit dem statischen Datenelement `HowManyCats`. Diese Variable verzeichnet die Anzahl der erzeugten `Cat`-Objekte. Dazu inkrementiert das Programm die statische Variable `HowManyCats` bei jeder Konstruktion und dekrementiert sie bei jedem Abbau.

Listing 20.1: static-member.cpp – Statische Datenelemente

```
// Listing 20.1 Statische Datenelemente
#include <iostream>

class Cat
{
public:
    Cat(int age = 1):itsAge(age){HowManyCats++; }
    virtual ~Cat() { HowManyCats--; }
    virtual int GetAge() { return itsAge; }
    virtual void SetAge(int age) { itsAge = age; }
    static int HowManyCats;

private:
    int itsAge;

};

int Cat::HowManyCats = 0;

int main()
{
    const int MaxCats = 5;
    Cat *CatHouse[MaxCats];
    int i;
    for (i = 0; i<MaxCats; i++)
        CatHouse[i] = new Cat(i);

    for (i = 0; i<MaxCats; i++)
    {
        std::cout << "Es bleiben ";
        std::cout << Cat::HowManyCats;
        std::cout << " Katzen uebrig!\n";
        std::cout << "Diejenige loeschen, die ";
        std::cout << CatHouse[i]->GetAge();
        std::cout << " Jahre alt ist.\n";
        delete CatHouse[i];
```

```
37:         CatHouse[i] = 0;
38:     }
39:     return 0;
40: }
```

```
Es bleiben 5 Katzen uebrig!
Diejenige loeschen, die 0 Jahre alt ist.
Es bleiben 4 Katzen uebrig!
Diejenige loeschen, die 1 Jahre alt ist.
Es bleiben 3 Katzen uebrig!
Diejenige loeschen, die 2 Jahre alt ist.
Es bleiben 2 Katzen uebrig!
Diejenige loeschen, die 3 Jahre alt ist.
Es bleiben 1 Katzen uebrig!
Diejenige loeschen, die 4 Jahre alt ist.
```

Die Zeilen 4 bis 16 deklarieren die vereinfachte Klasse `Cat`. In Zeile 11 steht die Deklaration von `HowManyCats` als statische Elementvariable vom Typ `int`.

Die Deklaration von `HowManyCats` definiert keine Integer-Zahl. Es wird damit auch kein Speicher reserviert. Während die Instanzierung eines `Cat`-Objekts für nicht statische Elemente Speicher reserviert, ist das bei statischen Elementen nicht der Fall, da sich die Elementvariable (im Beispiel `HowManyCats`) *nicht im* Objekt befindet. Aus diesem Grund definiert und initialisiert die Anweisung in Zeile 18 diese Variable.

Es ist ein verbreiteter Fehler, die Definition der statischen Elementvariablen von Klassen zu vergessen. Achten Sie darauf, dass Ihnen das nicht passiert! Natürlich kommt es trotzdem irgendwann vor, und der Linker reagiert daraufhin mit einer Fehlermeldung wie hier beim C++BuilderX:

```
Error: Ungelöste externe 'Cat::HowManyCats' referenziert von
E:\...\STATICMEMBER.OBJ
```

Für die Variable `itsAge` ist keine derartige Definition erforderlich, da es sich um eine nicht statische Elementvariable handelt. Diese wird jedes Mal definiert, wenn man ein `Cat`-Objekt erzeugt, wie es hier in Zeile 26 geschieht.

Der Konstruktor für `Cat` inkrementiert die statische Elementvariable in Zeile 7. Der Destruktor dekrementiert sie in Zeile 8. Damit verzeichnet `HowManyCats` zu jedem Zeitpunkt die Anzahl der erzeugten und noch nicht abgebauten `Cat`-Objekte.

Das Rahmenprogramm in den Zeilen 20 bis 40 instanziert fünf Katzen und legt sie in einem Array ab. Die Instanzierung ist mit fünf Aufrufen von `Cat`-Konstruktoren verbunden. Demzufolge wird `HowManyCats` fünfmal vom Anfangswert `0` aus inkrementiert.

Das Programm durchläuft dann alle fünf Positionen im Array und gibt den Wert von `HowManyCats` aus, bevor der aktuelle `Cat`-Zeiger gelöscht wird. Die Ausgabe zeigt, dass der Anfangswert gleich 5 ist (schließlich wurden fünf Objekte konstruiert) und dass nach jedem Schleifendurchlauf ein `Cat`-Objekt weniger übrig bleibt.

Beachten Sie, dass `HowManyCats` öffentlich ist und `main()` direkt darauf zugreift. Es gibt keinen Grund, diese Elementvariable in dieser Form freizulegen. Besser ist es, sie mit den anderen Elementvariablen als privat zu deklarieren und eine öffentliche Zugriffsmethode bereitzustellen, solange man auf die Daten immer über eine Instanz von `Cat` zugreift. Will man andererseits auf diese Daten direkt zugreifen, ohne dass unbedingt ein `Cat`-Objekt verfügbar sein muss, gibt es zwei Möglichkeiten: die Variable öffentlich halten oder eine statische Elementfunktion bereitstellen.

20.2 Statische Elementfunktionen

Statische Elementfunktionen sind vergleichbar mit statischen Elementvariablen: Sie existieren nicht in einem Objekt, aber im Gültigkeitsbereich der Klasse. Demzufolge kann man sie aufrufen, ohne ein Objekt dieser Klasse verfügbar zu haben. Listing 20.2 zeigt dazu ein Beispiel.

Listing 20.2: static-function.cpp – Statische Element-funktionen

```
// Listing 20.2 Statische Elementfunktionen
#include <iostream>

class Cat
{
public:
    Cat(int age = 1):itsAge(age){HowManyCats++; }
    virtual ~Cat() { HowManyCats--; }
    virtual int GetAge() { return itsAge; }
    virtual void SetAge(int age) { itsAge = age; }
    static int GetHowMany() { return HowManyCats; }
private:
    int itsAge;
    static int HowManyCats;
};

int Cat::HowManyCats = 0;

void TelepathicFunction();

int main()
{
    const int MaxCats = 5;
```

```
24:     Cat *CatHouse[MaxCats];
25:     int i;
26:     for (i = 0; i<MaxCats; i++)
27:     {
28:         CatHouse[i] = new Cat(i);
29:         TelepathicFunction();
30:     }
31:
32:     for ( i = 0; i<MaxCats; i++)
33:     {
34:         delete CatHouse[i];
35:         TelepathicFunction();
36:     }
37:     return 0;
38: }
39:
40: void TelepathicFunction()
41: {
42:     std::cout << "Es bleiben " << Cat::GetHowMany()
43:         << " Katzen am Leben!\n";
44: }
```

```
Es bleiben 1 Katzen am Leben.
Es bleiben 2 Katzen am Leben.
Es bleiben 3 Katzen am Leben.
Es bleiben 4 Katzen am Leben.
Es bleiben 5 Katzen am Leben.
Es bleiben 4 Katzen am Leben.
Es bleiben 3 Katzen am Leben.
Es bleiben 2 Katzen am Leben.
Es bleiben 1 Katzen am Leben.
Es bleiben 0 Katzen am Leben.
```

Zeile 14 der `Cat`-Deklaration deklariert die statische Elementvariable `HowManyCats` für den privaten Zugriff. Zeile 11 deklariert die öffentliche Zugriffsfunktion `GetHowMany()` sowohl öffentlich als auch statisch.

Da `GetHowMany()` öffentlich ist, kann man darauf mit jeder Funktion zugreifen. Durch ihren statischen Charakter benötigt man kein Objekt vom Typ `Cat`, auf dem man diese Funktion aufruft. Demzufolge kann die Funktion `TelepathicFunction()` in Zeile 42 die öffentliche Zugriffsfunktion aufrufen, obwohl sie selbst keinen Zugriff auf ein `Cat`-Objekt hat. Natürlich könnte man `GetHowMany()` auch auf den in `main()` verfügbaren `Cat`-Objekten aufrufen, genau wie bei allen anderen Zugriffsfunktionen.

Statische Elementfunktionen haben keinen `this`-Zeiger. Demzufolge kann man sie nicht als `const` deklarieren. Da außerdem der Zugriff auf Datenelemente in Elementfunktionen über den Zeiger `this` erfolgt, bleibt den statischen Elementfunktionen der Zugriff auf nicht statische Elementvariablen verwehrt!

20.3 Containment von Klassen

Wie Sie in den vorherigen Beispielen gesehen haben, kann ein Datenelement einer Klasse Objekte einer anderen Klasse einbinden. C++-Programmierer sprechen davon, dass die äußere Klasse die innere Klasse enthält (englisch: contain – enthalten). Demzufolge kann beispielsweise eine `Employee`-Klasse sowohl `String`-Objekte (für den Namen des Mitarbeiters) als auch Integer-Werte (für das Gehalt des Mitarbeiters) enthalten.

Listing 20.3 zeigt eine abgespeckte aber nützliche `String`-Klasse.

Listing 20.3: string.hpp – Die Klasse String

```
// Listing 20.3. string.hpp
#include <iostream>
#include <string.h>
class String
{
public:
    // Konstruktoren
    String();
    String(const char *const);
    String(const String &);
    ~String();

    // Überladene Operatoren
    char & operator[](int offset);
    char operator[](int offset) const;
    String operator+(const String&);
    void operator+=(const String&);
    String & operator= (const String &);

    // Allgemeine Zugriffsfunktionen
    int GetLen()const { return itsLen; }
    const char * GetString() const { return itsString; }
    // static int ConstructorCount;

private:
    String(int);         // Privater Konstruktor
    char * itsString;
    int itsLen;
};
```

```
30:
31: // Standardkonstruktor erzeugt String von 0 Bytes Länge
32: String::String()
33: {
34:     itsString = new char[1];
35:     itsString[0] = '\0';
36:     itsLen=0;
37:     // std::cout << "\tString-Standardkonstruktor\n";
38:     // ConstructorCount++;
39: }
40:
41: // Privater (Hilfs-) Konstruktor, nur von Klassen-
42: // methoden zum Erzeugen eines neuen Strings der
43: // erforderlichen Größe verwendet. Mit Null gefüllt.
44: String::String(int len)
45: {
46:     itsString = new char[len+1];
47:     int i;
48:     for (i = 0; i<=len; i++)
49:         itsString[i] = '\0';
50:     itsLen=len;
51:     // std::cout << "\tString(int)-Konstruktor\n";
52:     // ConstructorCount++;
53: }
54:
55: String::String(const char * const cString)
56: {
57:     itsLen = strlen(cString);
58:     itsString = new char[itsLen+1];
59:     int i;
60:     for (i = 0; i<itsLen; i++)
61:        itsString[i] = cString[i];
62:     itsString[itsLen]='\0';
63:     // std::cout << "\tString(char*)-Konstruktor\n";
64:     // ConstructorCount++;
65: }
66:
67: String::String (const String & rhs)
68: {
69:     itsLen=rhs.GetLen();
70:     itsString = new char[itsLen+1];
71:     int i;
72:     for ( i = 0; i<itsLen;i++)
73:         itsString[i] = rhs[i];
74:     itsString[itsLen] = '\0';
75:     // std::cout << "\tString(String&)-Konstruktor\n";
76:     // ConstructorCount++;
77: }
```

```
 78:
 79: String::~String ()
 80: {
 81:     delete [] itsString;
 82:     itsLen = 0;
 83:     // std::cout << "\tString-Destruktor\n";
 84: }
 85:
 86: // Gleich-Operator, gibt vorhandenen Speicher frei,
 87: // kopiert dann String und Größe
 88: String& String::operator=(const String & rhs)
 89: {
 90:     if (this == &rhs)
 91:         return *this;
 92:     delete [] itsString;
 93:     itsLen=rhs.GetLen();
 94:     itsString = new char[itsLen+1];
 95:     int i;
 96:     for (i = 0; i<itsLen;i++)
 97:         itsString[i] = rhs[i];
 98:     itsString[itsLen] = '\0';
 99:     return *this;
100:     // std::cout << "\tString-Operator=\n";
101: }
102:
103: // Nicht-konstanter Offset-Operator, gibt
104: // Referenz auf Zeichen zurück, das sich damit
105: // ändern lässt!
106: char & String::operator[](int offset)
107: {
108:     if (offset > itsLen)
109:         return itsString[itsLen-1];
110:     else
111:         return itsString[offset];
112: }
113:
114: // Konstanter Offset-Operator für konstante
115: // Objekte (siehe Kopierkonstruktor!)
116: char String::operator[](int offset) const
117: {
118:     if (offset > itsLen)
119:         return itsString[itsLen-1];
120:     else
121:         return itsString[offset];
122: }
123:
```

```
124: // Erzeugt einen neuen String durch Anfügen des
125: // aktuellen Strings an rhs
126: String String::operator+(const String& rhs)
127: {
128:     int  totalLen = itsLen + rhs.GetLen();
129:     int i,j;
130:     String temp(totalLen);
131:     for (i = 0; i<itsLen; i++)
132:         temp[i] = itsString[i];
133:     for (j = 0; j<rhs.GetLen(); j++, i++)
134:         temp[i] = rhs[j];
135:     temp[totalLen]='\0';
136:     return temp;
137: }
138:
139: // Ändert aktuellen String, gibt nichts zurück
140: void String::operator+=(const String& rhs)
141: {
142:     int rhsLen = rhs.GetLen();
143:     int totalLen = itsLen + rhsLen;
144:     int i,j;
145:     String  temp(totalLen);
146:     for (i = 0; i<itsLen; i++)
147:         temp[i] = itsString[i];
148:     for (j = 0; j<rhs.GetLen(); j++, i++)
149:         temp[i] = rhs[i-itsLen];
150:     temp[totalLen]='\0';
151:     *this = temp;
152: }
153:
154: // int String::ConstructorCount = 0;
```

Dieses Programm hat keine Ausgabe.

Das Programm deklariert die statische Elementvariable `ConstructorCount` in Zeile 23 und initialisiert sie in Zeile 154. In jedem `String`-Konstruktor wird diese Variable inkrementiert.

Listing 20.4 beschreibt eine `Employee`-Klasse, die drei `String`-Objekte enthält. Beachten Sie, dass eine Reihe von Anweisungen auskommentiert ist. Diese werden in den Übungen verwendet.

Wiederverwendbarkeit

Der Code von Listing 20.3 steht in der Datei `string.hpp`. Wenn Sie die Klasse `String` benötigen, können Sie Listing 20.3 mithilfe der `#include`-Anweisung einbinden. Zum Beispiel erscheint am Anfang von Listing 20.4 die Anweisung `#include "string.hpp"`. Damit nehmen Sie die `String`-Klasse in Ihr Programm auf.

Listing 20.4: employee-main.cpp – Die Klasse Employee und das Rahmenprogramm

```
#include "string.hpp"
// iostream bereits eingebunden
using std::cout; // Diese Datei verwendet std::cout


class Employee
{
public:
    Employee();
    Employee(char *, char *, char *, long);
    ~Employee();
    Employee(const Employee&);
    Employee & operator= (const Employee &);

    const String & GetFirstName() const { return itsFirstName; }
    const String & GetLastName() const { return itsLastName; }
    const String & GetAddress() const { return itsAddress; }
    long GetSalary() const { return itsSalary; }

    void SetFirstName(const String & fName)
        { itsFirstName = fName; }
    void SetLastName(const String & lName)
        { itsLastName = lName; }
    void SetAddress(const String & address)
        { itsAddress = address; }
    void SetSalary(long salary) { itsSalary = salary; }
private:
    String    itsFirstName;
    String    itsLastName;
    String    itsAddress;
    long      itsSalary;
};

Employee::Employee():
    itsFirstName(""),
    itsLastName(""),
    itsAddress(""),
    itsSalary(0)
{}
```

```
40:
41: Employee::Employee(char * firstName, char * lastName,
42:     char * address, long salary):
43:     itsFirstName(firstName),
44:     itsLastName(lastName),
45:     itsAddress(address),
46:     itsSalary(salary)
47: {}
48:
49: Employee::Employee(const Employee & rhs):
50:     itsFirstName(rhs.GetFirstName()),
51:     itsLastName(rhs.GetLastName()),
52:     itsAddress(rhs.GetAddress()),
53:     itsSalary(rhs.GetSalary())
54: {}
55:
56: Employee::~Employee() {}
57:
58: Employee & Employee::operator= (const Employee & rhs)
59: {
60:     if (this == &rhs)
61:         return *this;
62:
63:     itsFirstName = rhs.GetFirstName();
64:     itsLastName = rhs.GetLastName();
65:     itsAddress = rhs.GetAddress();
66:     itsSalary = rhs.GetSalary();
67:
68:     return *this;
69: }
70:
71: int main()
72: {
73:     Employee Edie("Jane","Doe","1461 Shore Parkway", 20000);
74:     Edie.SetSalary(50000);
75:     String LastName("Levine");
76:     Edie.SetLastName(LastName);
77:     Edie.SetFirstName("Edythe");
78:
79:     cout << "Name: ";
80:     cout << Edie.GetFirstName().GetString();
81:     cout << " " << Edie.GetLastName().GetString();
82:     cout << ".\nAdresse: ";
83:     cout << Edie.GetAddress().GetString();
84:     cout << ".\nGehalt: " ;
85:     cout << Edie.GetSalary();
86:     return 0;
87: }
```

```
Name: Edythe Levine.
Adresse: 1461 Shore Parkway.
Gehalt: 50000
```

Listing 20.4 zeigt die Klasse `Employee` (Angestellte) die drei `String`-Objekte enthält: `itsFirstName` (Vorname), `itsLastName` (Nachname) und `itsAddress` (Adresse).

Die Anweisung in Zeile 73 erzeugt ein `Employee`-Objekt und übergibt ihm vier Werte zur Initialisierung. Zeile 74 ruft die `Employee`-Zugriffsfunktion `SetSalary()` (Gehalt festlegen) mit dem konstanten Wert `50000` auf. Beachten Sie, dass man an dieser Stelle in einem realen Programm entweder einen dynamischen Wert (zur Laufzeit berechnet) oder eine Konstante übergibt.

Zeile 75 erzeugt einen String und initialisiert ihn mit einer C++-String-Konstanten. Dieses String-Objekt wird dann als Argument an `SetLastName()` (Nachname festlegen) in Zeile 76 verwendet.

Zeile 77 ruft die `Employee`-Funktion `SetFirstName()` (Vorname festlegen) mit einer anderen String-Konstanten auf. Sicherlich haben Sie bemerkt, dass `Employee` gar nicht über eine Funktion `SetFirstName()` verfügt, die eine Zeichenfolge als Argument übernimmt – `SetFirstName()` erfordert eine konstante String-Referenz.

Der Compiler kann diese Umwandlung einer konstanten Zeichenfolge in einen String vornehmen, da die Anleitung dazu in Zeile 9 von Listing 20.3 steht.

20.3.1 Auf Elemente der enthaltenen Klasse zugreifen

`Employee`-Objekte haben keinen speziellen Zugriff auf die Elementvariablen von `String`. Wenn das `Employee`-Objekt `Edie` versucht, auf die Elementvariable `itsLen` seiner eigenen Elementvariablen `itsFirstName` zuzugreifen, erhält man einen Compiler-Fehler. Allerdings ist das kein großes Problem. Die Zugriffsfunktionen bieten eine Schnittstelle für die `String`-Klasse und die `Employee`-Klasse braucht sich um die Details der Implementierung genauso wenig zu kümmern, wie um die Art der Speicherung von Informationen in der Integer-Variablen `itsSalary`.

20.3.2 Zugriff auf enthaltene Elemente filtern

Die `String`-Klasse stellt den Operator `operator+` bereit. Der Designer der `Employee`-Klasse hat den Zugriff auf den Operator `operator+` gegenüber Aufrufen auf `Employee`-Objekten blockiert, indem er alle `String`-Zugriffsfunktio-

nen wie zum Beispiel `GetFirstName()` für die Rückgabe einer konstanten Referenz deklariert hat. Da `operator+` keine konstante Funktion ist (und auch nicht sein kann, da er das Objekt verändert, auf dem er aufgerufen wird), führen Versuche wie die folgende Anweisung zu einem Compiler-Fehler:

```
String buffer = Edie.GetFirstName() + Edie.GetLastName();
```

`GetFirstName()` gibt ein konstantes `String`-Objekt zurück und `operator+` kann man nicht auf einem konstanten Objekt aufrufen.

Um dieses Problem zu lösen, überladen Sie `GetFirstName()` als nicht konstant:

```
const String & GetFirstName() const { return itsFirstName; }
String & GetFirstName()  { return itsFirstName; }
```

Beachten Sie, dass jetzt weder der Rückgabewert noch die Elementfunktion selbst konstant sind. Die Änderung des Rückgabewertes reicht nicht aus, um den Funktionsnamen zu überladen. Man muss das Konstante der Funktion selbst ändern. Indem man sowohl eine konstante als auch eine nicht konstante Version bereitstellt, ruft der Compiler die konstante Version nach Möglichkeit auf (beispielsweise wenn ein Klient `GetFirstName()` aufruft) und die nicht konstante Version nach Bedarf (beispielsweise wenn man sie mit dem `operator+` aufruft).

20.3.3 Nachteile des Containments

Es sei darauf hingewiesen, dass der Benutzer einer `Employee`-Klasse jedes Mal den Preis für diese `String`-Objekte zahlt, wenn ein `String`-Konstruktor aufgerufen oder eine Kopie von `Employee` angelegt wird.

20.3.4 Als Wert und als Referenz kopieren

Übergibt man `Employee`-Objekte als Wert, werden ebenfalls alle enthaltenen Strings kopiert und demzufolge Kopierkonstruktoren aufgerufen. Das ist sehr aufwändig, verbraucht Speicher und kostet Zeit.

Wenn man `Employee`-Objekte als Referenz übergibt (mithilfe von Zeigern oder Referenzen), spart man das alles ein. Aus diesem Grund setzen C++-Programmierer alles daran, nichts als Wert zu übergeben, was eine Größe von einigen Bytes übersteigt.

20.4 Friend-Klassen

Manchmal erzeugt man Klassen, die in bestimmter Weise zu einer Gruppe gehören. Diese gepaarten Klassen benötigen gegebenenfalls untereinander Zugriff auf private Elemente, wobei diese Informationen aber nicht öffentlich sein sollen.

Wenn man private Datenelemente oder Funktionen für eine andere Klasse offen legt, muss man diese Klasse als *Friend* (Freund) deklarieren. Damit erweitert man die Schnittstelle einer Klasse, um die Friend-Klasse einzubinden.

Allerdings ist Freundschaft nicht übertragbar. Nur weil du mein Freund bist und Joe dein Freund ist, bedeutet das nicht, dass Joe mein Freund ist. Freundschaft wird darüber hinaus auch nicht vererbt. Auch hier eine Analogie: Nur weil du mein Freund bis und ich mit dir meine Geheimnisse teile, heißt das noch nicht, dass ich meine Geheimnisse auch deinen Kindern anvertrauen möchte.

Schließlich ist Freundschaft nicht kommutativ. Die Zuweisung von `ClassOne` als Freund von `ClassTwo` macht nicht `ClassTwo` zu einem Freund von `ClassOne`. Nur weil du mir deine Geheimnisse mitteilen willst, bedeutet das nicht, dass ich dir auch meine Geheimnisse erzählen will.

Deklarationen von Friend-Klassen sollte man mit äußerster Vorsicht verwenden. Wenn zwei Klassen durch und durch verflochten sind und man häufig auf Daten in der anderen zugreifen muss, gibt es gute Gründe, diese Deklaration zu verwenden. Trotzdem sollte man sparsam damit umgehen. Es ist oftmals genauso einfach, die öffentlichen Zugriffsmethoden zu nutzen. Das hat außerdem den Vorteil, dass man eine Klasse ändern kann, ohne die andere neu kompilieren zu müssen.

Kapselung

Oftmals beklagen sich Neueinsteiger in die C++-Programmierung, dass Friend-Deklarationen die für die objektorientierte Programmierung so bedeutsame Kapselung »untergraben«. Offen gesagt ist das blanker Unsinn. Die Friend-Deklaration macht den deklarierten Freund zum Teil der Klassenschnittstelle und ist genauso wenig ein Unterwandern der Kapselung wie eine öffentliche Ableitung.

20.5 Friend-Funktionen

Manchmal will man den freundschaftlichen Zugriff nicht auf eine gesamte Klasse, sondern nur auf eine oder zwei Funktionen dieser Klasse gewähren. Das lässt sich realisieren, indem man die Elementfunktionen der anderen Klasse und nicht die gesamte Klasse als Freund deklariert. In der Tat kann man jede Funktion als Friend-Funktion deklarieren, ob sie nun eine Elementfunktion einer anderen Klasse ist oder nicht.

20.6 Zeiger auf Funktionen

Genau wie ein Array-Name ein konstanter Zeiger auf das erste Element des Arrays ist, stellt ein Funktionsname einen konstanten Zeiger auf die Funktion dar. Man kann also eine Variable als Zeiger auf eine Funktion deklarieren und die Funktion mithilfe dieses Zeigers aufrufen. Damit lässt sich in einem Programm zur Laufzeit – etwa in Abhängigkeit von einer Benutzereingabe – entscheiden, welche Funktion aufzurufen ist.

Allerdings muss man bei Funktionszeigern den genauen Objekttyp kennen, auf den man zeigen möchte. Ein Zeiger auf `int` verweist auf eine Integer-Variable und ein Zeiger auf eine Funktion muss auf eine Funktion des passenden Rückgabetyps und der passenden Signatur zeigen.

In der Deklaration

```
long (* funcPtr) (int);
```

wird `funcPtr` als Zeiger deklariert (beachten Sie das * vor dem Namen). Er zeigt auf eine Funktion, die einen Integer-Parameter übernimmt und einen Wert vom Typ `long` zurückgibt. Die Klammern um `* funcPtr` sind erforderlich, da die Klammern um `int` enger binden – das heißt, einen höheren Vorrang als der Indirektionsoperator (*) haben. Ohne die ersten Klammern würde diese Anweisung eine Funktion deklarieren, die einen Integer übernimmt und einen Zeiger auf einen `long` zurückgibt. (Denken Sie daran, dass Leerzeichen hier bedeutungslos sind.)

Sehen Sie sich die folgenden Deklarationen an:

```
long * Function (int);
long (* funcPtr) (int);
```

Die erste Deklaration, `Function()`, ist eine Funktion, die einen Integer übernimmt und einen Zeiger auf eine Variable vom Typ `long` zurückgibt. Die zweite Deklaration, `funcPtr`, ist ein Zeiger auf eine Funktion, die einen Integer übernimmt und eine Variable vom Typ `long` zurückgibt.

Die Deklaration eines Funktionszeigers schließt immer den Rückgabetyp und die Klammern zur Kennzeichnung der Parametertypen (falls vorhanden) ein. Listing 20.5 zeigt die Deklaration und Verwendung von Funktionszeigern.

Listing 20.5: ptrtofunction.cpp – Zeiger auf Funktionen

```
// Listing 20.5 Zeiger auf Funktionen
#include <iostream>

void Square (int&,int&);
void Cube (int&, int&);
void Swap (int&, int &);
void GetVals(int&, int&);
void PrintVals(int, int);

int main()
{
    void (* pFunc) (int &, int &);
    bool fQuit = false;

    int valOne=1, valTwo=2;
    int choice;
    while (fQuit == false)
    {
        std::cout << "(0)Beenden (1)Werte aendern "
            << "(2)Quadrat (3)Dritte Potenz (4)Vertauschen: ";
        std::cin >> choice;
        switch (choice)
        {
        case 1:
            pFunc = GetVals;
            break;
        case 2:
            pFunc = Square;
            break;
        case 3:
            pFunc = Cube;
            break;
        case 4:
            pFunc = Swap;
            break;
        default :
            fQuit = true;
            break;
        }

        if (fQuit)
            break;

```

```
44:         PrintVals(valOne, valTwo);
45:         pFunc(valOne, valTwo);
46:         PrintVals(valOne, valTwo);
47:     }
48:     return 0;
49: }
50:
51: void PrintVals(int x, int y)
52: {
53:     std::cout << "x: " << x << " y: " << y << std::endl;
54: }
55:
56: void Square (int & rX, int & rY)
57: {
58:     rX *= rX;
59:     rY *= rY;
60: }
61:
62: void Cube (int & rX, int & rY)
63: {
64:     int tmp;
65:
66:     tmp = rX;
67:     rX *= rX;
68:     rX = rX * tmp;
69:
70:     tmp = rY;
71:     rY *= rY;
72:     rY = rY * tmp;
73: }
74:
75: void Swap(int & rX, int & rY)
76: {
77:     int temp;
78:     temp = rX;
79:     rX = rY;
80:     rY = temp;
81: }
82:
83: void GetVals (int & rValOne, int & rValTwo)
84: {
85:     std::cout << "Neuer Wert fuer ValOne: ";
86:     std::cin >> rValOne;
87:     std::cout << "Neuer Wert fuer ValTwo: ";
88:     std::cin >> rValTwo;
89: }
```

```
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 1
x: 1 y:2
Neuer Wert fuer ValOne: 2
Neuer Wert fuer ValTwo: 3
x: 2 y:3
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 3
x: 2 y:3
x: 8 y: 27
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 2
x: 8 y: 27
x:64 y:729
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 4
x:64 y:729
x:729 y:64
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 0
```

Die Zeilen 4 bis 7 deklarieren vier Funktionen mit gleichem Rückgabetyp und gleicher Signatur. Diese Funktionen geben `void` zurück und übernehmen zwei Referenzen auf Ganzzahlen.

In Zeile 12 wird `pFunc` als Zeiger auf eine Funktion deklariert, die `void` zurückgibt und zwei Referenzen auf `int` als Parameter übernimmt. Damit lässt sich mit `pFunc` auf die eingangs erwähnten vier Funktionen zeigen. Die Schleifenkonstruktion in den Zeilen 17 bis 47 erlaubt dem Benutzer wiederholt die Auswahl der aufzurufenden Funktion. Nach der Auswahl wird die entsprechende Funktion an `pFunc` zugewiesen. Die Zeilen 44 bis 46 geben die aktuellen Werte der beiden Ganzzahlen aus, rufen die momentan zugewiesene Funktion auf und geben dann die Werte erneut aus.

20.6.1 Kurzaufruf

Den Zeiger auf eine Funktion muss man nicht dereferenzieren, obwohl man es durchaus tun könnte. Wenn also `pFunc` ein Zeiger auf eine Funktion ist, die einen Integer übernimmt und eine Variable vom Typ `long` zurückgibt, und man `pFunc` eine passende Funktion zuweist, kann man diese Funktion entweder mit

```
pFunc(x);
```

oder mit

```
(*pFunc)(x);
```

aufrufen. Beide Formen sind identisch. Die erste ist nur eine Kurzversion der zweiten.

20.6.2 Arrays mit Zeigern auf Funktionen

Genau wie man ein Array mit Zeigern auf Integer deklarieren kann, lässt sich auch ein Array mit Zeigern auf Funktionen deklarieren, die einen bestimmten Werttyp zurückgeben und mit einer bestimmten Signatur versehen sind. (Siehe Listing 20.6.)

Listing 20.6: ptrarray-function.cpp – Ein Array von Zeigern auf Funktionen

```
 1: // Listing 20.6 Array mit Zeigern auf Funktionen
 2: #include <iostream>
 3:
 4: void Square (int&,int&);
 5: void Cube (int&, int&);
 6: void Swap (int&, int &);
 7: void GetVals(int&, int&);
 8: void PrintVals(int, int);
 9:
10: int main()
11: {
12:     int valOne=1, valTwo=2;
13:     int choice,i;
14:     const int MaxArray = 5;
15:     void (*pFuncArray[MaxArray])(int&, int&);
16:
17:     for (i=0;i<MaxArray;i++)
18:     {
19:         std::cout << "(1)Werte aendern "
20:             << "(2)Quadrat (3)Dritte Potenz (4)Vertauschen: ";
21:         std::cin >> choice;
22:         switch (choice)
23:         {
24:         case 1:
25:             pFuncArray[i] = GetVals;
26:             break;
27:         case 2:
28:             pFuncArray[i] = Square;
29:             break;
30:         case 3:
31:             pFuncArray[i] = Cube;
32:             break;
33:         case 4:
34:             pFuncArray[i] = Swap;
35:             break;
```

```
36:         default:
37:             pFuncArray[i] = 0;
38:         }
39:     }
40:
41:     for (i=0;i<MaxArray; i++)
42:     {
43:         pFuncArray[i](valOne,valTwo);
44:         PrintVals(valOne,valTwo);
45:     }
46:     return 0;
47: }
48:
49: void PrintVals(int x, int y)
50: {
51:     std::cout << "x: " << x << " y: " << y << std::endl;
52: }
53:
54: void Square (int & rX, int & rY)
55: {
56:     rX *= rX;
57:     rY *= rY;
58: }
59:
60: void Cube (int & rX, int & rY)
61: {
62:     int tmp;
63:
64:     tmp = rX;
65:     rX *= rX;
66:     rX = rX * tmp;
67:
68:     tmp = rY;
69:     rY *= rY;
70:     rY = rY * tmp;
71: }
72:
73: void Swap(int & rX, int & rY)
74: {
75:     int temp;
76:     temp = rX;
77:     rX = rY;
78:     rY = temp;
79: }
80:
```

```
81: void GetVals (int & rValOne, int & rValTwo)
82: {
83:     std::cout << "Neuer Wert fuer ValOne: ";
84:     std::cin >> rValOne;
85:     std::cout << "Neuer Wert fuer ValTwo: ";
86:     std::cin >> rValTwo;
87: }
```

Vorsicht mit diesem Beispiel!

Wie bereits zu Beginn dieses Buches erwähnt, sind die Beispiele nicht »narrensicher« im Sinne einer »defensiven Programmierung« ausgelegt. Wenn Sie im Beispiel von Listing 20.6 eine andere Option als 1, 2, 3 oder 4 wählen, erhalten Sie eine Fehlermeldung. Der konkrete Wortlaut hängt von Betriebssystem, Compiler und Debugger ab; die Ursache ist ein Aufruf einer nicht definierten Funktion (siehe Zeile 37).

Dieses Problem ließe sich zwar vermeiden, doch dann wäre das Programm wesentlich länger und nicht so anschaulich.

```
(1)Werte aendern (2)Quadrat (3)Dritte Potenz (4)Vertauschen: 1
(1)Werte aendern (2)Quadrat (3)Dritte Potenz (4)Vertauschen: 2
(1)Werte aendern (2)Quadrat (3)Dritte Potenz (4)Vertauschen: 3
(1)Werte aendern (2)Quadrat (3)Dritte Potenz (4)Vertauschen: 4
(1)Werte aendern (2)Quadrat (3)Dritte Potenz (4)Vertauschen: 2
Neuer Wert fuer ValOne: 2
Neuer Wert fuer ValTwo: 3
x: 2 y: 3
x: 4 y: 9
x: 64 y: 729
x: 729 y: 64
x: 531441 y:4096
```

In der Schleife in den Zeilen 17 bis 39 wählt der Benutzer die aufzurufende Funktion aus. Den Elementen des Arrays wird die Adresse der entsprechenden Funktion zugewiesen. Die Schleife in den Zeilen 41 bis 45 ruft jede Funktion der Reihe nach auf (Zeile 43 führt die Funktion aus, deren Adresse im Array gespeichert ist). Das Ergebnis erscheint nach jedem Aufruf auf dem Bildschirm.

20.6.3 Zeiger auf Funktionen an andere Funktionen übergeben

Die Zeiger auf Funktionen (und somit auch Arrays mit Zeigern auf Funktionen) kann man an andere Funktionen übergeben, die Aktionen ausführen und dann die richtige Funktion unter Verwendung des Zeigers aufrufen.

Beispielsweise kann man Listing 20.6 verbessern, indem man den gewählten Funktionszeiger an eine andere Funktion (außerhalb von `main()`) übergibt. Diese Funktion gibt dann die Werte aus, ruft die gewählte Funktion auf und zeigt die Werte erneut an. Listing 20.7 zeigt diese Variante.

Listing 20.7: passingptrfunction.cpp – Übergabe von Zeigern auf Funktionen als Funktionsargumente

```
// Listing 20.7 Pointers to Functions as args
#include <iostream>
using namespace std; // Diese Datei verwendet std:: objects

void Square (int&,int&);
void Cube (int&, int&);
void Swap (int&, int &);
void GetVals(int&, int&);
void PrintVals(void (*)(int&, int&),int&, int&);

int main()
{
    int valOne=1, valTwo=2;
    int choice;
    bool fQuit = false;

    void (*pFunc)(int&, int&);

    while (fQuit == false)
    {
        cout << "(0)Beenden (1)Werte aendern "
            << "(2)Quadrat (3)Dritte Potenz (4)Vertauschen: ";
        cin >> choice;
        switch (choice)
        {
        case 1:
            pFunc = GetVals;
            break;
        case 2:
            pFunc = Square;
            break;
        case 3:
            pFunc = Cube;
            break;
        case 4:
            pFunc = Swap;
            break;
```

```
38:         default:
39:             fQuit = true;
40:             break;
41:         }
42:         if (fQuit == true)
43:             break;
44:         PrintVals ( pFunc, valOne, valTwo);
45:     }
46:
47:   return 0;
48: }
49:
50: void PrintVals( void (*pFunc)(int&, int&),int& x, int& y)
51: {
52:     cout << "x: " << x << " y: " << y << endl;
53:     pFunc(x,y);
54:     cout << "x: " << x << " y: " << y << endl;
55: }
56:
57: void Square (int & rX, int & rY)
58: {
59:     rX *= rX;
60:     rY *= rY;
61: }
62:
63: void Cube (int & rX, int & rY)
64: {
65:     int tmp;
66:
67:     tmp = rX;
68:     rX *= rX;
69:     rX = rX * tmp;
70:
71:     tmp = rY;
72:     rY *= rY;
73:     rY = rY * tmp;
74: }
75:
76: void Swap(int & rX, int & rY)
77: {
78:     int temp;
79:     temp = rX;
80:     rX = rY;
81:     rY = temp;
82: }
83:
84: void GetVals (int & rValOne, int & rValTwo)
85: {
```

```
86:     cout << "Neuer Wert fuer ValOne: ";
87:     cin >> rValOne;
88:     cout << "Neuer Wert fuer ValTwo: ";
89:     cin >> rValTwo;
90:}
```

```
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 1
x: 1 y: 2
Neuer Wert fuer ValOne: 2
Neuer Wert fuer ValTwo: 3
x: 2 y: 3
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 3
x: 2 y: 3
x: 8 y: 27
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 2
x: 8 y: 27
x: 64 y: 729
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 4
x: 64 y: 729
x: 729 y: 64
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 0
```

Zeile 17 deklariert `pFunc` als Zeiger auf eine Funktion, die `void` zurückgibt und zwei Referenzen auf `int` als Parameter übernimmt. Zeile 9 deklariert `PrintVals()` als Funktion mit drei Parametern. Der erste ist ein Zeiger auf eine Funktion, die `void` zurückgibt und zwei Integer-Referenzen als Parameter übernimmt. Die beiden anderen Argumente an `PrintVals()` sind Integer-Referenzen. Der Benutzer wählt wieder aus, welche Funktion aufzurufen ist. In Zeile 44 wird dann `PrintVals()` aufgerufen.

Fragen Sie doch mal einen C++-Programmierer, was die folgende Deklaration bedeutet:

```
void PrintVals(void (*)(int&, int&),int&, int&);
```

Die Antwort: Es ist eine Deklaration einer Funktion, die `void` zurückgibt und als ihre Parameter eine Funktion mit zwei Integer-Referenzen sowie zwei andere Integer-Referenzen übernimmt. Diese Art von Deklaration werden Sie selten verwenden und wahrscheinlich jedes Mal im Buch nachsehen, wenn

Sie sie brauchen. Manchmal rettet aber gerade eine derartige Deklaration Ihren Programmentwurf. Mit `typedef` lässt sich das aber verständlicher formulieren.

20.6.4 typedef bei Zeigern auf Funktionen

Die Konstruktion `void (*)(int&, int&)` ist bestenfalls unbequem. Das Ganze lässt sich mit `typedef` vereinfachen. Man deklariert einen Typ `VPF` als Zeiger auf eine Funktion, die `void` zurückgibt und zwei Integer-Referenzen übernimmt. In Listing 20.8 ist der erste Teil von Listing 20.7 in der Version mit `typedef` neu geschrieben.

Listing 20.8: usingtypedef – Einsatz von typedef, um Zeiger auf Funktionen verständlicher zu machen

```
// Listing 20.8. Einsatz von typedef
#include <iostream>
using namespace std; // Diese Datei verwendet std::-Objekte

void Square (int&,int&);
void Cube (int&, int&);
void Swap (int&, int &);
void GetVals(int&, int&);
typedef  void (*VPF) (int&, int&) ;
void PrintVals(VPF,int&, int&);

int main()
{
    int valOne=1, valTwo=2;
    int choice;
    bool fQuit = false;

    VPF pFunc;

    while (fQuit == false)
    {
        cout << "(0)Beenden (1)Werte aendern "
            << "(2)Quadrat (3)Dritte Potenz (4)Vertauschen: ";
        cin >> choice;
        switch (choice)
        {
        case 1:
            pFunc = GetVals;
            break;
        case 2:
            pFunc = Square;
            break;
        case 3:
            pFunc = Cube;
            break;
```

```
36:         case 4:
37:             pFunc = Swap;
38:             break;
39:         default:
40:             fQuit = true;
41:             break;
42:         }
43:         if (fQuit == true)
44:             break;
45:         PrintVals ( pFunc, valOne, valTwo);
46:     }
47:     return 0;
48: }
49:
50: void PrintVals( VPF pFunc,int& x, int& y)
51: {
52:     cout << "x: " << x << " y: " << y << endl;
53:     pFunc(x,y);
54:     cout << "x: " << x << " y: " << y << endl;
55: }
56:
57: void Square (int & rX, int & rY)
58: {
59:     rX *= rX;
60:     rY *= rY;
61: }
62:
63: void Cube (int & rX, int & rY)
64: {
65:     int tmp;
66:
67:     tmp = rX;
68:     rX *= rX;
69:     rX = rX * tmp;
70:
71:     tmp = rY;
72:     rY *= rY;
73:     rY = rY * tmp;
74: }
75:
76: void Swap(int & rX, int & rY)
77: {
78:     int temp;
79:     temp = rX;
80:     rX = rY;
81:     rY = temp;
82: }
83:
```

```
84: void GetVals (int & rValOne, int & rValTwo)
85: {
86:     cout << "Neuer Wert fuer ValOne: ";
87:     cin >> rValOne;
88:     cout << "Neuer Wert fuer ValTwo: ";
89:     cin >> rValTwo;
90: }
```

```
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 1
x: 1 y: 2
Neuer Wert fuer ValOne: 2
Neuer Wert fuer ValTwo: 3
x: 2 y: 3
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 3
x: 2 y: 3
x: 8 y: 27
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 2
x: 8 y: 27
x: 64 y: 729
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 4
x: 64 y: 729
x: 729 y: 64
(0)Beenden (1)Werte aendern (2)Quadrat (3)Dritte Potenz
(4)Vertauschen: 0
```

In Zeile 9 wird `VPF` mittels `typedef` vom Typ »Funktion, die `void` zurückgibt und zwei Referenzen auf `int` als Parameter übernimmt« deklariert.

Zeile 10 deklariert die Funktion `PrintVals()` mit drei Parametern: einem Parameter vom Typ `VPF` und zwei Referenzen auf `int`. Zeile 18 deklariert nun `pFunc` vom Typ `VPF`.

Nach Einführung des Typs `VPF` sind alle folgenden Deklarationen von `pFunc` und `PrintVals()` wesentlich besser zu lesen. Denken Sie daran, dass `typedef` ein Synonym erzeugt; der einzige Unterschied zwischen Listing 20.7 und Listing 20.8 ist die Lesbarkeit!

20.7 Zeiger auf Elementfunktionen

Bis jetzt haben wir nur Funktionszeiger für allgemeine Funktionen, die nicht zu Klassen gehören, erzeugt. Genauso gut kann man Zeiger auf Funktionen erzeugen, die Elemente von Klassen sind.

Bei einem Zeiger auf eine Elementfunktion verwendet man die gleiche Syntax wie bei einem Zeiger auf eine Funktion, schließt aber den Klassennamen und den Zugriffsoperator (`::`) ein. Zeigt zum Beispiel `pFunc` auf eine `Shape`-Elementfunktion, die zwei Integer übernimmt und `void` zurückgibt, lautet die Deklaration für `pFunc` folgendermaßen:

```
void (Shape::*pFunc) (int, int);
```

Zeiger auf Elementfunktionen setzt man in genau der gleichen Weise ein wie Zeiger auf Funktionen. Allerdings erfordern sie ein Objekt der richtigen Klasse, auf dem sie aufgerufen werden. Listing 20.9 zeigt die Verwendung von Zeigern auf Elementfunktionen.

Funktionsname und Adresse

Manche Compiler unterstützen keine Funktionsnamen als Synonym für die Adresse. Wenn der Compiler bei diesem Listing eine Warnung oder eine Fehlermeldung ausgibt, ändern Sie die Zeilen 72 bis 77 wie folgt ab:

```
72:         case 1:
73:             pFunc = & Mammal::Speak;
74:             break;
75:         default:
76:             pFunc = & Mammal::Move;
77:             break;
```

Führen Sie diesbezügliche Änderungen auch an allen nachfolgenden Listings durch. (Beachten Sie das zusätzliche Et-Zeichen (`&`) auf jeder Zeile.)

Beim Borland-Compiler gibt es keine Probleme.

Listing 20.9: ptrtomember.cpp – Zeiger auf Elementfunktionen

```
 1: // Listing 20.9 Zeiger auf Elementfunktionen
 2: #include <iostream>
 3:
 4: enum BOOL {FALSE, TRUE};
 5:
 6: class Mammal
 7: {
 8: public:
 9:     Mammal():itsAge(1) {  }
10:     virtual ~Mammal() { }
```

```
11:     virtual void Speak() const = 0;
12:     virtual void Move() const = 0;
13: protected:
14:     int itsAge;
15: };
16:
17: class Dog : public Mammal
18: {
19: public:
20:     void Speak()const { std::cout << "Wuff!\n"; }
21:     void Move() const { std::cout << "Laufen...\n"; }
22: };
23:
24: class Cat : public Mammal
25: {
26: public:
27:     void Speak()const { std::cout << "Miau!\n"; }
28:     void Move() const { std::cout << "Schleichen...\n"; }
29: };
30:
31: class Horse : public Mammal
32: {
33: public:
34:     void Speak()const { std::cout << "Wieher!\n"; }
35:     void Move() const { std::cout << "Galoppieren...\n"; }
36: };
37:
38: int main()
39: {
40:     void (Mammal::*pFunc)() const =0;
41:     Mammal* ptr =0;
42:     int Animal;
43:     int Method;
44:     bool fQuit = false;
45:
46:     while (fQuit == false)
47:     {
48:         std::cout << "(0)Beenden (1)Hund (2)Katze (3)Pferd: ";
49:         std::cin >> Animal;
50:         switch (Animal)
51:         {
52:         case 1:
53:             ptr = new Dog;
54:             break;
55:         case 2:
56:             ptr = new Cat;
57:             break;
```

```
58:         case 3:
59:             ptr = new Horse;
60:             break;
61:         default:
62:             fQuit = true;
63:             break;
64:         }
65:         if (fQuit)
66:             break;
67:
68:         std::cout << "(1)Sprechen (2)Bewegen: ";
69:         std::cin >> Method;
70:         switch (Method)
71:         {
72:         case 1:
73:             pFunc = Mammal::Speak;
74:             break;
75:         default:
76:             pFunc = Mammal::Move;
77:             break;
78:         }
79:
80:         (ptr->*pFunc)();
81:         delete ptr;
82:     }
83:     return 0;
84: }
```

```
(0)Beenden (1)Hund (2)Katze (3)Pferd: 1
(1)Sprechen (2)Bewegen: 1
Wuff!
(0)Beenden (1)Hund (2)Katze (3)Pferd: 2
(1)Sprechen (2)Bewegen: 1
Miau!
(0)Beenden (1)Hund (2)Katze (3)Pferd: 3
(1)Sprechen (2)Bewegen: 2
Galoppieren...
(0)Beenden (1)Hund (2)Katze (3)Pferd: 0
```

Die Zeilen 6 bis 15 deklarieren den abstrakten Datentyp `Mammal` mit den zwei abstrakten Methoden `Speak()` und `Move()`. Von `Mammal` leiten sich die Klassen `Dog`, `Cat` und `Horse` ab, die jeweils `Speak()` und `Move()` redefinieren.

Das Rahmenprogramm in `main()` fordert den Benutzer auf, ein Tier auszuwählen. Dann erzeugt es eine neue Unterklasse von `Animal` auf dem Heap und weist sie in den Zeilen 50 bis 64 an `ptr` zu.

Der Benutzer soll nun die aufzurufende Methode wählen. Diese wird dem Zeiger `pFunc` zugewiesen. In Zeile 80 ruft das erzeugte Objekt die gewählte Methode auf. Dabei kommt der Zeiger `ptr` für den Zugriff auf das Objekt und `pFunc` für den Zugriff auf die Funktion zum Einsatz.

Schließlich ruft Zeile 81 `delete` auf dem Zeiger `ptr` auf, um den für das Objekt auf dem Heap reservierten Speicher zurückzugeben. Ein Aufruf von `delete` auf `pFunc` ist nicht erforderlich, da es sich um einen Zeiger auf den Code und nicht auf ein Objekt im Heap handelt. Ein entsprechender Versuch erzeugt sogar einen Compiler-Fehler.

20.7.1 Arrays mit Zeigern auf Elementfunktionen

Genau wie bei Zeigern auf Funktionen lassen sich Zeiger auf Elementfunktionen in einem Array speichern. Das Array kann mit den Adressen der verschiedenen Elementfunktionen initialisiert werden und diese lassen sich durch Indizierung des Arrays aufrufen. Listing 20.10 verdeutlicht diese Technik.

Listing 20.10: arrayptrfunction.cpp – Array mit Zeigern auf Elementfunktionen

```
 1: // Listing 20.10 Array mit Zeigern auf Elementfunktionen
 2: #include <iostream>
 3:
 4: class Dog
 5: {
 6: public:
 7:     void Speak()const { std::cout << "Wuff!\n"; }
 8:     void Move() const { std::cout << "Laufen...\n"; }
 9:     void Eat() const { std::cout << "Fressen...\n"; }
10:     void Growl() const { std::cout << "Grrrrr\n"; }
11:     void Whimper() const { std::cout << "Heulen...\n"; }
12:     void RollOver() const { std::cout << "Tot stellen...\n"; }
13:     void PlayDead() const
14:         { std::cout << "Ist das das Ende von Little Caesar?\n";
}
15: };
16:
17: typedef void (Dog::*PDF)()const ;
18: int main()
19: {
20:     const int MaxFuncs = 7;
21:     PDF DogFunctions[MaxFuncs] =
22:         {   Dog::Speak,
23:             Dog::Move,
24:             Dog::Eat,
25:             Dog::Growl,
26:             Dog::Whimper,
27:             Dog::RollOver,
28:             Dog::PlayDead
```

```
 29:         };
 30:
 31:     Dog* pDog =0;
 32:     int Method;
 33:     bool fQuit = false;
 34:
 35:     while (!fQuit)
 36:     {
 37:         std::cout << "(0)Beenden (1)Sprechen (2)Bewegen
(3)Fressen (4)Knurren";
 38:         std::cout << " (5)Winseln (6)Herumwaelzen (7)Tot
stellen: ";
 39:         std::cin >> Method;
 40:         if (Method == 0)
 41:         {
 42:             fQuit = true;
 43:             break;
 44:         }
 45:         else
 46:         {
 47:             pDog = new Dog;
 48:             (pDog->*DogFunctions[Method-1])();
 49:             delete pDog;
 50:         }
 51:     }
 52:     return 0;
 53: }
```

```
(0)Beenden (1)Sprechen (2)Bewegen (3)Fressen (4)Knurren (5)Winseln
(6)Herumwaelzen (7)Tot stellen: 1
Wuff!
(0)Beenden (1)Sprechen (2)Bewegen (3)Fressen (4)Knurren (5)Winseln
(6)Herumwaelzen (7)Tot stellen: 4
Grrrrr
(0)Beenden (1)Sprechen (2)Bewegen (3)Fressen (4)Knurren (5)Winseln
(6)Herumwaelzen (7)Tot stellen: 7
Ist das das Ende von Little Caesar?
(0)Beenden (1)Sprechen (2)Bewegen (3)Fressen (4)Knurren (5)Winseln
(6)Herumwaelzen (7)Tot stellen: 0
```

Die Zeilen 4 bis 15 erzeugen die Klasse `Dog` mit sieben Elementfunktionen. Rückgabetyp und Signatur sind bei allen Funktionen gleich. Zeile 17 deklariert `PDF` mit `typedef` als Zeiger auf eine Elementfunktion von `Dog`, die keine Parameter übernimmt, keine Werte zurückgibt und konstant ist – entsprechend der Signatur der sieben Elementfunktionen von `Dog`.

In den Zeilen 21 bis 29 wird das Array `DogFunctions` für die Aufnahme von sieben derartigen Elementfunktionen deklariert und mit den Adressen dieser Funktionen initialisiert.

Die Zeilen 37 und 38 fordern den Benutzer zur Auswahl einer Methode auf. Je nach Auswahl – außer bei *Beenden* – wird ein neues `Dog`-Objekt auf dem Heap erzeugt und dann die richtige Methode auf dem Array in Zeile 48 aufgerufen. Die folgende Zeile können Sie ebenfalls einem eingefleischten C++-Programmierer vorlegen und ihn um eine Erklärung bitten:

```
(pDog->*DogFunctions[Method-1])();
```

Diese etwas esoterische Konstruktion lässt sich insbesondere bei Tabellen mit Elementfunktionen hervorragend einsetzen. Das Programm ist dadurch besser zu lesen und verständlicher.

20.8 Fragen und Antworten

F: Warum verwendet man statische Daten, wenn man auch mit globalen Daten arbeiten kann?

A: Der Gültigkeitsbereich von statischen Daten beschränkt sich auf die Klasse. Demzufolge sind die Daten nur über ein Objekt der Klasse zugänglich – über einen expliziten und vollständigen Aufruf unter Verwendung des Klassennamens (bei öffentlichen Daten) oder über eine statische Elementfunktion. Allerdings sind statische Daten an den Klassentyp gebunden. Durch den eingeschränkten Zugriff und die strenge Typisierung sind statische Daten sicherer als globale Daten.

F: Warum setzt man statische Elementfunktionen ein, wenn man globale Funktionen verwenden kann?

A: Der Gültigkeitsbereich statischer Elementfunktionen ist auf die Klasse beschränkt. Man kann diese Funktionen nur mithilfe eines Objekts der Klasse oder einer expliziten und vollständigen Spezifikation (wie zum Beispiel `ClassName::FunctionName()`) aufrufen.

F: Warum macht man nicht alle Klassen zu Freunden aller Klassen, die sie verwenden?

A: Wenn man eine Klasse zum Freund einer anderen macht, legt man die Details der Implementierung offen und verringert die Kapselung. Im Idealfall sollte man möglichst viele Details jeder Klasse vor allen anderen Klassen verbergen.

20.9 Workshop

Nachdem Sie sich mit speziellen Klassen, Funktionen und Zeigern auseinander gesetzt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu festigen.

20.9.1 Kontrollfragen

1. Welchen Zweck erfüllen statische Datenelemente in einer Klasse?
2. Welchen Zweck erfüllen statische Methoden in einer Klasse?
3. Welchen Vorteil bringt es, eine Datei wie `string.hpp` (siehe Listing 20.3) zu erstellen, anstatt sie gleich in Listing 20.4 einzubinden, wo sie verwendet wird?
4. Welchen Vorteil haben Friend-Klassen und -Methoden?

20.9.2 Übungen

1. Entfernen Sie die Kommentare vor den `cout`-Anweisungen in Listing 20.3 (`string.hpp`, Zeilen 37, 51, 63, 75, 83 und 100) und führen Sie das Programm in Listing 20.4 (`employeemain.cpp`) erneut aus, um sich anzusehen, wie oft diese Methoden aufgerufen werden.
2. Wie lässt sich Listing 20.6 (`ptrarrayfunction.cpp`) modifizieren, um Eingaben vom Benutzer in geeigneter Weise zu behandeln? (Hinweis: Welcher Wert wird im Array-Element für einen ungültigen Eintrag gespeichert? Sehen Sie sich auch Zeile 43 an.)
3. Vergleichen Sie die Funktionalität der in Listing 20.3 (`string.hpp`) definierten Klasse `String` mit der Standardbibliothek, die zum Lieferumfang des Compilers gehört. Um in C++BuilderX diese Informationen zu erhalten, wählen Sie HILFE | HILFETHEMEN und klicken auf der Registerkarte INHALT auf *Referenz*. Im Hauptfenster der Hilfe finden Sie den Link *<string>*.

20.9.3 Antworten zu den Kontrollfragen

1. Für alle Objekte dieser Klasse wird nur eine Version instanziert. Hier können Sie wichtige globale Daten über diese Objekte speichern (beispielsweise die Anzahl der erzeugten Objekte).
2. Wie bei statischen Datenelementen wird nur eine Version für alle Objekte dieser Klasse instanziert. Man verwendet statische Methoden üblicherweise, um auf statische Elementdaten zuzugreifen.

3. Indem man eine Include-Datei wie `string.hpp` erstellt, kann man denselben Code in mehreren Modulen oder zusammen mit Team-Mitgliedern wiederverwenden. Diese Verfahrensweise gehört zum guten Programmierstil. Der Compiler verlangt dies nicht, bietet aber die Möglichkeit.

4. Friends erlauben die gemeinsame Nutzung unter sich. So wie `proteced` den abgeleiteten Klassen erlaubt, auf Elementdaten und -funktionen zuzugreifen, während das Außenstehenden nicht möglich ist (`protected` liegt zwischen `public` und `private`), erlaubt `friend` das Gleiche für bestimmte Klassen und Methoden innerhalb von Funktionen.

KAPITEL 21

jetzt lerne ich

Der Präprozessor

In dieser Stunde lernen Sie,

- was bedingte Kompilierung ist und wie man sie handhabt,
- wie man Makros mithilfe des Präprozessors schreibt,
- wie man den Präprozessor bei der Fehlersuche einsetzt.

21.1 Präprozessor und Compiler

Bei jeder Ausführung des Compilers startet als Erstes der Präprozessor. Dieser sucht nach Anweisungen, die mit einem Nummernzeichen (#) beginnen. Derartige Anweisungen bewirken eine Änderung des Quelltextes. Im Ergebnis entsteht eine neue Quellcodedatei: eine temporäre Datei, die man normalerweise nicht zu Gesicht bekommt, die man aber vom Compiler speichern lassen kann, um sie bei Bedarf zu untersuchen.

Der Compiler liest und kompiliert nicht die originale Quellcodedatei, sondern die Ausgabe des Präprozessors. Die Anweisung `#include` teilt dem Präprozessor mit, die Datei mit dem auf die `#include`-Direktive folgenden Namen zu suchen und sie an dieser Stelle in die Zwischendatei zu schreiben. Das verhält sich genauso, als hätten Sie selbst die gesamte Datei direkt in Ihre Quellcodedatei geschrieben. Zu dem Zeitpunkt, zu dem der Compiler den Quellcode zu sehen bekommt, befindet sich die eingebundene Datei bereits an der richtigen Stelle.

21.2 Das Zwischenformat ansehen

Bei nahezu jedem Compiler kann man über einen »Schalter« veranlassen, dass der Compiler die Zwischendatei speichert. Diesen Schalter setzt man entweder in der integrierten Entwicklungsumgebung oder auf der Befehlszeile. Eine Auflistung und Erläuterung der Schalter finden Sie im Handbuch zu Ihrem Compiler. Dort erfahren Sie auch, welcher Schalter für das Speichern der Zwischendatei infrage kommt, falls Sie diese Datei untersuchen möchten.

21.3 Die Anweisung #define

Der Befehl `#define` definiert eine String-Ersetzung. Schreibt man zum Beispiel

```
#define BIG 512
```

hat man den Präprozessor angewiesen, den String `BIG` an allen Stellen gegen den String `512` auszutauschen. Es handelt sich hier aber nicht um einen String im Sinne von C++. Die Zeichenfolge `512` wird im Quellcode an allen Stellen, wo das Token `BIG` erscheint, buchstabengetreu ersetzt. Ein *Token* ist eine Zeichenfolge, die sich überall dort einsetzen lässt, wo man einen String, eine Konstante oder eine andere Buchstabengruppe verwenden könnte. Schreibt man demzufolge

```
#define BIG 512
int myArray[BIG];
```

sieht die vom Präprozessor produzierte Zwischendatei folgendermaßen aus:

```
int myArray[512];
```

Die `#define`-Anweisung ist also verschwunden. Die Anweisungen an den Präprozessor werden alle aus der Zwischendatei entfernt und erscheinen überhaupt nicht im endgültigen Quellcode.

21.3.1 Konstanten mit #define erzeugen

Mit `#define` lassen sich unter anderem Konstanten ersetzen. Allerdings ist das fast nie zu empfehlen, da `#define` lediglich eine String-Ersetzung vornimmt und keine Typprüfung durchführt. Die Verwendung des Schlüsselwortes `const` bringt erhebliche Vorteile gegenüber `#define`.

21.3.2 Testen mittels #define und #ifdef

Man kann mit `#define` auch einfach deklarieren, dass eine bestimmte Zeichenfolge definiert ist. Zum Beispiel schreibt man:

```
#define BIG
```

Später prüft man auf die Definition von `BIG` und leitet entsprechende Aktivitäten ein. Die Präprozessor-Anweisungen `#ifdef` und `#ifndef` testen, ob ein String definiert bzw. nicht definiert wurde. Bei beiden Befehlen muss vor dem Blockende (das heißt, vor der nächsten schließenden geschweiften Klammer) ein abschließender `#endif`-Befehl erscheinen.

Der Befehl `#ifdef` liefert `true`, wenn die getestete Zeichenfolge bereits definiert ist. Man kann also schreiben:

```
#ifdef DEBUG
cout << "DEBUG definiert";
#endif
```

Trifft der Präprozessor auf `#ifdef`, durchsucht er eine von ihm angelegte Tabelle, ob die Zeichenfolge `DEBUG` definiert ist. Sollte das der Fall sein, liefert `#ifdef` das Ergebnis `true` und alles bis zum nächsten `#else` oder `#endif` wird in die Zwischendatei für die Kompilierung geschrieben. Ergibt die Auswertung von `#ifdef` den Wert `false`, schreibt der Präprozessor keine der Anweisungen zwischen `#ifdef DEBUG` und `#endif` in die Zwischendatei, praktisch so, als hätten diese Anweisungen niemals in der Quelldatei gestanden.

Das logische Gegenstück zu `#ifdef` ist `#ifndef`. Diese Anweisung liefert `true`, wenn die Zeichenfolge bis zu diesem Punkt noch nicht in der Datei definiert wurde.

21.3.3 Der Präprozessor-Befehl #else

Wie Sie vielleicht erraten, lässt sich der Befehl `#else` zwischen `#ifdef` bzw. `#ifndef` und dem schließenden `#endif` einfügen. Listing 21.1 zeigt Einsatzbeispiele für diese Anweisungen.

Listing 21.1: using-define.cpp – Einsatz von #define

```
 1: // Listing 21.1 Einsatz von #define
 2: #define DemoVersion
 3: #define DOS_VERSION 5
 4: #include <iostream>
 5: int main()
 6: {
 7:     std::cout << "Auf Definitionen von DemoVersion, "
 8:         << "DOS_VERSION und WINDOWS_VERSION pruefen...\n";
 9:
10: #ifdef DemoVersion
11:     std::cout << "DemoVersion definiert.\n";
```

```
12: #else // DemoVersion
13:     std::cout << "DemoVersion nicht definiert.\n";
14: #endif // DemoVersion
15:
16: #ifndef DOS_VERSION
17:     std::cout << "DOS_VERSION nicht definiert!\n";
18: #else // DOS_VERSION
19:     std::cout << "DOS_VERSION definiert als: "
20:          << DOS_VERSION << std::endl;
21: #endif // DOS_VERSION
22:
23: #ifdef WINDOWS_VERSION
24:     std::cout << "WINDOWS_VERSION definiert!\n";
25: #else // WINDOWS_VERSION
26:     std::cout << "WINDOWS_VERSION wurde nicht definiert.\n";
27: #endif // WINDOWS_VERSION
28:
29:     std::cout << "Fertig.\n";
30:     return 0;
31: }
```

```
Auf Definitionen von DemoVersion, DOS_VERSION und WINDOWS_VERSION
pruefen...
DemoVersion definiert.
DOS_VERSION definiert als: 5
WINDOWS_VERSION wurde nicht definiert.
Fertig.
```

Die Zeilen 1 und 2 definieren `DemoVersion` und `DOS_VERSION`, wobei `DOS_VERSION` mit dem String `5` definiert ist. Zeile 10 testet die Definition von `DemoVersion`. Da `DemoVersion` – wenn auch ohne Wert – definiert ist, liefert der Test das Ergebnis `true` und der String in Zeile 11 wird ausgegeben.

Der Test in Zeile 16 prüft, ob `DOS_VERSION` *nicht* definiert ist. Wie wir eben festgestellt haben, ist `DOS_VERSION` aber definiert. Damit liefert der Test das Ergebnis `false` und es wird Zeile 19 statt Zeile 17 kompiliert. Hier findet die Ersetzung des Wortes `DOS_VERSION` durch den String `5` statt. Dem Compiler stellt sich diese Zeile dann wie folgt dar:

```
std::cout << "DOS_VERSION definiert als: " << 5 << endl;
```

Das erste Vorkommen von `DOS_VERSION` in dieser Anweisung wird nicht ersetzt, da der String in Anführungszeichen steht. Die Substitution erfolgt nur für das zweite `DOS_VERSION`. Damit findet der Compiler an dieser Stelle eine `5` vor, genauso, als hätte man sie direkt hier eingetippt.

Schließlich testet das Programm in Zeile 23 auf `WINDOWS_VERSION`. Da `WINDOWS_VERSION` nicht definiert ist, liefert der Test das Ergebnis `false` und es wird Zeile 26 statt Zeile 23 kompiliert.

21.4 Schutz vor Mehrfachdeklarationen

Projekte bestehen gewöhnlich aus einer größeren Anzahl von Dateien. Dabei kann jede Klasse eine eigene Header-Datei (`.hpp`) mit der Klassendeklaration und eine Implementierungsdatei (`.cpp`) mit dem Quellcode der Klassenmethoden erhalten.

Die Funktion `main()` steht in einer separaten `.cpp`-Datei. Alle `.cpp`-Dateien werden zu `.obj`-Dateien kompiliert und mit dem Linker zu einem einzigen Programm gebunden.

Da Ihre Programme auf Methoden aus vielen Klassen zurückgreifen, sind in jede Datei mehrere Header-Dateien aufzunehmen. Außerdem müssen sich Header-Dateien oftmals untereinander einbinden. Beispielsweise muss die Header-Datei für die Deklaration einer abgeleiteten Klasse die Header-Datei für ihre Basisklasse einschließen.

Nehmen wir an, dass die Klasse `Animal` in der Datei `animal.hpp` deklariert ist. Die von `Animal` abgeleitete Klasse `Dog` muss die Datei `animal.hpp` in `dog.hpp` einbinden, da sich `Dog` sonst nicht von `Animal` ableiten lässt. Der `Cat`-Header schließt `animal.cpp` aus demselben Grund ein. Wenn man eine Methode erzeugt, die sowohl `Cat` als auch `Dog` verwendet, läuft man Gefahr, `animal.hpp` zweimal einzubinden.

Das Ganze führt zu einem Compiler-Fehler, da es nicht zulässig ist, eine Klasse (`Animal`) zweimal zu deklarieren, selbst wenn die Deklarationen identisch sind. Dieses Problem lässt sich mit entsprechenden Schutzvorkehrungen lösen. Am Beginn der Header-Datei `animal.hpp` schreiben Sie folgende Zeilen:

```
#ifndef ANIMAL_HPP
#define ANIMAL_HPP
...             // Hier steht der gesamte Inhalt der Datei
#endif
```

Diese Zeilen sagen aus: »Wenn der Begriff `ANIMAL_HPP` noch nicht definiert ist, dann definiere ihn jetzt.« Zwischen der `#define`-Anweisung und dem schließenden `#endif` steht der gesamte Inhalt der Datei.

Wenn das Programm erstmalig diese Datei einbindet und der Compiler die erste Zeile auswertet, liefert der Test `true`. Das heißt: `ANIMAL_HPP` ist noch nicht definiert. Demzufolge holt der Compiler jetzt die Definition von `ANIMAL_HPP` nach und bindet dann die gesamte Datei ein.

Wenn Ihr Programm die Datei ANIMAL.HPP ein zweites Mal einbindet, liefert die Auswertung der ersten Zeile das Ergebnis false, da ANIMAL_HPP nun definiert ist. Damit springt das Programm zur nächsten #else-Klausel (hier gibt es keine) oder zur nächsten #endif-Anweisung (am Ende der Datei) und übergeht damit den gesamten Inhalt der Datei. Die Klasse wird damit nicht zweimal deklariert.

Der tatsächliche Name des definierten Symbols (ANIMAL_HPP) spielt keine Rolle. Es ist üblich, den Dateinamen zu verwenden, ihn durchweg in Großbuchstaben zu schreiben und dabei den Punkt zwischen Dateiname und Erweiterung durch einen Unterstrich zu ersetzen. Allerdings ist das nur eine Konvention.

Es schadet nichts, Schutzmaßnahmen gegen Mehrfachdeklarationen generell vorzusehen. Oftmals spart man sich dadurch eine stundenlange Fehlersuche.

21.4.1 Definitionen auf der Befehlszeile

Fast alle C++-Compiler erlauben die Definition von Werten in der Befehlszeile und über die integrierte Entwicklungsumgebung (wobei gewöhnlich beides möglich ist). Daher kann man die Zeilen 1 und 2 aus Listing 21.1 weglassen und DemoVersion bzw. DOS_VERSION von der Befehlszeile aus bei bestimmten Kompilierungen definieren und bei anderen nicht.

Es ist üblich, speziellen Code zur Fehlersuche (Debugging) zwischen #ifdef DEBUG und #endif einzuschließen. Damit lässt sich der gesamte Debugging-Code leicht aus der Quelldatei entfernen, wenn man die endgültige Version kompiliert: Man definiert einfach nicht den Begriff DEBUG.

21.4.2 Definitionen aufheben

Wenn man einen Namen definiert hat und diese Definition aus dem Code heraus aufheben möchte, kann man die Anweisung #undef verwenden, die entgegengesetzt zur Anweisung #define arbeitet.

21.4.3 Bedingte Kompilierung

Kombiniert man #define bzw. Definitionen auf der Befehlszeile mit #ifdef, #else und #ifndef, lassen sich Programme mit unterschiedlichem Code kompilieren, abhängig davon, was bereits mit #define definiert ist. Damit kann man zum Beispiel einen einheitlichen Satz von Quellcodedateien erstellen und für verschiedenartige Plattformen – etwa DOS und Windows – kompilieren.

Später in diesem Kapitel zeigt der Abschnitt »Das Makro assert()« eine andere gebräuchliche Anwendung dieser Technik: die bedingte Kompilierung einzelner Code-Abschnitte bei der Fehlersuche.

21.5 Makros

Mit der `#define`-Direktive lassen sich auch Makrofunktionen – oder kurz: Makros – erzeugen. Ein Makro ist ein Symbol, das mithilfe von `#define` erzeugt wurde und in der Art einer Funktion ein Integer-Argument übernimmt. Der Präprozessor substituiert den Ersetzungsstring durch das jeweils übergebene Argument. Beispielsweise kann man das Makro `TWICE` (Verdoppeln) wie folgt definieren:

```
#define TWICE(x) ( (x) * 2 )
```

Im Code schreibt man dann zum Beispiel

```
TWICE(4)
```

Der gesamte String `TWICE(4)` wird entfernt und durch den Wert `8` ersetzt. Trifft der Präprozessor auf die `4`, setzt er dafür `( (4) * 2 )`, was dann zu `4 * 2` oder `8` ausgewertet wird.

Ein Makro kann über mehrere Parameter verfügen, wobei man jeden Parameter wiederholt im Ersetzungstext verwenden kann. Zwei häufig anzutreffende Makros sind `MAX` und `MIN`:

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

Beachten Sie, dass in einer Makro-Definition die öffnende Klammer für die Parameterliste unmittelbar auf den Makronamen folgen muss – ohne Leerzeichen dazwischen. Der Präprozessor ist nicht so nachsichtig mit Whitespaces wie der Compiler.

Wenn man etwa schreibt

```
#define MAX (x,y) ( (x) > (y) ? (x) : (y) )
```

und anschließend versucht, `MAX` in der Anweisungsfolge

```
int x = 5, y = 7, z;
z = MAX(x,y);
```

zu verwenden, erhält man den Zwischencode

```
int x = 5, y = 7, z;
z = (x,y) ( (x) > (y) ? (x) : (y) ) (x,y)
```

Es findet eine einfache Textersetzung statt und nicht der Aufruf des Makros. Das Token `MAX` wird durch `(x,y) ( (x) > (y) ? (x) : (y) )` substituiert, woran sich das nach `MAX` angegebene `(x,y)` anschließt.

Entfernt man das Leerzeichen zwischen `MAX` und `(x,y)`, erhält man dagegen den folgenden Zwischencode:

```
int x = 5, y = 7, z;
z =7;
```

21.5.1 Warum so viele Klammern?

Vielleicht fragen Sie sich, warum in den bisher präsentierten Makros so viele Klammern vorkommen. Der Präprozessor ist nicht darauf angewiesen, dass Klammern um die Argumente in der Ersetzungszeichenfolge stehen. Allerdings helfen Ihnen die Klammern, unerwünschte Nebeneffekte zu vermeiden, wenn Sie komplizierte Werte an ein Makro übergeben. Definieren Sie zum Beispiel `MAX` als

```
#define MAX(x,y) x > y ? x : y
```

und übergeben die Werte `5` und `7`, funktioniert das Makro wie erwartet. Die Übergabe komplizierterer Ausdrücke führt aber zu unerwarteten Ergebnissen wie es Listing 21.2 demonstriert.

Listing 21.2: using-paren.cpp – Klammern in Makros

```
 1: // Listing 21.2 Makro-Erweiterung
 2: #include <iostream>
 3:
 4: #define CUBE(a) ( (a) * (a) * (a) )
 5: #define THREE(a) a * a * a
 6:
 7: int main()
 8: {
 9:     long x = 5;
10:     long y = CUBE(x);
11:     long z = THREE(x);
12:
13:     std::cout << "y: " << y << std::endl;
14:     std::cout << "z: " << z << std::endl;
15:
16:     long a = 5, b = 7;
17:     y = CUBE(a+b);
18:     z = THREE(a+b);
19:
20:     std::cout << "y: " << y << std::endl;
21:     std::cout << "z: " << z << std::endl;
22:     return 0;
23: }
```

```
y: 125
z: 125
y: 1728
z: 82
```

Zeile 4 definiert das Makro `CUBE`. Bei jedem Aufruf dieses Makros wird das Argument `a` zwischen die Klammern eingefügt. Zeile 5 definiert das Makro `THREE` ohne Klammern.

Bei der ersten Verwendung dieser Makros übergibt man den Wert `5` als Parameter und beide Makros arbeiten wie erwartet. `CUBE(5)` ergibt die Makro-Erweiterung `( (5) * (5) * (5) )` und liefert damit den Wert `125`. Die Erweiterung von `THREE(5)` führt zu `5 * 5 * 5` und wird ebenfalls zu `125` ausgewertet.

Bei der zweiten Verwendung in den Zeilen 16 bis 18 lautet der Parameter `5 + 7`. In diesem Fall liefert `CUBE(5+7)` die Erweiterung

```
( (5+7) * (5+7) * (5+7) )
```

damit

```
( (12) * (12) * (12) )
```

und führt zum Ergebnis 1728. Dagegen wird `THREE(5+7)` zu

```
5 + 7 * 5 + 7 * 5 + 7
```

erweitert. Da die Multiplikation einen höheren Vorrang als die Addition hat, ergibt sich

```
5 + (7 * 5) + (7 * 5) + 7
```

damit

```
5 + (35) + (35) + 7
```

was schließlich das Ergebnis 82 liefert.

21.5.2 Makros, Funktionen und Templates

In C++ sind Makros mit vier Problemen verbunden. Erstens werden große Makros unübersichtlich und sind schwer zu handhaben, da man das gesamte Makro auf einer Zeile definieren muss. Man kann zwar das Makro mit dem Backslash-Zeichen (\) auf einer neuen Zeile fortsetzen, das grundsätzliche Problem bleibt aber bestehen.

Zweitens werden Makros bei jedem Aufruf inline erweitert. Wenn man das Makro ein Dutzend Mal verwendet, erscheint die Substitution zwölf Mal im Programm und nicht nur einmal wie bei einem Funktionsaufruf. Auf der an-

deren Seite ist ein Makro gewöhnlich schneller als eine Funktion, da es den Overhead eines Funktionsaufrufs vermeidet.

Die Inline-Erweiterung führt zu einem dritten Problem: Das Makro erscheint nicht im Zwischencode, den der Compiler verarbeitet. Damit ist das Makro für die meisten Debugger nicht verfügbar und die Fehlersuche in Makros wird zur kniffligen Angelegenheit.

Das letzte Problem ist allerdings das größte: Makros sind nicht typensicher. Es ist zwar bequem, absolut jedes Argument in einem Makro verwenden zu können, doch unterläuft das vollständig die strenge Typisierung von C++. Für C++-Programmierer kommen Makros deshalb nicht infrage. Wie Stunde 23 zeigt, lässt sich dieses Problem mit Templates beseitigen.

21.6 String-Manipulation

Der Präprozessor stellt zwei spezielle Operatoren für die Manipulation von Strings in Makros bereit. Der Operator zur Zeichenkettenbildung (#) wandelt das übergebene Argument in eine Zeichenfolge um. Der Verkettungsoperator (##) verbindet zwei Strings zu einem.

21.6.1 Zeichenkettenbildung

Der Operator zur Zeichenkettenbildung (#, stringizing operator) schließt alle Zeichen, die bis zum nächsten Whitespace auf den Operator folgen, in Anführungszeichen ein. Schreibt man also:

```
#define WRITESTRING(x) cout << #x
```

und ruft dann

```
WRITESTRING(Das ist ein String);
```

auf, wandelt das der Präprozessor wie folgt um:

```
cout << "Das ist ein String";
```

Der String `Das ist ein String` ist nun in Anführungszeichen eingeschlossen, wie es für `cout` erforderlich ist.

21.6.2 Verkettung

Mit dem Verkettungsoperator (##) lassen sich mehrere Terme zu einem neuen Wort verbinden. Das neue Wort ist eigentlich ein Token, das man als Klassenname, Variablenname, Index auf ein Array oder überall, wo eine Buchstabenfolge stehen darf, verwenden kann.

Nehmen wir zunächst fünf Funktionen an, die mit `fOnePrint`, `fTwoPrint`, `fThreePrint`, `fFourPrint` und `fFivePrint` benannt sind. Mit der Deklaration

```
#define fPRINT(x) f ## x ## Print
```

lassen sich dann mit `fPRINT(Two)` die Zeichenfolge `fTwoPrint` und mit `fPRINT(Three)` die Zeichenfolge `fThreePrint` generieren.

In Stunde 19 haben wir eine verkettete Liste entwickelt, die allerdings nur Objekte vom Typ `Data` behandeln konnte. Nehmen wir an, dass diese Liste ausgezeichnet funktioniert. Allerdings wollen wir auch Listen mit Tieren, Autos, Computern usw. aufbauen.

Man könnte nun den Code für `AnimalList`, `CarList`, `ComputerList` schreiben, indem man die entsprechenden Codeabschnitte mehrfach per Ausschneiden und Einfügen erzeugt. Das kann allerdings zum Albtraum ausarten, da jede Änderung an einer Liste in alle anderen zu übernehmen ist.

Als Alternative bieten sich hier Makros und der Verkettungsoperator an. Beispielsweise könnte man Folgendes definieren:

```
#define Listof(Type)  class Type##List \
{ \
public: \
Type##List(){} \
private:          \
int itsLength; \
};
```

Das ist zwar ein sehr vereinfachtes Beispiel, zeigt aber das Konzept, um alle erforderlichen Methoden und Daten zu übernehmen. Um eine `AnimalList` zu erzeugen, schreibt man:

```
Listof(Animal)
```

Das Ganze wird in die Deklaration der Klasse `AnimalList` umgewandelt. Bei diesem Verfahren gibt es einige Probleme, auf die Stunde 23 bei der Behandlung von Templates im Detail eingeht.

21.7 Vordefinierte Makros

Viele Compiler definieren eine Reihe nützlicher Makros. Dazu gehören `__DATE__`, `__TIME__`, `__LINE__` und `__FILE__`. Vor und nach diesen Namen stehen jeweils zwei Unterstriche, um Konflikte mit anderen Namen, die Sie in Ihrem Programm vergeben, nach Möglichkeit zu vermeiden.

Trifft der Compiler auf eines dieser Makros, nimmt er die entsprechende Ersetzung vor. Für `__DATE__` wird das aktuelle Datum substituiert, für `__TIME__` die aktuelle Uhrzeit. Die Makros `__LINE__` und `__FILE__` ersetzt der Compiler

durch die Zeilennummern des Quellcodes bzw. den Dateinamen. Diese Substitution wird ausgeführt, wenn die Quelle vorkompiliert wird, und nicht, wenn das Programm läuft. Fügt man im Programm die Ausgabe des Datums mit `__DATE__` ein, erhält man nicht das aktuelle Datum bei Programmstart, sondern das Datum, zu dem das Programm kompiliert wurde. Bei der Fehlersuche stellen diese Makros eine wertvolle Hilfe dar.

21.7.1 Das Makro assert()

Das Makro `assert()` liefert `true` zurück, wenn der Parameter zu `true` ausgewertet wird. Ergibt die Auswertung des Parameters den Wert `false`, brechen bestimmte Compiler das Programm ab, andere lösen eine Ausnahme aus (siehe Stunde 24).

Ein leistungsfähiges Charakteristikum des Makros `assert()` ist es, dass der Präprozessor überhaupt keinen Code dafür produziert, wenn `DEBUG` nicht definiert ist. Während der Entwicklungsphase ist das Makro eine große Hilfe und wenn man das fertige Produkt vertreibt, gibt es weder eine Leistungseinbuße noch eine Vergrößerung der ausführbaren Programmdatei.

Anstatt sich auf das vom Compiler bereitgestellte `assert()` zu stützen, kann man auch ein eigenes `assert()`-Makro schreiben. Listing 21.3 zeigt dazu ein Beispiel.

Listing 21.3: simple-assert.cpp – Ein einfaches assert()-Makro

```
 1: // Listing 21.3 ASSERTS
 2: #define DEBUG
 3: #include <iostream>
 4:
 5: #ifndef DEBUG
 6:     #define ASSERT(x)
 7: #else
 8:     #define ASSERT(x) \
 9:         if (! (x)) \
10:         { \
11:             std::cout << "FEHLER!! Annahme " << #x << " nicht
zutreffend\n"; \
12:             std::cout << " in Zeile " << __LINE__  << "\n"; \
13:             std::cout << " in Datei " << __FILE__ << "\n";  \
14:         }
15: #endif
16:
17: int main()
18: {
19:     int x = 5;
20:     std::cout << "Erste Annahme: \n";
21:     ASSERT(x==5);
22:     std::cout << "\nZweite Annahme: \n";
23:     ASSERT(x != 5);
```

```
24:     std::cout << "\nFertig.\n";
25:     return 0;
26: }
```

Unterschiedliche Dateiformate

Möglicherweise hat Ihr Compiler die Zeilen 8 bis 14 von Listing 21.3 auf mehrere Zeilen aufgeteilt. Gegebenenfalls müssen Sie die zusätzlichen Leerzeilen löschen, damit sich dieses Beispiel kompilieren lässt. Das Problem tritt auf, weil verschiedene Betriebssysteme, Entwicklungsumgebungen und Compiler das Ende einer Zeile mit unterschiedlichen Zeichen kennzeichnen und diese Zeichen unterschiedlich interpretieren.

Einzeilige Makros

Aus technischer Sicht dürfen Makros nur eine einzige Zeile einnehmen.

Das in den Zeilen 8 bis 14 von Listing 21.3 definierte Makro `ASSERT()` scheint über mehrere Zeilen zu reichen, obwohl es sich tatsächlich nur um eine einzige Zeile handelt. Der Backslash (\) am Ende jeder Zeile setzt das `newline`-Zeichen (Neue Zeile) außer Kraft, das physikalisch dort steht.

Beachten Sie, dass der Backslash nur ein `newline`-Zeichen deaktiviert; hier haben wir es also mit einer der wenigen Stellen zu tun, an denen Whitespaces signifikant sind.

```
Erste Annahme:

Zweite Annahme:
FEHLER!! Annahme x != 5 nicht zutreffend
 in Zeile 24
 in Datei E:\Examples\Hour21\simpleassert.cpp
```

Zeile 2 definiert den Begriff `DEBUG`. Normalerweise erledigt man das von der Befehlszeile (oder der IDE) zur Kompilierzeit, sodass man die Fehlersuche bei Bedarf ein- und ausschalten kann. Die Zeilen 8 bis 14 definieren das Makro `ASSERT()`. In der Regel schreibt man das Ganze in eine Header-Datei und diesen Header (`assert.hpp`) bindet man in alle Implementierungsdateien ein.

In Zeile 5 findet der Test des Begriffs `DEBUG` statt. Ist dieser Begriff nicht angegeben, stellt Zeile 6 eine Definition für `ASSERT()` bereit, die überhaupt keinen Code erzeugt. Ist `DEBUG` definiert, kommt die in den Zeilen 8 bis 14 definierte Funktionalität zum Tragen.

Das Makro ASSERT() selbst besteht aus einer einzigen langen Anweisung, die über sieben Quellcodezeilen verteilt ist. Zeile 9 testet den als Parameter übergebenen Wert. Liefert dieser Test das Ergebnis false, geben die Anweisungen in den Zeilen 11 bis 13 eine Fehlermeldung aus. Wenn der übergebene Wert das Ergebnis true liefert, finden keine Aktionen statt.

Fehlersuche mit assert()

Wenn man ein Programm schreibt, ist man oft hundertprozentig sicher, dass bestimmte Bedingungen wahr sind: eine Funktion liefert einen bestimmten Wert, ein Zeiger ist gültig usw. Es liegt in der Natur der Fehler, dass diese Annahmen unter bestimmten Bedingungen nicht zutreffen. Beispielsweise stürzt ein Programm ab, obwohl man absolut sicher war, dass ein Zeiger gültig ist. Bei der Suche nach Fehlern dieser Art kann Sie assert() unterstützen. Dazu müssen Sie es sich aber zur Gewohnheit machen, die assert()-Anweisungen großzügig im Code vorzusehen. Bei jeder Zuweisung oder Übergabe eines Zeigers als Parameter oder Rückgabewert einer Funktion sollten Sie mit assert() prüfen, ob dieser Zeiger gültig ist. Ist ein Codeabschnitt von einem bestimmten Wert einer Variablen abhängig, verifizieren Sie mit assert(), dass die Annahmen auch zutreffen.

Der häufige Einsatz von assert()-Anweisungen zieht keine Leistungseinbußen nach sich. Diese Anweisungen werden vollständig aus dem Code entfernt, wenn man die Fehlersuche über die entsprechenden #define-Anweisungen deaktiviert. Darüber hinaus stellen die assert()-Anweisungen eine gute interne Dokumentation dar, die den Leser darauf hinweisen, was der Programmierer an einem bestimmten Punkt im Programmablauf als wahr angenommen hat.

Seiteneffekte

Es ist nicht ungewöhnlich, dass sich ein Fehler erst zeigt, nachdem man die assert()-Anweisungen aus dem Code entfernt hat. Fast immer hängt das mit unbeabsichtigten Seiteneffekten zusammen, die in assert()-Anweisungen und anderen, ausschließlich für die Fehlersuche vorgesehenen Code-Abschnitten, auftreten. Zum Beispiel erzeugt man mit der Anweisung

```
ASSERT (x = 5)
```

einen besonders üblen Fehler, da man eigentlich die Bedingung x == 5 testen wollte.

Nehmen wir an, dass Sie unmittelbar vor dieser assert()-Anweisung eine Funktion aufgerufen haben, die x gleich 0 gesetzt hat. Von der obigen assert()-Anweisung nehmen Sie an, dass sie einen Test auf x gleich 5 vornimmt. Stattdessen setzen Sie x gleich 5. Der Test liefert true, da x = 5 nicht

nur x auf 5 setzt, sondern auch den Wert 5 zurückgibt. Und da 5 ungleich null ist, ergibt die Auswertung das Ergebnis true.

Unmittelbar nach der assert()-Anweisung ist dann x wirklich gleich 5 (Sie haben es ja so gewollt!). Ihr Programm läuft wunderbar. Daher bereiten Sie es für den Vertrieb vor und schalten die Unterstützung für die Fehlersuche aus. Nun verschwindet die assert()-Anweisung und x wird nicht mehr auf 5 gesetzt. Da x aber unmittelbar vor dieser assert()-Anweisung den Wert 0 erhalten hat, bleibt dieser Wert auch 0 und das Programm stürzt an dieser Stelle ab (oder macht anderen Unsinn).

In Ihrer Verzweiflung aktivieren Sie die assert()-Anweisungen wieder und siehe da: Der Fehler hat sich verkrümelt. Mit der Zeit können derartige Dinge ganz schön auf die Nerven gehen. Achten Sie also in Ihrem Code zur Fehlersuche genau auf mögliche Seiteneffekte. Tritt ein Fehler nur dann auf, wenn diese Unterstützung abgeschaltet ist, sehen Sie sich den betreffenden Code an und halten Ausschau nach gemeinen Seiteneffekten.

Seiteneffekte können in vielen Makros auftreten

Wie Listing 21.2 zeigt, lassen sich Makros nicht nur mit einzelnen Variablen oder Werten (wie in den Zeilen 10 und 11) aufrufen, sondern auch mit einem komplexen Ausdruck (wie in den Zeilen 17 und 18). Wenn jedoch der komplexe Ausdruck eine Variable ändert, kann das eine unbeabsichtigte Wirkung haben. Nehmen wir beispielsweise an, Sie rufen CUBE mit der Anweisung

```
y = CUBE (a++);
```

auf a++ auf und erwarten, dass a nach Ausführung der Anweisung um 1 inkrementiert ist. Allerdings wird das Makro zu

```
( (a++) * (a++) * (a++) )
```

erweitert und a demzufolge dreimal inkrementiert und nicht wie angenommen nur einmal.

Klasseninvarianten

In vielen Klassen gibt es Bedingungen, die auch nach Abschluss einer Elementfunktion immer true sein sollten. Diese Klasseninvarianten sind die unerlässlichen Bedingungen Ihrer Klasse. Zum Beispiel kann die Forderung bestehen, dass ein CIRCLE-Objekt immer einen Radius größer als 0 hat oder dass das Alter in einem ANIMAL-Objekt im Wertebereich zwischen 0 und 100 liegt.

In diesem Zusammenhang erweist sich eine Invariants()-Methode als hilfreich, die nur dann true zurückgibt, wenn alle diese Bedingungen noch wahr sind. Am Anfang und Ende jeder Klassenmethode baut man dann assert(In-

variants()) ein. Lediglich vor Aufruf des Konstruktors und nach Abschluss des Destruktors kann man von der Invariants()-Methode kein true-Ergebnis erwarten. Listing 21.4 demonstriert die Verwendung der Invariants()-Methode in einer trivialen Klasse.

Listing 21.4: using-invariants.cpp – Die Methode Invariants()

```
// Listing 21.4 Invariants
#define DEBUG
#define SHOW_INVARIANTS
#include <iostream>
#include <string.h>

#ifndef DEBUG
#define ASSERT(x)
#else
#define ASSERT(x) \
    if (! (x)) \
    { \
        std::cout << "FEHLER!! Annahme " << #x << " nicht
zutreffend\n"; \
        std::cout << " in Zeile " << __LINE__  << "\n"; \
        std::cout << " in Datei " << __FILE__ << "\n";  \
    }
#endif

class String
{
public:
    // Konstruktoren
    String();
    String(const char *const);
    String(const String &);
    ~String();

    char & operator[](int offset);
    char operator[](int offset) const;

    String & operator= (const String &);
    int GetLen()const { return itsLen; }
    const char * GetString() const { return itsString; }
    bool Invariants() const;

private:
    String (int);         // Privater Konstruktor
    char * itsString;
    unsigned short itsLen;
};

```

```
42: // Standardkonstruktor erzeugt String von 0 Bytes Länge
43: String::String()
44: {
45:     itsString = new char[1];
46:     itsString[0] = '\0';
47:     itsLen=0;
48:     ASSERT(Invariants());
49: }
50:
51: // Privater (Hilfs-) Konstruktor, nur von Klassen-
52: // methoden für das Erzeugen eines neuen Strings der
53: // erforderlichen Größe verwendet.  Mit Null gefüllt.
54: String::String(int len)
55: {
56:     itsString = new char[len+1];
57:     for (int i = 0; i<=len; i++)
58:         itsString[i] = '\0';
59:     itsLen=len;
60:     ASSERT(Invariants());
61: }
62:
63: // Konvertiert ein Zeichen-Array in einen String
64: String::String(const char * const cString)
65: {
66:     itsLen = strlen(cString);
67:     itsString = new char[itsLen+1];
68:     for (int i = 0; i<itsLen; i++)
69:         itsString[i] = cString[i];
70:     itsString[itsLen]='\0';
71:     ASSERT(Invariants());
72: }
73:
74: // Kopierkonstruktor
75: String::String (const String & rhs)
76: {
77:     itsLen=rhs.GetLen();
78:     itsString = new char[itsLen+1];
79:     for (int i = 0; i<itsLen;i++)
80:         itsString[i] = rhs[i];
81:     itsString[itsLen] = '\0';
82:     ASSERT(Invariants());
83: }
84:
85: // Destruktor, gibt reservierten Speicher frei
86: String::~String ()
87: {
88:     ASSERT(Invariants());
89:     delete [] itsString;
```

```
 90:     itsLen = 0;
 91: }
 92:
 93: // Gleich-Operator, gibt vorhandenen Speicher frei,
 94: // kopiert dann String und Größe
 95: String& String::operator=(const String & rhs)
 96: {
 97:     ASSERT(Invariants());
 98:     if (this == &rhs)
 99:         return *this;
100:     delete [] itsString;
101:     itsLen=rhs.GetLen();
102:     itsString = new char[itsLen+1];
103:     for (int i = 0; i<itsLen;i++)
104:         itsString[i] = rhs[i];
105:     itsString[itsLen] = '\0';
106:     ASSERT(Invariants());
107:     return *this;
108: }
109:
110: // Nicht-konstanter Offset-Operator, gibt Referenz
111: // auf Zeichen zurück, das sich damit ändern
112: // lässt!
113: char & String::operator[](int offset)
114: {
115:     ASSERT(Invariants());
116:     if (offset > itsLen)
117:         return itsString[itsLen-1];
118:     else
119:         return itsString[offset];
120:     ASSERT(Invariants());
121: }
122:
123: // Konstanter Offset-Operator für konstante
124: // Objekte (siehe Kopierkonstruktor!)
125: char String::operator[](int offset) const
126: {
127:     ASSERT(Invariants());
128:     if (offset > itsLen)
129:         return itsString[itsLen-1];
130:     else
131:         return itsString[offset];
132:     ASSERT(Invariants());
133: }
134:
135: // Sicherstellen, dass der String eine bestimmte Länge hat, oder
136: // der Zeiger ist NULL und die Länge ist null.
137: bool String::Invariants() const
```

```
138: {
139: #ifdef SHOW_INVARIANTS
140:     std::cout << " String OK ";
141: #endif
142:     return ( (itsLen && itsString) || (!itsLen && !itsString) );
143: }
144:
145: class Animal
146: {
147: public:
148:     Animal():itsAge(1),itsName("John Q. Animal")
149:         {ASSERT(Invariants());}
150:     Animal(int, const String&);
151:     ~Animal(){}
152:     int GetAge() { ASSERT(Invariants()); return itsAge;}
153:     void SetAge(int Age)
154:         {
155:             ASSERT(Invariants());
156:             itsAge = Age;
157:             ASSERT(Invariants());
158:         }
159:     String& GetName() { ASSERT(Invariants()); return itsName;  }
160:     void SetName(const String& name)
161:         {
162:             ASSERT(Invariants());
163:             itsName = name;
164:             ASSERT(Invariants());
165:         }
166:     bool Invariants();
167: private:
168:     int itsAge;
169:     String itsName;
170: };
171:
172: Animal::Animal(int age, const String& name):
173: itsAge(age),
174: itsName(name)
175: {
176:     ASSERT(Invariants());
177: }
178:
179: bool Animal::Invariants()
180: {
181: #ifdef SHOW_INVARIANTS
182:     std::cout << " Animal OK ";
183: #endif
184:     return (itsAge > 0 && itsName.GetLen());
185: }
```

```
186:
187: int main()
188: {
189:     Animal sparky(5,"Sparky");
190:     std::cout << "\n" << sparky.GetName().GetString() << " ist
";
191:     std::cout << sparky.GetAge() << " Jahre alt.";
192:     sparky.SetAge(8);
193:     std::cout << "\n" << sparky.GetName().GetString() << " ist
";
194:     std::cout << sparky.GetAge() << " Jahre alt.";
195:    return 0;
196: }
```

```
String OK  String OK  String OK  String OK
String OK  String OK  String OK  String OK
Animal OK  String OK  Animal OK
Sparky ist  Animal OK 5 Jahre alt. Animal OK  Animal OK  Animal OK
Sparky ist  Animal OK 8 Jahre alt. String OK
```

Warnungen des Compilers

Der Compiler gibt möglicherweise Warnungen wie die folgenden aus, dass bestimmte `ASSERT()`-Makros niemals ausgeführt werden. Auch wenn die Platzierung der Makros ungewöhnlich scheint, sie folgt dem Muster, ein `ASSERT()` am Anfang und Ende jeder Funktion einzufügen.

```
"usinginvariants.cpp": W8066 Code wird nie erreicht in Funktion
String::operator [](int) in Zeile 120

"usinginvariants.cpp": W8070 Funktion sollte einen Wert zurückgeben
in Funktion String::operator [](int) in Zeile 121

"usinginvariants.cpp": W8066 Code wird nie erreicht in Funktion
String::operator [](int) const in Zeile 132

"usinginvariants.cpp": W8070 Funktion sollte einen Wert zurückgeben
in Funktion String::operator [](int) const in Zeile 133
```

Die Zeilen 7 bis 17 definieren das Makro `ASSERT()`. Ist `DEBUG` definiert, gibt dieses Makro eine Fehlermeldung aus, wenn es das Ergebnis `false` liefert.

Zeile 34 deklariert die Elementfunktion `Invariants()` der Klasse `String`. Die Definition steht in den Zeilen 137 bis 143. Die Konstruktoren werden in den Zeilen 23 bis 25 deklariert. Nachdem das Objekt vollständig konstruiert ist, ruft Zeile 48 die Methode `Invariants()` auf, um die geeignete Konstruktion zu bestätigen.

Dieses Schema wiederholt sich bei den anderen Konstruktoren und der Destruktor ruft `Invariants()` nur auf, bevor er sich an den Abbau des Objekts heranmacht. Die restlichen Klassenfunktionen rufen `Invariants()` einmal vor Ausführung einer Aktion und erneut vor der Rückkehr auf. Das bestätigt und validiert ein grundlegendes Prinzip von C++: Elementfunktionen (außer Konstruktoren und Destruktoren) sollten auf gültigen Objekten arbeiten und sie in einem gültigen Zustand belassen.

Die Klasse `Animal` deklariert in Zeile 166 ihre eigene `Invariants()`-Methode und implementiert sie in den Zeilen 179 bis 185. Beachten Sie in den Zeilen 149, 152, 155, 157, 159, 162 und 164, dass Inline-Funktionen die `Invariants()`-Methode aufrufen können.

Zwischenwerte ausgeben

Neben der Prüfung von Bedingungen mit dem Makro `assert()` möchte man bei Bedarf auch die aktuellen Werte von Zeigern, Variablen und Strings ausgeben. Das kann sehr hilfreich sein, wenn man Annahmen zum Verlauf des Programms überprüft oder in Schleifen nach Fehlern sucht, die mit Abweichungen von plus/minus 1 bei Zählerwerten zusammenhängen. Listing 21.5 verdeutlicht dieses Konzept.

Listing 21.5: printing-values.cpp – Ausgabe von Werten im DEBUG-Modus

```
 1: // Listing 21.5 - Werte im DEBUG-Modus ausgeben
 2: #include <iostream>
 3: #define DEBUG
 4:
 5: #ifndef DEBUG
 6: #define PRINT(x)
 7: #else
 8: #define PRINT(x) \
 9:     std::cout << #x << ":\t" << x << std::endl;
10: #endif
11:
12: int main()
13: {
14:     int x = 5;
15:     long y = 738981;
16:     PRINT(x);
17:     for (int i = 0; i < x; i++)
18:     {
19:         PRINT(i);
20:     }
21:
22:     PRINT (y);
23:     PRINT("Hi.");
24:     int *px = &x;
25:     PRINT(px);
```

```
26:     PRINT (*px);
27:     return 0;
28: }
```

```
x:              5
i:              0
i:              1
i:              2
i:              3
i:              4
y:              73898
"Hi.":          Hi.
px:             1245064
*px:            5
```

Bei `px:` kann bei Ihnen ein anderer Wert erscheinen (abhängig vom konkreten Computer bzw. vom verwendeten Compiler).

Das Makro in den Zeilen 5 bis 10 schreibt den aktuellen Wert des übergebenen Parameters auf den Bildschirm. Als Erstes erhält `cout` die mit Anführungszeichen versehene Version des Parameters. Wenn man also `x` übergibt, empfängt `cout` die Zeichenfolge `"x"`.

Als Nächstes erhält `cout` den mit Anführungszeichen versehenen String `":\t"`, der die Ausgabe eines Doppelpunktes mit einem anschließenden Tabulator bewirkt. Als Drittes wird an `cout` der Wert des Parameters (`x`) und schließlich `endl`, das eine neue Zeile ausgibt und den Puffer leert, übergeben.

Ebenen der Fehlersuche

In großen, komplexen Projekten möchte man mehr Einfluss auf das Programm nehmen können, als lediglich `DEBUG` ein- und auszuschalten. Zu diesem Zweck definiert man verschiedene Ebenen der Fehlersuche. Im Programm testet man dann diese Ebenen, um zu entscheiden, welche Makros zu verwenden sind und welche unbeachtet bleiben.

Um eine Ebene zu definieren, schreiben Sie einfach nach der Anweisung `#define DEBUG` die entsprechende Zahl. Obwohl man eine beliebige Anzahl von Ebenen einführen kann, haben sich im Allgemeinen vier Ebenen durchgesetzt: `HIGH`, `MEDIUM`, `LOW` und `NONE` (hoch, mittel, niedrig und keine). Listing 21.6 zeigt ein Realisierungsbeispiel anhand der Klassen `String` und `Animal` aus Listing 21.4. Außer bei `Invariants()` wurden die Definitionen der Klassenmethoden aus Platzgründen weggelassen. Diese Definitionen weisen gegenüber Listing 21.4 keine Änderungen auf.

Listing 21.6: debugging-levels.cpp – Ebenen für die Fehlersuche

```
 1: // Listing 21.6 Debugging-Ebenen
 2: #include <iostream>
 3: #include <string.h>
 4:
 5: enum LEVEL { NONE, LOW, MEDIUM, HIGH };
 6:
 7: #define DEBUGLEVEL HIGH
 8:
 9: #if DEBUGLEVEL < LOW  // Muss MEDIUM oder HIGH sein
10: #define ASSERT(x)
11: #else
12: #define ASSERT(x) \
13:     if (! (x)) \
14:     { \
15:         std::cout << "FEHLER!! Annahme " << #x << " nicht
zutreffend\n"; \
16:         std::cout << " in Zeile " << __LINE__  << "\n"; \
17:         std::cout << " in Datei " << __FILE__ << "\n";  \
18:     }
19: #endif
20:
21: #if DEBUGLEVEL < MEDIUM
22: #define EVAL(x)
23: #else
24: #define EVAL(x) \
25:     std::cout << #x << ":\t" << x << std::endl;
26: #endif
27:
28: #if DEBUGLEVEL < HIGH
29: #define PRINT(x)
30: #else
31: #define PRINT(x) \
32:     std::cout << x << std::endl;
33: #endif
34:
35: class String
36: {
37: public:
38:     // Konstruktoren
39:     String();
40:     String(const char *const);
41:     String(const String &);
42:     ~String();
43:
44:     char & operator[](int offset);
45:     char operator[](int offset) const;
46:
```

```
47:     String & operator= (const String &);
48:     int GetLen()const { return itsLen; }
49:     const char * GetString() const
50:         { return itsString; }
51:     bool Invariants() const;
52:
53: private:
54:     String (int);         // Privater Konstruktor
55:     char * itsString;
56:     unsigned short itsLen;
57: };
58:
59: bool String::Invariants() const
60: {
61:     PRINT("(String-Invariants ueberprueft)");
62:     return ( (bool) (itsLen && itsString) ||
63:         (!itsLen && !itsString) );
64: }
65:
66: class Animal
67: {
68: public:
69:     Animal():itsAge(1),itsName("John Q. Animal")
70:     {ASSERT(Invariants());}
71:
72:     Animal(int, const String&);
73:     ~Animal(){}
74:
75:     int GetAge()
76:     {
77:         ASSERT(Invariants());
78:         return itsAge;
79:     }
80:
81:     void SetAge(int Age)
82:     {
83:         ASSERT(Invariants());
84:         itsAge = Age;
85:         ASSERT(Invariants());
86:     }
87:     String& GetName()
88:     {
89:         ASSERT(Invariants());
90:         return itsName;
91:     }
92:
```

```
 93:     void SetName(const String& name)
 94:     {
 95:         ASSERT(Invariants());
 96:         itsName = name;
 97:         ASSERT(Invariants());
 98:     }
 99:
100:     bool Invariants();
101: private:
102:     int itsAge;
103:     String itsName;
104: };
105:
106: // Standardkonstruktor erzeugt String von 0 Bytes Länge
107: String::String()
108: {
109:     itsString = new char[1];
110:     itsString[0] = '\0';
111:     itsLen=0;
112:     ASSERT(Invariants());
113: }
114:
115: // Privater (Hilfs-) Konstruktor, nur von Klassen-
116: // methoden für das Erzeugen eines neuen Strings der
117: // erforderlichen Größe verwendet.  Mit Null gefüllt.
118: String::String(int len)
119: {
120:     itsString = new char[len+1];
121:     for (int i = 0; i<=len; i++)
122:         itsString[i] = '\0';
123:     itsLen=len;
124:     ASSERT(Invariants());
125: }
126:
127: // Konvertiert ein Zeichen-Array in einen String
128: String::String(const char * const cString)
129: {
130:     itsLen = strlen(cString);
131:     itsString = new char[itsLen+1];
132:     for (int i = 0; i<itsLen; i++)
133:         itsString[i] = cString[i];
134:     itsString[itsLen]='\0';
135:     ASSERT(Invariants());
136: }
137:
```

```
138: // Kopierkonstruktor
139: String::String (const String & rhs)
140: {
141:     itsLen=rhs.GetLen();
142:     itsString = new char[itsLen+1];
143:     for (int i = 0; i<itsLen;i++)
144:         itsString[i] = rhs[i];
145:     itsString[itsLen] = '\0';
146:     ASSERT(Invariants());
147: }
148:
149: // Destruktor, gibt reservierten Speicher frei
150: String::~String ()
151: {
152:     ASSERT(Invariants());
153:     delete [] itsString;
154:     itsLen = 0;
155: }
156:
157: // Gleich-Operator, gibt vorhandenen Speicher frei,
158: // kopiert dann String und Größe
159: String& String::operator=(const String & rhs)
160: {
161:     ASSERT(Invariants());
162:     if (this == &rhs)
163:         return *this;
164:     delete [] itsString;
165:     itsLen=rhs.GetLen();
166:     itsString = new char[itsLen+1];
167:     for (int i = 0; i<itsLen;i++)
168:         itsString[i] = rhs[i];
169:     itsString[itsLen] = '\0';
170:     ASSERT(Invariants());
171:     return *this;
172: }
173:
174: // Nicht-konstanter Offset-Operator, gibt Referenz
175: // auf Zeichen zurück, das sich damit ändern
176: // lässt!
177: char & String::operator[](int offset)
178: {
179:     ASSERT(Invariants());
180:     if (offset > itsLen)
181:         return itsString[itsLen-1];
182:     else
183:         return itsString[offset];
184:     ASSERT(Invariants());
185: }
```

```
186:
187: // Konstanter Offset-Operator für konstante
188: // Objekte (siehe Kopierkonstruktor!)
189: char String::operator[](int offset) const
190: {
191:     ASSERT(Invariants());
192:     if (offset > itsLen)
193:         return itsString[itsLen-1];
194:     else
195:         return itsString[offset];
196:     ASSERT(Invariants());
197: }
198:
199: Animal::Animal(int age, const String& name):
200: itsAge(age),
201: itsName(name)
202: {
203:     ASSERT(Invariants());
204: }
205:
206: bool Animal::Invariants()
207: {
208:     PRINT("(Animal-Invariants ueberprueft)");
209:     return (itsAge > 0 && itsName.GetLen());
210: }
211:
212: int main()
213: {
214:     const int AGE = 5;
215:     EVAL(AGE);
216:     Animal sparky(AGE,"Sparky");
217:     std::cout << "\n" << sparky.GetName().GetString();
218:     std::cout << " ist ";
219:     std::cout << sparky.GetAge() << " Jahre alt.";
220:     sparky.SetAge(8);
221:     std::cout << "\n" << sparky.GetName().GetString();
222:     std::cout << " ist ";
223:     std::cout << sparky.GetAge() << " Jahre alt.";
224:     return 0;
225: }
```

```
AGE:     5
(String-Invariants ueberprueft)
(String-Invariants ueberprueft)
(String-Invariants ueberprueft)
(String-Invariants ueberprueft)
(String-Invariants ueberprueft)
(String-Invariants ueberprueft)
(String-Invariants ueberprueft)
(String-Invariants ueberprueft)
(Animal-Invariants ueberprueft)
(String-Invariants ueberprueft)
(Animal-Invariants ueberprueft)

Sparky ist (Animal-Invariants ueberprueft)
5 Jahre alt. (Animal-Invariants ueberprueft)
(Animal-Invariants ueberprueft)
(Animal-Invariants ueberprueft)

Sparky ist (Animal-Invariants ueberprueft)
8 Jahre alt. (String-Invariants ueberprueft)

// Erneuter Durchlauf mit DEBUG = MEDIUM

AGE:     5
Sparky ist 5 Jahre alt.
Sparky ist 8 Jahre alt.
```

Die Definition des Makros `ASSERT()` in den Zeilen 9 bis 19 ist so ausgelegt, dass das Makro bei `DEBUGLEVEL` kleiner als `LOW` (das heißt, `DEBUGLEVEL` hat den Wert `NONE`) nicht zum Zuge kommt. Ist irgendeine Ebene für die Fehlersuche aktiviert, nimmt das Makro seine Arbeit auf. Die Definition von `EVAL` in den Zeilen 21 bis 26 ist so konzipiert, dass dieses Makro bei `DEBUGLEVEL` kleiner als `MEDIUM` übergangen wird. Das ist der Fall, wenn `DEBUGLEVEL` die Werte `NONE` oder `LOW` aufweist.

Schließlich deklarieren die Zeilen 28 bis 33 das Makro `PRINT`. Es wird übergangen, wenn `DEBUGLEVEL` kleiner als `HIGH` ist. Das Makro `PRINT` kommt nur zum Einsatz, wenn `DEBUGLEVEL` gleich `HIGH` ist, und man kann dieses Makro ausblenden, indem man `DEBUGLEVEL` auf `MEDIUM` setzt. In diesem Fall bleiben die Makros `EVAL` und `ASSERT()` aktiv.

Das Makro `PRINT` kommt innerhalb der `Invariants()`-Methoden zum Einsatz, um informative Meldungen auszugeben. In Zeile 215 wertet das Programm mit dem Makro `EVAL` den aktuellen Wert der Integer-Konstanten `AGE` aus.

Was Sie tun sollten

Schreiben Sie Makronamen durchgängig in Großbuchstaben. Da sich diese Konvention durchgesetzt hat, würde es andere Programmierer nur verwirren, wenn Sie andere Schreibweisen wählen.

Schreiben Sie Klammern um alle Argumente von Makros.

Achtung

Lassen Sie in Ihren Makros keine Seiteneffekte zu. Verzichten Sie darauf, aus einem Makro heraus Variablen zu inkrementieren oder ihnen Werte zuzuweisen.

21.8 Fragen und Antworten

F: Wenn C++ bessere Alternativen zum Präprozessor bietet, warum gibt es dann diese Option noch?

A: Erstens ist C++ abwärtskompatibel zu C und muss deshalb alle wesentlichen Elemente von C unterstützen. Zweitens gibt es einige Einsatzfälle des Präprozessors, auf die man in C++ weiterhin häufig zurückgreift. Dazu gehören zum Beispiel Schutzmaßnahmen gegen Mehrfachdeklarationen.

F: Warum setzt man Makros ein, wenn man eine normale Funktion verwenden kann?

A: Makros werden inline erweitert und dienen der Ersetzung wiederkehrender Eingaben der gleichen Befehle mit kleineren Variationen. Dennoch bieten Templates eine bessere Alternative.

F: Welche Alternativen zum Präprozessor gibt es, um Zwischenwerte während der Fehlersuche auszugeben?

A: Am besten eignen sich `watch`-Anweisungen innerhalb eines Debuggers. Informationen zu `watch`-Anweisungen finden Sie in der Dokumentation zum Compiler bzw. zum Debugger.

21.9 Workshop

Nachdem Sie sich mit dem Präprozessor beschäftigt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu festigen.

21.9.1 Kontrollfragen

1. Wie können Sie Makros eliminieren?
2. Wann werden Makros aufgelöst oder gesetzt?
3. Wie konvertiert man einen Makroparameter in einen String?
4. Welchen Operator verwendet man in Markos zur Verkettung?

21.9.2 Übungen

1. Modifizieren Sie Listing 20.4 (`employeemain.cpp`) von Stunde 20, um `string.hpp` zweimal einzubinden (duplizieren Sie die erste Zeile). Was passiert, wenn Sie das Programm kompilieren?
2. Implementieren Sie einen Schutz gegen Mehrfachdeklaration in `string.hpp` (Listing 20.3) von Stunde 20. Kompilieren Sie `employeemain.cpp` mit der Änderung nach Übung 1 erneut. Was passiert jetzt?
3. Stellen Sie für Ihren Compiler fest, wie sich eine Makrovariable von der Befehlszeile festlegen lässt. Schreiben Sie Code, um den Wert auszugeben, auf den Sie das Makro gesetzt haben.

21.9.3 Antworten zu den Kontrollfragen

1. Mit der Anweisung `#undef` lässt sich die Definition von Makros, die in anderem Code oder auf der Befehlszeile des Compilers definiert sind, aufheben.
2. Die Makro-*Erweiterung* (Modifikation des Quellcodes) findet zur Kompilierzeit statt, sodass zu diesem Zeitpunkt alle Variablen gesetzt sind und der Code geändert wird. Nach der Kompilierung lassen sich die Werte in Makros nicht mehr ändern, weil das der Code unmittelbar vor dem Kompilieren ist, der geändert wird.
3. Mit dem Operator `#` zur Zeichenkettenbildung. Er setzt Anführungszeichen um das angegebene Argument. Wenn Sie Anführungszeichen um den Parameter selbst innerhalb des Makros setzen, wird er wie eine Konstante an sich (`"x"`) und nicht als der Inhalt von `x` behandelt.
4. In Makros verwendet man den Verkettungsoperator, der aus zwei Nummernzeichen (`##`) besteht.

KAPITEL 22

jetzt lerne ich

Objektorientierte Analyse und objektorientierter Entwurf

In dieser Stunde lernen Sie,

- wie man Probleme aus objektorientierter Sicht analysiert,
- wie man Programme objektorientiert entwirft,
- wie der Entwicklungszyklus aussieht und wie man die Entwurfsziele mit der Syntax von C++ umsetzt,
- wie man einen wiederverwendbaren und erweiterbaren Entwurf realisiert.

22.1 Der Entwicklungszyklus

Viele dicke Bücher wurden schon über den Entwicklungszyklus verfasst. Manche schlagen eine Methode vor, bei der der Entwickler bestimmt, was das Programm tun sollte. Die Architekten legen den Aufbau des Programms und die zum Einsatz kommenden Klassen fest. Dann implementieren Programmierer den Entwurf und die Architektur. Wenn der Programmierer den Entwurf und die Architektur erhält, sind diese Elemente bereits fertig gestellt. Er muss dann lediglich noch die geforderte Funktionalität implementieren.

Selbst wenn diese Methode funktioniert, eignet sie sich kaum für das Schreiben guter Programme. Die eigentliche Programmierung ist mit einer ständigen Rückkopplung verbunden, um die bereits geschriebenen Teile mit den noch zu schreibenden Teilen abzugleichen. Es stimmt zwar, dass gute C++-Programme zum größten Teil bereits im Detailentwurf vorliegen, bevor man

überhaupt eine Codezeile schreibt, trotzdem bleibt der Entwurf nicht über den gesamten Entwicklungszyklus unverändert.

Der Anteil des bereits vor der Programmierung feststehenden Entwurfs ist eine Funktion der Programmgröße. Im Vergleich zu einem schnell niedergeschriebenen Utility, das ein einzelner Programmierer zwischen Frühstück und Mittag verfasst, erfordert ein hochkomplexes System, an dem Dutzende von Programmierern mehrere Monate arbeiten, eine vollständig ausgebaute Architektur.

Dieses Kapitel konzentriert sich auf den Entwurf von großen, komplexen Programmen, die über mehrere Jahre hinweg Erweiterungen und Verbesserungen erfahren. Viele Programmierer sind auf neueste Programmiertechnologien geradezu versessen und schöpfen in ihren Programmen alle Möglichkeiten ihrer Werkzeuge aus. Die Entwicklung von C++ zielt in vielerlei Hinsicht darauf ab, dem Programmierer – ob allein oder im Team – die Bewältigung komplexer Aufgaben zu erleichtern.

22.2 Ein Alarmsystem simulieren

Eine *Simulation* ist ein Computer-Modell für einen Teil eines realen Systems. Für das Erstellen einer Simulation gibt es viele Gründe. Ein guter Entwurf muss aber zunächst klären, welche Fragen die Simulation beantworten soll.

Als Ausgangspunkt untersuchen wir folgendes Problem: Wir haben den Auftrag erhalten, das Alarmsystem für ein Haus zu simulieren. Es handelt sich um ein Haus mit zentralem Aufenthaltsraum, vier Schlafzimmern, einem ausgebauten Keller und einer Tiefgarage.

Das Erdgeschoss hat folgende Fenster: drei in der Küche, vier im Esszimmer, eines im Gästebad, je zwei im Wohnzimmer und im Familienzimmer und zwei kleine neben der Tür. Alle vier Schlafzimmer befinden sich im Dachgeschoss, wobei jedes zwei Fenster hat, mit Ausnahme des Hauptschlafzimmers, das mit vier Fenstern ausgestattet ist. Es gibt zwei Bäder, jedes mit einem Fenster. Schließlich hat der Keller vier Lichtschächte und die Garage ein Fenster.

Normalerweise gelangt man über die Vordertür ins Haus. Zusätzlich hat die Küche eine Schiebetür aus Glas. Die Garage ist mit zwei Toren für die Autos und einer Verbindungstür zum Keller ausgestattet. Eine Kellertür führt zum Hof.

Alle Fenster und Türen sind mit Alarmeinrichtungen versehen. Es gibt eine Notruftaste auf jedem Telefon und eine neben dem Bett im Hauptschlafzimmer. Die Fußböden sind ebenfalls mit Alarmmeldern ausgestattet, wenn diese auch so eingestellt sind, dass sie nicht von kleinen Haustieren oder Vögeln ausgelöst werden können.

Im Kellergeschoss ist ein zentrales Alarmsystem installiert, das im Alarmfall eine Sirene ertönen lässt. Wenn man den Alarm nicht innerhalb einer bestimmten Zeitspanne deaktiviert, wird automatisch die Polizei gerufen. Mit dem Drücken der Notruftaste alarmiert man sofort die Polizei.

Außerdem ist das Alarmsystem mit Sensoren für Rauch und Feuer verbunden und mit einer Sprinkleranlage ausgestattet. Das System selbst ist fehlertolerant, verfügt über zwei interne Notstromversorgungen und ist in ein feuersicheres Gehäuse eingebaut.

22.2.1 Konzeptionelle Phase

In der *konzeptionellen Phase* versucht man zu klären, was der Kunde vom Programm erwartet: Wozu dient das Programm? Welche Ergebnisse kann die Simulation liefern? Beispielsweise könnte man anhand der Simulation die Fragen beantworten »Wie lange darf ein Sensor unterbrochen werden, bevor es jemand bemerkt?« oder »Gibt es eine Möglichkeit, den Fensteralarm auszuschalten, ohne dass die Polizei gerufen wird?«

In der konzeptionellen Phase kann man sich Gedanken darüber machen, was innerhalb und was außerhalb des Programms zu realisieren ist. Gehören die Abläufe bei der Alarmierung der Polizei zur Simulation? Ist die Steuerung des eigentlichen Hausalarms im System selbst enthalten?

22.2.2 Analyse und Anforderungen

Die konzeptionelle Phase geht in die Analysephase über. Während der Analyse ist es Ihre Aufgabe als objektorientierter Analytiker, dem Kunden die vorgesehenen Programmfunktionen auseinander zu setzen. Welches Verhalten wird das Programm an den Tag legen? Welche Arten der Interaktion kann der Kunde erwarten?

Diese Anforderungen fasst man normalerweise in einer Reihe von Dokumenten zusammen. Dazu gehören auch *Anwendungsfälle*. Ein Anwendungsfall beschreibt den Einsatz des Systems, die Interaktionen und Anwendungsschemata. Anhand der Anwendungsfälle kann der Programmierer auch die Entwurfsziele formulieren.

Unified Modeling Language (UML)

Die Unified Modeling Language (UML) ist ein Weg, Ihre Anforderungen und die Analyse darzustellen. UML bietet vor allem zwei Vorteile: Sie ist grafisch orientiert (leicht erfassbar) und folgt einer standardisierten Form (sodass sie allgemein verständlich ist). Auch wenn es über den Rahmen dieses Buches hinausgeht, UML zu vermitteln (allein zu diesem Thema gibt es viele gute Bücher), sollten Sie wissen, dass UML Möglichkeiten beinhaltet, Anwendungsfälle darzustellen. Insbesondere ist UML für die objektorientierte Entwicklung prädestiniert, da sie abstrakte und andere Klassentypen direkt unterstützt.

Empfohlen sei zu diesem Thema das Buch *Jetzt lerne ich UML* von Josef Schmuller (Markt+Technik, 2003, ISBN 3-8272-6591-6).

22.2.3 High-Level- und Low-Level-Entwurf

Nachdem man das Produkt vollständig erfasst hat und die Anforderungen in der entsprechenden Dokumentation niedergelegt sind, ist es an der Zeit, sich dem High-Level-Entwurf zu widmen. In dieser Entwurfsphase braucht sich der Programmierer nicht um die Plattform, das Betriebssystem oder die Programmiersprache zu kümmern. Er konzentriert sich nur darauf, wie das System arbeiten wird: Was sind die Hauptkomponenten? Wie arbeiten sie miteinander?

Um sich diesem Problem zu nähern, lässt man alle Themen außen vor, die sich mit der Benutzeroberfläche beschäftigen, und konzentriert sich ausschließlich auf die Komponenten im Problemraum.

Zum *Problemraum* gehören die Probleme und Fragen, die ein Programm zu lösen versucht. Der *Lösungsraum* umfasst die möglichen Lösungen für die Probleme.

Im Zuge des High-Level-Entwurfs legt man die Zuständigkeiten der bezeichneten Objekte fest – was sie bewirken und welche Informationen sie aufnehmen. Damit verbunden ist auch die Art ihrer Verknüpfung – d.h. welche Objekte mit welchen in Wechselwirkung treten.

Zunächst einmal sind natürlich Sensoren der verschiedensten Typen, ein zentrales Alarmsystem, Tasten, Kabel und Telefone vorhanden. Eine genauere Betrachtung zeigt, dass die Räume, die Fußböden und möglicherweise auch Personengruppen wie etwa die Bewohner und die Polizei zu simulieren sind.

Die Sensoren lassen sich in Bewegungsmelder, Stolperdrähte, Geräuschmelder, Rauchmelder usw. unterteilen. Das sind alles Typen von Sensoren – den Sensor *an sich* gibt es nicht. Hier deutet bereits alles darauf hin, dass man einen Sensor als abstrakten Datentyp (ADT) einrichtet.

Als ADT stellt die Klasse `Sensor` die komplette Schnittstelle für alle Arten von Sensoren bereit, während alle abgeleiteten Typen die Implementierung liefern. Klienten der verschiedenartigen Sensoren nutzen diese Schnittstelle unabhängig von der konkreten Art des Sensors und werden auf der Basis ihres realen Typs »das Richtige tun«.

Um einen guten ADT zu erzeugen, muss man die Wirkung (nicht die innere Funktionsweise) der Sensoren vollständig erfassen. Sind Sensoren zum Beispiel passive oder aktive Geräte? Warten sie darauf, ob sich ein Element erwärmt, ein Draht bricht oder eine Dichtung schmilzt oder sondieren sie ihre Umgebung? Manche Sensoren kennen nur zwei Zustände (Alarm und OK), während andere Sensoren mehrere analoge Zustände (etwa bei Temperaturmessungen) einnehmen können. Die Schnittstelle zum abstrakten Datentyp sollte umfassend genug sein, um bereits im Vorfeld alle möglichen Anforderungen für das gesamte Spektrum abgeleiteter Klassen berücksichtigen zu können.

22.2.4 Andere Objekte

Der Entwurf setzt sich nun in dieser Weise fort. Man arbeitet die verschiedenartigen Klassen heraus, die gemäß der Spezifikation erforderlich sind. Wenn beispielsweise ein Protokoll (`Log`) zu führen ist, benötigt man einen Timer. In diesem Zusammenhang taucht die Frage auf: Muss der Timer jeden Sensor abfragen oder meldet jeder Sensor in regelmäßigen Abständen seinen Zustand?

Der Benutzer muss in der Lage sein, das System einzurichten, außer Kraft zu setzen und zu programmieren. Daher ist in irgendeiner Form ein Terminal erforderlich. In der Simulation kann man auch ein separates Objekt für das Alarmprogramm selbst vorsehen.

22.2.5 Wie sehen die Klassen aus?

Im Zuge der Problemlösung beginnt man mit dem Entwurf der Klassen. Beispielsweise hat man schon einen Anhaltspunkt, dass sich `HeatSensor` (Wärmesensor) von `Sensor` ableitet. Soll der Sensor seinen Zustand periodisch melden, kann man ihn auch via Mehrfachvererbung von `Timer` ableiten oder ihm einen Timer als Variable mitgeben.

Der Wärmesensor wird wahrscheinlich über Elementfunktionen wie `CurrentTemp()` (aktuelle Temperatur) und `SetTempLimit()` (Temperaturschwelle festlegen) verfügen und Funktionen wie `SoundAlarm()` von seiner Basisklasse `Sensor` erben.

Ein häufiges Thema im objektorientierten Entwurf ist die Kapselung. Man kann sich einen Entwurf vorstellen, in dem das Alarmsystem eine Einstellung

für MaxTemp hat. Das Alarmsystem fragt vom Wärmesensor die aktuelle Temperatur ab, vergleicht sie mit der Maximaltemperatur und löst bei Überschreitung der Temperaturschwelle den Alarm aus. Man könnte vermuten, dass dies gegen das Prinzip der Kapselung verstößt. Vielleicht wäre es besser, wenn das Alarmsystem die Details der Temperaturanalyse nicht kennt oder sich nicht darum kümmern muss. Sollte man also diese Abläufe im HeatSensor realisieren?

Es sind genau diese Entscheidungen, auf die Sie sich während der Problemanalyse konzentrieren müssen. Im speziellen Fall könnte man davon ausgehen, dass nur der Sensor und das Log-Objekt alle Einzelheiten kennen, wie die Aktivitäten des Sensors aufzuzeichnen sind. Das Alarm-Objekt braucht davon nichts zu wissen und muss sich auch nicht darum kümmern.

Gute Kapselung zeichnet sich dadurch aus, dass jede Klasse über einen in sich geschlossenen Satz von Zuständigkeiten verfügt und keine andere Klasse die gleichen Aufgaben realisiert. Wenn der Sensor für die Aufzeichnung der aktuellen Temperatur verantwortlich ist, sollte diese Aufgabe nicht in den Zuständigkeitsbereich einer anderen Klasse fallen.

Auf der anderen Seite können weitere Klassen als Zulieferer der erforderlichen Funktionalität fungieren. Beispielsweise ist die Klasse Sensor für die Erfassung und Aufzeichnung der aktuellen Temperatur zuständig. Diese Verantwortlichkeit kann die Klasse dadurch implementieren, dass sie die eigentliche Aufzeichnung der Daten an ein Log-Objekt delegiert.

Durch eine strenge »Gewaltenteilung« lässt sich ein Programm leichter erweitern und warten. Will man das Alarmsystem gegen ein verbessertes Modul austauschen, ist dessen Schnittstelle zum Protokoll (Log-Objekt) und den Sensoren bereits in engen Grenzen vorgegeben. Änderungen am Alarmsystem sollten nicht auf die Sensor-Klassen zurückwirken und umgekehrt.

Sollte der HeatSensor über eine Funktion zur Alarmierung (ReportAlarm()) verfügen? Da alle Sensoren in der Lage sein müssen, einen Alarm zu melden, deutet das daraufhin, dass ReportAlarm() eine virtuelle Methode von Sensor und Sensor eine abstrakte Basisklasse sein könnte. HeatSensor kann auch mit der allgemeineren Methode ReportAlarm() der Basisklasse Sensor verkettet sein. Die redefinierte Funktion in HeatSensor ergänzt dann die Funktionalität, für deren Bereitstellung diese Klasse im Speziellen vorgesehen ist.

22.2.6 Wie wird ein Alarm gemeldet?

Wenn die Sensoren einen Alarmzustand melden, sollen sie gleichzeitig dem Objekt, das die Polizei anruft, und dem Protokoll-Objekt (Log) eine Reihe von Zusatzinformationen übermitteln. Für diesen Zweck lässt sich eine Zustandsklasse Condition erzeugen, deren Konstruktor eine Reihe von Messungen

übernimmt. Je nach Komplexität der Messungen können das ebenfalls Objekte oder einfache Skalarwerte (beispielsweise vom Typ Integer) sein.

`Condition`-Objekte kann man einerseits an das zentrale `Alarm`-Objekt übergeben oder andererseits als Unterklassen von `Alarm`-Objekten ausbilden. Bei der zweiten Variante wissen die `Alarm`-Objekte selbst, wie in Notfällen zu handeln ist. Eventuell ist gar kein zentrales Objekt vorgesehen. Stattdessen sind Sensoren denkbar, die `Condition`-Objekte erzeugen können. Bestimmte `Condition`-Objekte sind in der Lage, sich selbst zu protokollieren, andere können die Polizei herbeirufen.

In einem gut konzipierten, ereignisgesteuerten System ist kein zentraler Koordinator erforderlich. Man kann sich vorstellen, dass alle Sensoren sowohl untereinander als auch unabhängig voneinander Meldungen empfangen und senden, Parameter einrichten, Lesevorgänge unternehmen und das Haus überwachen. Bei einem Störfall wird ein `Alarm`-Objekt erzeugt. Dieses zeichnet das Problem auf (durch Senden einer Meldung an das `Log`-Objekt?) und löst die vorgesehene Aktion aus.

22.2.7 Ereignisschleifen

Um ein derartiges ereignisgesteuertes System zu simulieren, muss das Programm eine Ereignisschleife erzeugen. Dabei handelt es sich normalerweise um eine Endlosschleife wie `while(1)`, die Nachrichten vom Betriebssystem erhält (zum Beispiel Mausklicks und Tastenbetätigungen), sie nacheinander verteilt und erst bei einer erfüllten Endebedingung aus der Schleife zurückkehrt. Listing 22.1 zeigt eine einfache Ereignisschleife.

Listing 22.1: simple-event.cpp – Eine einfache Ereignisschleife

```
// Listing 22.1
#include <iostream>

class Condition
{
public:
    Condition() { }
    virtual ~Condition() {}
    virtual void Log() = 0;
};

class Normal : public Condition
{
public:
    Normal() { Log(); }
    virtual ~Normal() {}
    virtual void Log()
        { std::cout << "Normalzustand aufzeichnen...\n"; }
};
```

```
 20:
 21: class Error : public Condition
 22: {
 23: public:
 24:     Error() {Log();}
 25:     virtual ~Error() {}
 26:     virtual void Log() { std::cout << "Fehler protokollieren!\n"; }
 27: };
 28:
 29: class Alarm : public Condition
 30: {
 31: public:
 32:     Alarm ();
 33:     virtual   ~Alarm() {}
 34:     virtual void Warn() { std::cout << "Warnung!\n"; }
 35:     virtual void Log() { std::cout << "Allgemeines
Alarmprotokoll\n"; }
 36:     virtual void Call() = 0;
 37: };
 38:
 39: Alarm::Alarm()
 40: {
 41:     Log();
 42:    Warn();
 43: }
 44:
 45: class FireAlarm : public Alarm
 46: {
 47: public:
 48:     FireAlarm(){Log();};
 49:     virtual ~FireAlarm() {}
 50:     virtual void Call() { std::cout<< "Feuerwehr alarmieren!\n"; }
 51:     virtual void Log() { std::cout << "Feuerwehrnotruf
aufzeichnen.\n"; }
 52: };
 53:
 54: int main()
 55: {
 56:     int input;
 57:     int okay = 1;
 58:     Condition * pCondition;
 59:     while (okay)
 60:     {
 61:         std::cout << "(0)Beenden (1)Normal (2)Feuer: ";
 62:         std::cin >> input;
 63:         okay = input;
 64:         switch (input)
 65:         {
```

```
66:         case 0:
67:             break;
68:         case 1:
69:             pCondition = new Normal;
70:             delete pCondition;
71:             break;
72:         case 2:
73:             pCondition = new FireAlarm;
74:             delete pCondition;
75:             break;
76:         default:
77:             pCondition = new Error;
78:             delete pCondition;
79:             okay = 0;
80:             break;
81:         }
82:     }
83:     return 0;
84: }
```

```
(0)Beenden (1)Normal (2)Feuer: 1
Normalzustand aufzeichnen...
(0)Beenden (1)Normal (2)Feuer: 2
Allgemeines Alarmprotokoll
Warnung!
Feuerwehrnotruf aufzeichnen.
(0)Beenden (1)Normal (2)Feuer: 0
```

Die einfache Schleife in den Zeilen 59 bis 82 erlaubt dem Benutzer, Eingaben vorzunehmen, um eine normale Meldung von einem Sensor und einen Feueralarm zu simulieren. Diese Meldung bringt ein `Condition`-Objekt hervor, dessen Konstruktor verschiedene Elementfunktionen aufruft.

Der Aufruf virtueller Elementfunktionen von einem Konstruktor kann zu verwirrenden Ergebnissen führen, wenn man nicht genau auf die Reihenfolge der Objektkonstruktion achtet. Wenn zum Beispiel das `FireAlarm`-Objekt in Zeile 73 erzeugt wird, lautet die Konstruktionsreihenfolge `Condition`, `Alarm`, `FireAlarm`. Der `Alarm`-Konstruktor ruft `Log()` auf, wobei es sich aber um `Log()` von `Alarm` und nicht von `FireAlarm` handelt, obwohl `Log()` als virtuell deklariert ist. Das hängt damit zusammen, dass bei Ausführung des `Alarm`-Konstruktors noch kein `FireAlarm`-Objekt existiert. Nachdem dann `FireAlarm` selbst konstruiert ist, ruft dessen Konstruktor erneut `Log()` auf – jetzt aber `FireAlarm::Log`.

22.3 PostMaster: Eine Fallstudie

Hier ein weiteres Problem, an dem Sie Ihre objektorientierte Analyse ausprobieren können: Die Firma Acme Software hat Sie eingestellt, um ein neues Software-Projekt zu beginnen und ein Team von C++-Programmierern zusammenzustellen, das Ihr Programm implementiert. Ihr neuer Chef ist Jim Grandiose, Leiter der Produktentwicklung. Er gibt Ihnen den Auftrag, PostMaster zu entwerfen und zu erstellen – ein Dienstprogramm, das E-Mails der verschiedensten E-Mail-Anbieter lesen soll. Als potentielle Kunden sind Geschäftsleute vorgesehen, die mit mehreren E-Mail-Produkten arbeiten, beispielsweise CompuServe, America Online, Internet Mail und Lotus Notes.

Der Kunde soll in der Lage sein, die Verbindungen zu allen E-Mail-Providern mit PostMaster herzustellen. PostMaster holt dann die E-Mail ab und präsentiert sie in einer einheitlichen Form. Somit kann der Kunde die Mail organisieren, darauf antworten, Briefe zwischen den Diensten übermitteln usw.

PostMaster Professional, das als Version 2 von PostMaster zu entwickeln ist, liegt bereits in der Planung vor. Es soll hier ein Verwaltungsmodus hinzukommen, über den der Benutzer andere Personen benennen kann, die einen Teil oder die gesamte Post lesen, Routinekorrespondenz abwickeln oder ähnliche Aufgaben ausführen. In der Marketing-Abteilung spricht man außerdem schon von einer Komponente der künstlichen Intelligenz, die die Fähigkeit des PostMasters zur Vorsortierung und Kategorisierung (Prioritäten, Dringlichkeit) der Mail auf der Basis assoziativer Verknüpfungen von Schlüsselwörtern im Betreff und im Inhalt realisiert.

Andere Erweiterungen wurden ebenfalls angesprochen. Das Programm soll nicht nur einfache Mails behandeln, sondern auch Diskussionsgruppen wie etwa Internet-Newsgroups und Mailing-Listen bedienen können. Offensichtlich verknüpft Acme große Erwartungen mit PostMaster. Damit stehen Sie unter strengem Termindruck, obwohl Sie scheinbar ein nahezu unbegrenztes Budget zur Verfügung haben.

22.3.1 Wegweisende Entscheidungen

Sie richten Ihr Büro ein und bestellen die Ausrüstung. Als erste Amtshandlung stellen Sie eine gute Spezifikation für das Produkt zusammen. Nach einer Marktanalyse schlagen Sie die Entwicklung für eine einzelne Plattform vor und stellen UNIX, den Macintosh und Windows zur Disposition.

Nach mehreren unangenehmen Treffen mit Jim Grandiose zeichnet sich ab, dass es gar keine richtige Auswahl gibt. Daher trennen Sie den Frontend – das heißt, die Benutzeroberfläche – vom Backend – den Teilen für Kommunikation und Datenbank. Aus Zeitgründen entscheiden Sie sich für Windows, worauf später UNIX und vielleicht der Mac folgen sollen.

Diese einfache Entscheidung wirkt sich auf Ihr gesamtes Projekt aus. Es wird schnell deutlich, dass Sie eine Klassenbibliothek oder eine Reihe von Bibliotheken brauchen, um die Speicherverwaltung, die verschiedenartigen Benutzeroberflächen und möglicherweise auch die Komponenten für Kommunikation und Datenbank in den Griff zu bekommen.

Herr Grandiose ist der festen Überzeugung, dass der Erfolg eines Projekts von einer starken Persönlichkeit mit einem klaren Ziel vor Augen abhängig ist. Er möchte, dass Sie die anfängliche Architekturanalyse und den Entwurf vornehmen, bevor irgendwelche Programmierer eingestellt werden. Daher machen Sie sich nun an die Analyse des Problems.

22.3.2 Teile und herrsche

Schnell wird deutlich, dass eigentlich mehrere Probleme zu lösen sind. Sie gliedern das Projekt in die folgenden Unterprojekte:

- Kommunikation: Alle Abläufe, die die Einwahl zu einem E-Mail-Provider über ein Modem oder die Verbindung über ein Netzwerk betreffen.
- Datenbank: Das Speichern und Abrufen der Daten auf Festplatte.
- E-Mail: Das Lesen verschiedener E-Mail-Formate und das Schreiben neuer Nachrichten für jedes System.
- Bearbeitung: Erstellen und Bearbeiten von Nachrichten mit modernen Editoren.
- Plattformen: Die auf den einzelnen Plattformen zu realisierenden Benutzeroberflächen.
- Erweiterbarkeit: Vorkehrungen für Vergrößerungen und Erweiterungen.
- Organisation und Planung: Koordinierung der verschiedenen Entwickler und der gegenseitigen Abhängigkeit ihrer Codeabschnitte. Jede Gruppe muss Pläne ausarbeiten und veröffentlichen, damit sich eine Gesamtplanung erstellen lässt. Leitung und Marketingabteilung müssen wissen, wann das Produkt fertig sein wird.

Sie entscheiden sich für die Einstellung eines Managers, der sich der Organisation und Planung widmet. Weiterhin holen Sie sich erfahrene Entwickler für die Analyse und den Entwurf, die später an der Implementierung der restlichen Bereiche arbeiten. Diese Entwickler bilden die folgenden Teams:

- Kommunikation: Verantwortlich für die Datenübertragung sowohl bei Einwahlverbindungen als auch bei Netzwerken. Diese Gruppe beschäftigt sich mit Paketen, Streams und Bits statt mit E-Mail-Nachrichten an sich.
- Nachrichtenformat: Verantwortlich für die Umwandlung der Nachrichten aus den Formaten der einzelnen E-Mail-Provider in eine Normalform

(PostMaster-Standard) und zurück. Die Aufgabe dieses Teams ist es auch, die Nachrichten auf Festplatte zu schreiben und sie bei Bedarf wieder abzurufen.

- Nachrichten-Editoren: Diese Gruppe ist für die gesamte Benutzeroberfläche des Produkts auf den einzelnen Plattformen verantwortlich und muss sicherstellen, dass die Schnittstelle zwischen dem Backend und dem Frontend des Produkts ausreichend scharf definiert ist, sodass die Erweiterung des Produkts für andere Plattformen keinen doppelten Code erfordert.

22.3.3 Nachrichtenformat

Als Erstes konzentrieren Sie sich auf das Nachrichtenformat und stellen die Themen hinsichtlich Kommunikation und Benutzeroberfläche hinten an. Diesen Problemen widmen Sie sich erst dann, wenn die Details klar sind. Es hat wenig Sinn, sich über die Präsentation der Informationen an den Benutzer Gedanken zu machen, solange man nicht die Informationen kennt, mit denen man arbeiten muss.

Eine Untersuchung der verschiedenen E-Mail-Formate zeigt trotz ihrer unterschiedlichen Herkunft viele Gemeinsamkeiten. Jede E-Mail-Nachricht hat einen Ursprung, ein Ziel und ein Erstellungsdatum. Nahezu alle Meldungen enthalten einen Titel oder eine Zeile mit dem Betreff sowie einen Hauptteil, der aus einfachem Text, formatiertem Text, Grafiken und vielleicht sogar Sound oder anderen netten Erweiterungen besteht. Die meisten derartigen E-Mail-Dienste unterstützen auch Anlagen, sodass die Benutzer Programme und andere Dateien versenden und empfangen können.

Sie bleiben bei Ihrer früheren Entscheidung, dass Sie jede E-Mail-Nachricht aus dem Originalformat lesen und in das PostMaster-Format schreiben. Auf diese Weise ist nur ein Aufzeichnungsformat zu speichern und das Lesen und Schreiben der Daten von der bzw. auf die Festplatte vereinfacht sich. Außerdem entscheiden Sie sich, die Kopfinformationen (Sender, Empfänger, Datum, Betreff usw.) vom Hauptteil der Nachricht zu trennen. Oftmals möchte der Benutzer die Kopfdaten durchsuchen, ohne dass er unbedingt den Inhalt aller Nachrichten lesen will. Damit nehmen Sie vorweg, dass der Benutzer irgendwann nur die Kopfinformationen des Nachrichtenanbieters herunterladen möchte, ohne den Text überhaupt abzuholen. Momentan aber sehen Sie in der Version 1 von PostMaster das Abholen der vollständigen Nachricht vor, auch wenn sie dem Benutzer eventuell nicht angezeigt wird.

22.3.4 Anfänglicher Klassenentwurf

Die Analyse der Nachrichten führt zum Entwurf der Klasse `Message`. In Vorwegnahme einer Programmerweiterung auf Nicht-E-Mail-Nachrichten leiten Sie `EmailMessage` von der abstrakten Basisklasse `Message` ab. Von `Email-`

`Message` leiten Sie wiederum `PostMasterMessage`, `InterchangeMessage`, `CISMessage`, `ProdigyMessage` usw. ab.

In einem Programm, das Nachrichten behandelt, ergibt sich die Wahl einer Nachrichten-Klasse (`Message`) fast von selbst. In einem komplexen System ist es aber eine der letzten Herausforderungen der objektorientierten Programmierung, die richtigen Objekte zu ermitteln. In manchen Fällen, beispielsweise bei Nachrichten, scheinen die Hauptobjekte förmlich aus der Problemformulierung »hervorzutreten«. Allerdings ist es häufiger so, dass Sie sich den Weg zu den richtigen Objekten hart erarbeiten müssen.

Werfen Sie die Flinte nicht gleich ins Korn. Die meisten Entwürfe sind nicht gleich von Anfang an perfekt. Ein guter Ausgangspunkt ist die Beschreibung des Projekts in Form eines Vortrags. Machen Sie sich eine Liste aller Substantive und Verben, die Sie für die Beschreibung des Projekts verwenden. Die Substantive sind gute Kandidaten für Objekte. Die Verben mutieren zu Methoden dieser Objekte (oder zu selbstständigen Objekten). Das ist zwar keine narrensichere Methode, eignet sich aber als Ausgangspunkt für einen ersten Entwurf.

Das war der leichte Teil. Nun stellt sich die Frage: »Sollte der Nachrichtenkopf eine vom Hauptteil getrennte Klasse sein?« Wenn ja, braucht man dann parallele Hierarchien – sowohl `NewsGroupBody` und `NewsGroupHeader` als auch `EmailBody` und `EmailHeader`?

Parallele Hierarchien sind oft ein Hinweis für einen schlechten Entwurf. Es ist ein allgemeiner Fehler im objektorientierten Entwurf, eine Gruppe von Objekten in der einen Hierarchie unterzubringen und einen entsprechenden Satz von Verwaltungsobjekten in einer anderen. Die Belastung durch die Aktualisierung und Synchronisierung dieser Hierarchien untereinander wird bald zum Albtraum.

Natürlich gibt es keine festen Regeln und manchmal sind parallele Hierarchien die effizienteste Art, um ein bestimmtes Problem zu lösen. Trotzdem sollten Sie Ihren Entwurf neu überdenken, wenn er in diese Richtung läuft. Möglicherweise gibt es eine elegante Lösung.

Wenn die Nachrichten vom E-Mail-Provider ankommen, liegen diese nicht unbedingt in Kopf- und Hauptteil getrennt vor. Viele werden aus einem großen Datenstrom bestehen, den Ihr Programm entwirren muss. Vielleicht sollte Ihre Hierarchie dieses Konzept direkt widerspiegeln.

Wenn man sich näher mit den praktischen Details beschäftigt, kann man die Eigenschaften der Nachrichten auflisten und dabei die vorzusehenden Fähigkeiten und die Datenspeicherung auf der richtigen Abstraktionsebene einordnen. Durch die Auflistung der Objekteigenschaften lassen sich auch die Datenelemente finden sowie andere eventuell benötigte Objekte herausarbeiten.

Neben den Mail-Nachrichten sind auch die Vorzugseinstellungen des Benutzers, Telefonnummern und ähnliche Angaben zu speichern. Es liegt auf der Hand, dass die Speicherung in den höheren Ebenen der Hierarchie angesiedelt sein muss. In diesem Zusammenhang stellt sich die Frage, ob man für Mail-Nachrichten und Vorzugseinstellungen eine gemeinsame Basisklasse vorsehen sollte.

22.3.5 Hierarchien mit und ohne Wurzel

Es gibt zwei allgemeine Ansätze für Vererbungshierarchien: Man leitet alle (oder nahezu alle) Klassen von einer gemeinsamen Wurzelklasse ab oder man baut mehrere Vererbungshierarchien auf. Ein Vorteil der gemeinsamen Wurzelklasse besteht darin, dass man in der Regel auf Mehrfachvererbung verzichten kann. Nachteilig ist, dass die Implementierung oftmals bis in die Basisklasse hinaufreicht.

Klassen sind *verwurzelt*, wenn sie alle von einem gemeinsamen Vorfahren abstammen. Bei *nicht verwurzelten* Hierarchien gehen nicht alle Klassen auf eine gemeinsame Basisklasse zurück.

Da Ihr Produkt auf mehreren Plattformen entwickelt werden soll und da nicht alle Compiler auf allen Plattformen die Mehrfachvererbung in gleichem Maße unterstützen, entscheiden Sie sich für eine verwurzelte Hierarchie und Einfachvererbung. Vorsichtshalber kennzeichnen Sie aber diejenigen Stellen, wo in Zukunft gegebenenfalls eine Mehrfachvererbung erforderlich sein könnte, und Sie gestalten den Entwurf von vornherein so, dass die Auftrennung der Hierarchie und das Hinzufügen der Mehrfachvererbung zu einem späteren Zeitpunkt nicht Ihren gesamten Entwurf infrage stellt.

Die Namen aller internen Klassen versehen Sie mit dem Präfix `p`, damit Sie ohne weiteres erkennen können, welche Klassen von Ihnen und welche aus anderen Bibliotheken stammen.

Die Wurzelklasse soll `pObject` heißen. Praktisch jede Klasse, die Sie erzeugen, wird von diesem Objekt abgeleitet. `pObject` selbst ist sehr einfach gehalten. Nur die wirklich von allen Elementen gemeinsam genutzten Daten erscheinen in dieser Klasse.

In einer verwurzelten Hierarchie erhält die Wurzelklasse in der Regel einen eher generischen Namen (wie etwa `pObject`) und nur ein Minimum an Fähigkeiten. Mit einem Wurzelobjekt verfolgt man die Absicht, Kollektionen von allen Nachfolgern zu bilden und sich auf sie als Instanzen von `pObject` zu beziehen. Bei verwurzelten Hierarchien muss man in Kauf nehmen, dass die Schnittstellen oftmals bis in die Wurzelklasse hinauf verzweigen.

Die nächsten Kandidaten für die oberste Ebene der Hierarchie sind `pStored` und `pWired`. Die `pStored`-Objekte werden zu verschiedenen Zeitpunkten

(wenn das Programm zum Beispiel nicht aktiv ist) auf Festplatte gespeichert, und `pWired`-Objekte verschickt das Programm über das Modem oder das Netzwerk. Da fast alle Ihre Objekte auf Festplatte zu speichern sind, bietet es sich an, diese Funktionalität in den höheren Hierarchieebenen anzusiedeln. Während man alle Objekte, die das Programm per Modem verschickt, speichern muss, aber nicht alle gespeicherten Objekte über die Leitung geschickt werden müssen, ist es sinnvoll, `pWired` von `pStored` abzuleiten.

Alle abgeleiteten Klassen erwerben das gesamte Wissen (die Daten) und die Funktionalität (die Methoden) ihrer Basisklasse und abgeleitete Klassen sollten die Funktionalität gegenüber der Basisklasse erweitern. Demzufolge nimmt man in `pWired` zusätzliche Methoden auf, die eine Übertragung von Daten via Modem realisieren.

Es ist durchaus möglich, dass das Programm alle `pWired`-Objekte speichert, dass es alle gespeicherten Objekte via Modem verschickt oder dass keine dieser Annahmen zutrifft. Wenn nur einige `pWired`-Objekte gespeichert werden und nur einige gespeicherte Objekte über die Leitung gehen, muss man entweder mit Mehrfachvererbung arbeiten oder das Problem umgehen. Als mögliche Lösung könnte man etwa `pWired` von `pStored` ableiten. Die Speichermethoden lässt man dann nichts ausführen oder gibt einen Fehler für diejenigen Objekte zurück, die via Modem verschickt, aber niemals gespeichert werden.

Gehen wir davon aus, dass einige gespeicherte Objekte – wie etwa die Vorzugseinstellungen des Benutzers (`pPreferences`) – mit Sicherheit nicht zu versenden sind. Dagegen werden alle über die Leitung geschickten Objekte auch gespeichert. Aus den bisherigen Annahmen ergibt sich die in Abbildung 22.1 dargestellte Vererbungshierarchie.

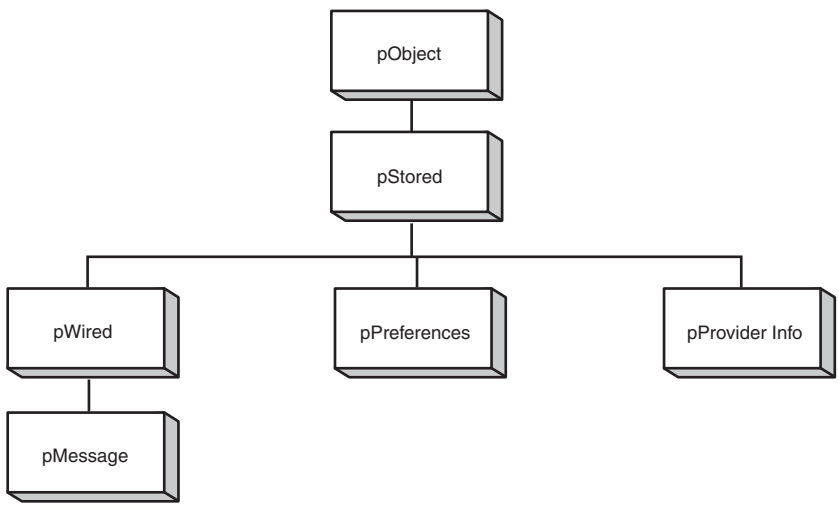


Abb. 22.1: Anfängliche Vererbungshierarchie

Kaufen oder selbst schreiben?

Diese Frage begleitet Sie während der gesamten Entwurfsphase Ihres Programms. Welche Routinen sollte man fertig kaufen und welche sollte man selbst schreiben? Es ist durchaus möglich, dass man vorhandene kommerzielle Bibliotheken nutzt, um einige oder alle Punkte der Kommunikation zu lösen. Dabei sind allerdings auch Lizenzgebühren und andere Belange außerhalb der technischen Realisierung zu berücksichtigen.

Es ist oftmals besser, eine derartige Bibliothek zu kaufen und die eigene Energie auf die spezifischen Programmteile zu konzentrieren, als das Rad neu zu erfinden. Man könnte sogar Bibliotheken kaufen, die nicht unbedingt für C++ vorgesehen sind, wenn sich damit grundlegende Funktionalität realisieren lässt, die man sonst selbst entwickeln müsste. Das Ganze kann bei der Einhaltung des Endtermins förderlich sein.

22.3.6 Die Schnittstellen entwerfen

Auf dieser Stufe des Produktentwurfs sollten Sie die Details der Implementierung außen vor lassen. Konzentrieren Sie Ihre gesamte Energie auf den Entwurf einer sauberen Schnittstelle zwischen den Klassen und skizzieren Sie dann, welche Daten und Methoden jede Klasse braucht.

Zunächst einmal sollte man sich tiefgründig mit den Basisklassen beschäftigen, bevor es an den Entwurf der abgeleiteten Klassen geht. Konzentrieren Sie sich also zunächst auf `pObject`, `pStored` und `pWired`.

Die Wurzelklasse `pObject` erhält nur diejenigen Daten und Methoden, die allen Elementen in Ihrem System gemeinsam sind. Vielleicht sollte jedes Objekt eine eindeutige Identifikationsnummer bekommen. Dazu erzeugen Sie `pID` (PostMaster-ID) und machen sie zu einem Element von `pObject`. Zuerst müssen Sie sich aber die Frage stellen: »Benötigt jedes Objekt, das nicht gespeichert und nicht übertragen wird, eine derartige Nummer?« Das wirft die Frage auf: »Gibt es Objekte, die nicht gespeichert werden, die aber Teil dieser Hierarchie sind?«

Sind keine derartigen Objekte vorhanden, empfiehlt es sich, `pObject` und `pStored` zu einer Klasse zu verschmelzen. Wenn schließlich alle Objekte gespeichert werden, wo liegt dann der Grund für eine Unterscheidung? Bei näherem Hinsehen entdecken Sie Objekte, wie zum Beispiel Adressenobjekte, bei denen eine Ableitung von `pObject` zwar sinnvoll ist, die aber nicht eigenständig gespeichert werden. Wenn man sie speichert, dann als Teil irgendeines anderen Objekts.

Momentan ist es also angebracht, eine separate `pObject`-Klasse zu haben. Denkbar ist ein Adressbuch, das eine Kollektion von `pAddress`-Objekten darstellt. Da man aber kein `pAddress`-Objekt für sich allein speichert, wäre es von

Nutzen, wenn jedes Objekt über eine eigene Identifikationsnummer verfügt. Vorläufig nehmen Sie pID in pObject auf. In der Minimalversion sieht pObject wie folgt aus:

```
class pObject
{
public:
    pObject();
    ~pObject();
    pID GetID()const;
    void SetID();
private:
    pID  itsID;
}
```

Zu dieser Klassendeklaration ist Verschiedenes anzumerken. Erstens leitet sich diese Klassendeklaration nicht von einer anderen Klasse ab – sie ist Ihre Wurzelklasse. Zweitens kümmern wir uns zunächst überhaupt nicht um die Implementierung. Auch nicht bei Methoden wie GetID(), die am Ende in einer Inline-Version vorliegen werden. Drittens sind konstante Methoden bereits gekennzeichnet. Das ist Teil der Schnittstelle und nicht der Implementierung. Schließlich wird ein neuer Datentyp impliziert: pID. Die Definition von pID als Typ statt beispielsweise als unsigned long bringt mehr Flexibilität in Ihren Entwurf.

Wenn sich nämlich herausstellt, dass man keinen unsigned long braucht oder dass ein unsigned long nicht groß genug ist, kann man pID einfach modifizieren. Diese Modifikation beeinflusst alle Stellen, wo pID vorkommt, und man muss pID nicht in allen Dateien suchen und bearbeiten.

Fürs Erste deklarieren Sie pID mithilfe von typedef als ULONG, wobei Sie ULONG wiederum als unsigned long deklarieren. Das wirft die Frage auf: Wohin gehören diese Deklarationen?

Bei einem umfangreichen Projekt benötigt man auch einen detaillierten Plan für die erforderlichen Dateien. Als Standardlösung, die auch in diesem Projekt zur Anwendung kommt, spendiert man jeder Klasse eine eigene Header-Datei und bringt die Implementierungen für die Klassenmethoden in zugehörigen .CPP-Dateien unter. Damit hat man eine Datei namens OBJECT.HPP und eine korrespondierende Datei OBJECT.CPP. Dazu kommen später weitere Dateien wie MSG.HPP und MSG.CPP für die Deklaration der Klasse pMessage bzw. für die Implementierung ihrer Methoden.

22.3.7 Einen Prototyp erstellen

Bei einem großen Projekt wie PostMaster ist es unwahrscheinlich, dass der anfängliche Entwurf vollständig und perfekt ist. Allein durch den Umfang des Problems ist man schnell überfordert. Um einer Katastrophe zu entgehen,

sollte man zuerst alle Klassen erzeugen und deren Schnittstellen fertig stellen, bevor man überhaupt eine Zeile mit arbeitsfähigem Code schreibt.

Es gibt eine Reihe guter Gründe, den Entwurf an einem Prototyp – einem schnell programmierten und arbeitsfähigen Beispiel für die Kernkonzepte – auszuprobieren. Für die jeweiligen Ansprüche existieren auch unterschiedliche Prototypen.

Ein Prototyp für den Entwurf der Schnittstellen bietet die Möglichkeit, Aussehen und Handhabung des Produkts mit potentiellen Benutzern zu testen.

Es lässt sich auch ein Prototyp für die Funktionalität entwerfen, dem zwar die endgültige Benutzeroberfläche fehlt, der aber dem Benutzer das Ausprobieren verschiedener Leistungsmerkmale erlaubt, beispielsweise das Abschicken von Meldungen oder das Anhängen von Dateien.

Schließlich lässt sich mit einem Prototyp für die Architektur eine kleinere Programmversion erstellen. Außerdem kann man abschätzen, in welchem Maße sich Entwurfsentscheidungen im fertig gestellten Programm auswirken.

Es ist zwingend erforderlich, die Ziele für den Prototyp klar zu umreißen. Untersucht man die Benutzeroberfläche, experimentiert man mit der Funktionalität oder baut man ein maßstäbliches Modell des Endprodukts? Ein guter Prototyp für die Architektur erzeugt einen schlechten Prototyp für die Benutzeroberfläche und umgekehrt.

Bei der Auslegung des Prototyps sollte man ein ausgewogenes Verhältnis zwischen Aufwand und Nutzen im Auge behalten. Wenn man zu viel Arbeit in den Prototyp investiert hat, geht man im Zuge der weiteren Programmentwicklung nur noch widerwillig daran, den Code gegebenenfalls niederzureißen und neu zu entwerfen.

22.3.8 Die 80/80-Regel

In dieser Entwicklungsphase gilt die Faustregel, dass man sich auf den Entwurf derjenigen Komponenten konzentriert, mit denen schätzungsweise 80% der potentiellen Anwender etwa 80% ihrer Zeit verbringen. Die restlichen 20% kommen später an die Reihe. Früher oder später muss man sich zwar den Randbedingungen zuwenden, der Kern des Entwurfs sollte sich aber an der 80/80-Regel orientieren.

Dementsprechend entwirft man zunächst die Hauptklassen und stellt die untergeordneten Klassen zurück. Gibt es Klassen mit ähnlichem Design, konzentriert man die Entwurfsarbeit auf einen typischen Vertreter dieser Gruppe und wendet sich den nahen Verwandten später zu.

Pareto-Regel

Eine weitere Regel – die 80/20-Regel – besagt, dass »die ersten 20% eines Programms etwa 80% der Zeit für die Codierung beanspruchen, während für die restlichen 80% dieses Programms nochmals 80% der Zeit zu veranschlagen sind!«

Die vom italienischen Volkswirtschaftler Vilfredo Pareto (1848 bis 1923) aufgestellte Regel besagt aber tatsächlich, dass schon nach 20% der veranschlagten Zeit etwa 80% der Lösung erarbeitet ist – oder dass 80% des Firmengewinns von 20% der Kunden kommen. Diese Regel ist weithin anwendbar.

22.3.9 Entwurf der Klasse PostMasterMessage

Im Einklang mit diesen Überlegungen konzentrieren Sie sich zunächst auf die Klasse `PostMasterMessage`. Es handelt sich dabei um die Klasse, die am ehesten unter Ihrer direkten Kontrolle steht.

Als Teil der Schnittstelle muss `PostMasterMessage` natürlich den Austausch mit anderen Nachrichtentypen beherrschen. Man hofft zwar auf eine enge Zusammenarbeit mit den einzelnen Nachrichtenanbietern, kann momentan aber hinsichtlich des Nachrichtenformats nur Vermutungen anstellen und einfach beobachten, was die jeweiligen Dienste über die Leitung schicken.

Auf jeden Fall ist bekannt, dass die `PostMasterMessage`-Nachrichten aus Absender, Empfänger, Datum und Betreff sowie Hauptteil der Nachricht und eventuell angehängten Dateien bestehen. Damit sind Zugriffsmethoden für diese Attribute sowie Methoden zur Meldung der Größe der angehängten Dateien, der Größe der Nachrichten usw. erforderlich.

Manche Dienste, mit denen ein Nachrichtenaustausch vorgesehen ist, verwenden formatierten Text, um Schriftart, Zeichengröße und Attribute wie fett und kursiv festzulegen. Hier ist außerdem mit proprietären Formaten zu rechnen. Andere Dienste unterstützen diese Attribute nicht. Die Klasse benötigt also Konvertierungsmethoden, um formatierten Text in reinen ASCII-Text zu überführen und gegebenenfalls andere Formate in die PostMaster-Formate umzuwandeln.

22.3.10 Die API

Eine Anwendungsprogrammierschnittstelle (API, Application Programming Interface) ist ein Satz von Dokumentationen und Routinen für die Nutzung eines Dienstes. Einige der Mail-Provider stellen Ihnen eine API bereit, sodass PostMaster-Mail auch mit formatiertem Text und eingebetteten Dateien umgehen kann. Für PostMaster werden Sie ebenfalls eine API veröffentlichen, damit in Zukunft andere Anbieter mit Ihrem Programm arbeiten können.

Für Ihre PostMasterMessage-Klasse ist eine gut konzipierte öffentliche Schnittstelle erforderlich. Die Konvertierungsfunktionen gehören dabei zu den Hauptkomponenten der PostMaster-API. Listing 22.2 veranschaulicht, wie die Schnittstelle von PostMasterMessage bislang aussieht.

Versuch zwecklos!

Listing 22.2 definiert die Basisklasse (MailMessage) noch nicht und lässt sich demnach nicht kompilieren.

Listing 22.2: postmaster-interface.cpp – Die Schnittstelle der Klasse PostMaster-Message

```
// Listing 22.2 Die Klasse PostMasterMessage

class PostMasterMessage : public MailMessage
{
public:
    PostMasterMessage();
    PostMasterMessage(
        pAddress Sender,
        pAddress Recipient,
        pString Subject,
        pDate creationDate);

    // Hier noch andere Konstruktoren vorsehen
    // Nicht vergessen: Kopierkonstruktor und
    // Konstruktor von Speicherung sowie Konstruktor
    // von Übertragungsformat.
    // Außerdem Konstruktoren von anderen Formaten.
    ~PostMasterMessage();
    pAddress& GetSender() const;
    void SetSender(pAddress&);
    // Andere Zugriffsfunktionen auf Datenelemente

    // Hier Operator-Methoden, u.a. Gleich-Operator
    // und Routinen zur Umwandlung der PostMaster-
    // Nachrichten in andere Formate.

private:
    pAddress itsSender;
    pAddress itsRecipient;
    pString  itsSubject;
    pDate itsCreationDate;
    pDate itsLastModDate;
    pDate itsReceiptDate;
    pDate itsFirstReadDate;
    pDate itsLastReadDate;
};
```

Dieses Listing produziert keine Ausgabe.

Die Deklaration der Klasse `PostMasterMessage` sieht eine Ableitung von `MailMessage` vor. Die bereitgestellten Konstruktoren erleichtern das Erzeugen von `PostMasterMessage`-Nachrichten aus anderen Typen von Mail-Nachrichten.

Bereits vorgesehen sind sowohl Zugriffsmethoden für das Lesen und Setzen der verschiedenen Datenelemente als auch Operatoren für die vollständige oder teilweise Umwandlung der Nachrichten in andere Nachrichtenformate. Es ist geplant, diese Nachrichten auf Festplatte zu speichern und sie über die Datenverbindung einzulesen, sodass Zugriffsmethoden für diese Zwecke ebenso vermerkt sind.

22.3.11 Team-Arbeit

Selbst aus dieser vorläufigen Architektur lässt sich bereits ableiten, wie die verschiedenen Entwicklungsgruppen vorgehen sollten. Die Gruppe *Kommunikation* kann mit der Arbeit am Backend für die Kommunikation beginnen und muss dabei eine eng umrissene Schnittstelle mit der Gruppe *Nachrichtenformat* aushandeln.

Die Gruppe *Nachrichtenformat* setzt die bereits weiter oben begonnene Ausarbeitung der allgemeinen Schnittstelle zu den `Message`-Klassen fort und wendet sich dann der Frage zu, wie die Daten auf Festplatte zu schreiben und von dort zu lesen sind. Nachdem alle Punkte in Bezug auf die Festplattenschnittstelle geklärt sind, kann dieses Team die Schnittstelle zur Kommunikationsschicht aushandeln.

Die Gruppe *Nachrichtenbearbeitung* könnte der Versuchung erliegen, Editoren mit genauer Kenntnis der Interna der `Message`-Klasse aufzubauen. Das wäre aber ein übler Entwurfsfehler. Diese Gruppe muss ebenfalls eine klar definierte Schnittstelle zur `Message`-Klasse aushandeln. Objekte der Nachrichtenbearbeitung sollten nur ein Mindestmaß an Informationen über die interne Struktur der Nachrichten erhalten.

22.3.12 Weiterführende Betrachtungen zum Entwurf

Im Zuge der Projektentwicklung werden Sie wiederholt mit dem folgenden grundlegenden Entwurfsproblem konfrontiert: In welcher Klasse sollte ein bestimmter Satz der Funktionalität (oder der Informationen) realisiert sein? Gehören die Funktionen in die Klasse `Message` oder vielleicht in die Klasse `Address`? Speichert der Editor diese Informationen oder übernimmt das die Nachrichtenklasse selbst?

Prinzipiell sollten Ihre Klassen nicht mehr voneinander wissen als absolut erforderlich.

Entwurfsentscheidungen

Die Programmentwicklung ist von Hunderten von Entwurfsproblemen begleitet. Diese reichen von allgemeineren Fragen (»Was soll hier passieren?«) bis zu konkreten (»Wie setzen wir das um?«).

Obwohl die Details der Implementierung erst kurz vor der Auslieferung des Produkts feststehen und sich manche Schnittstellen noch während Ihrer Arbeit ändern, müssen Sie sicherstellen, dass Ihr Entwurf bereits in einem frühen Stadium klar umrissen ist. Bevor Sie darangehen, den Code zu schreiben, muss das Konzept vollständig klar sein. Softwareprojekte bleiben vor allem auf der Strecke, weil es im frühen Entwurfsprozess keine genügenden Abstimmungen über das eigentliche Ziel des Programms gab.

Entscheidungen über Entscheidungen

Um eine Vorstellung vom Entwurfsprozess zu bekommen, untersuchen Sie die folgende Frage: »Wie sieht das Menü aus?« Als erster Menübefehl findet sich bei PostMaster wahrscheinlich der Eintrag NEUE MAIL-NACHRICHT und das führt sofort zu einem weiteren Entwurfsproblem: Wenn der Benutzer NEUE MAIL-NACHRICHT wählt, was passiert dann? Lässt man einen Editor erstellen, der seinerseits eine Mail-Nachricht erzeugt, oder lässt man eine neue Mail-Nachricht erzeugen, die dann den Editor erstellt?

Der Befehl lautet NEUE MAIL-NACHRICHT. Das Erzeugen einer neuen Mail-Nachricht scheint also auf der Hand zu liegen. Was passiert aber, wenn der Benutzer auf ABBRECHEN klickt, nachdem er mit dem Verfassen der Nachricht begonnen hat? Vielleicht wäre es die sauberste Lösung, zuerst den Editor zu erstellen und ihn dann die neue Nachricht erzeugen (und besitzen) zu lassen.

Bei dieser Lösung muss man den Editor aber für zwei unterschiedliche Betriebsarten auslegen – eine für das Erzeugen der Nachricht und eine für das Bearbeiten. Erzeugt man dagegen die Nachricht zuerst und übergibt sie an den Editor, muss nur ein Codeblock existieren, da es sich dann lediglich um die Bearbeitung einer vorhandenen Nachricht handelt.

Wird die Nachricht als Erstes erzeugt, wer erzeugt sie? Übernimmt das der zum Menübefehl gehörende Code? Wenn das so ist, bewirkt der Menübefehl dann auch, dass sich die Nachricht selbst bearbeitet, oder gehört das zur Konstruktor-Methode der Nachricht?

Auf den ersten Blick sieht es so aus, dass der Konstruktor diese Aufgabe übernehmen sollte. Immerhin muss man jedes Mal, wenn man eine Nachricht erzeugt, diese auch bearbeiten. Trotzdem ist das kein gutes Entwurfskonzept. Erstens ist es sehr gut möglich, dass die Annahme bereits falsch ist: Man kann

nämlich »vorgefertigte« Nachrichten (etwa Fehlermeldungen an den System-Operator) erzeugen, die nicht an einen Editor übergeben werden. Zweitens und wichtiger: Die Aufgabe eines Konstruktors ist das Erzeugen eines Objekts – nicht mehr und nicht weniger. Mit dem Erzeugen einer Mail ist der Job des Konstruktors abgeschlossen. Der Aufruf einer Bearbeitungsmethode im Konstruktor bringt das ganze Konzept durcheinander und macht die Mail-Nachricht gegen Fehler im Editor anfällig.

Noch ungünstiger ist es allerdings, dass die Bearbeitungsmethode eine andere Klasse aufruft: die Editor-Klasse und damit auch deren Konstruktor. Der Editor ist aber weder eine Basisklasse der Nachricht noch ist er in der Nachricht selbst enthalten. Es ist nicht gerade im Sinne des Erfinders, wenn die Konstruktion der Nachricht von der erfolgreichen Konstruktion des Editors abhängig wäre.

Sollte sich die Nachricht nicht erfolgreich erzeugen lassen, verzichtet man schließlich ganz auf den Aufruf des Editors. Im geschilderten Szenario hängt aber die erfolgreiche Erzeugung vom Aufruf des Editors ab! Die Logik gebietet, dass man zuerst den Aufruf des Nachrichten-Konstruktors vollständig abschließt, bevor der Aufruf von `Message::Edit()` erfolgt.

22.3.13 Rahmenprogramme

Bereits in frühen Entwicklungsstadien lassen sich Entwurfsprobleme mit so genannten *Rahmenprogrammen* erkennen. Dabei handelt es sich um eine Funktion, die nur zur Demonstration oder zum Testen anderer Funktionen existiert. Beispielsweise kann ein Rahmenprogramm für PostMaster ein sehr einfaches Menü bieten, das `PostMasterMessage`-Objekte erzeugt, sie manipuliert und ein paar andere Entwurfsdetails durchspielt.

Listing 22.3 zeigt eine verbesserte Definition der Klasse `PostMasterMessage` und ein einfaches Rahmenprogramm.

Listing 22.3: driverprogram.cpp – Ein Rahmenprogramm für PostMaster-Message

```
// Listing 22.3
#include <iostream>
#include <string.h>

typedef unsigned long pDate;
enum SERVICE { PostMaster, Interchange,
    CompuServe, Prodigy, AOL, Internet };

class String
{
public:
    // Konstruktoren
    String();
    String(const char *const);
```

```
15:     String(const String &);
16:     ~String();
17:
18:     // Überladene Operatoren
19:     char & operator[](int offset);
20:     char operator[](int offset) const;
21:     String operator+(const String&);
22:     void operator+=(const String&);
23:     String & operator= (const String &);
24:     friend std::ostream & operator<<
25: (std::ostream& theStream, String& theString);
26:     // Allgemeine Zugriffsfunktionen
27:     int GetLen()const { return itsLen; }
28:     const char * GetString() const { return itsString; }
29:     // static int ConstructorCount;
30:
31: private:
32:     String (int);         // Privater Konstruktor
33:     char * itsString;
34:     int itsLen;
35: };
36:
37: // Standardkonstruktor erzeugt String von 0 Bytes Länge
38: String::String()
39: {
40:     itsString = new char[1];
41:     itsString[0] = '\0';
42:     itsLen=0;
43:     // std::cout << "\tStandard-String-Konstruktor\n";
44:     // ConstructorCount++;
45: }
46:
47: // Privater (Hilfs-) Konstruktor, nur von Klassen-
48: // methoden für das Erzeugen eines neuen Strings der
49: // erforderlichen Größe verwendet.  Mit Null gefüllt.
50: String::String(int len)
51: {
52:     itsString = new char[len+1];
53:     int i;
54:     for ( i = 0; i<=len; i++)
55:         itsString[1] = '\0';
56:     itsLen=len;
57:     // std::cout << "\tString(int)-Konstruktor\n";
58:     // ConstructorCount++;
59: }
60:
61: // Konvertiert ein Zeichen-Array in einen String
62: String::String(const char * const cString)
```

```
 63: {
 64:     itsLen = strlen(cString);
 65:     itsString = new char[itsLen+1];
 66:     int i;
 67:     for ( i = 0; i<itsLen; i++)
 68:         itsString[i] = cString[i];
 69:     itsString[itsLen]='\0';
 70:     // std::cout << "\tString(char*)-Konstruktor\n";
 71:     // ConstructorCount++;
 72: }
 73:
 74: // Kopierkonstruktor
 75: String::String (const String & rhs)
 76: {
 77:     itsLen=rhs.GetLen();
 78:     itsString = new char[itsLen+1];
 79:     int i;
 80:     for (i = 0; i<itsLen;i++)
 81:         itsString[i] = rhs[i];
 82:     itsString[itsLen] = '\0';
 83:     // std::cout << "\tString(String&)-Konstruktor\n";
 84:     // ConstructorCount++;
 85: }
 86:
 87: // Destruktor, gibt reservierten Speicher frei
 88: String::~String ()
 89: {
 90:     delete [] itsString;
 91:     itsLen = 0;
 92:     // std::cout << "\tString-Destruktor\n";
 93: }
 94:
 95: String& String::operator=(const String & rhs)
 96: {
 97:     if (this == &rhs)
 98:         return *this;
 99:     delete [] itsString;
100:     itsLen=rhs.GetLen();
101:     itsString = new char[itsLen+1];
102:     int i;
103:     for (i = 0; i<itsLen;i++)
104:         itsString[i] = rhs[i];
105:     itsString[itsLen] = '\0';
106:     return *this;
107:     // std::cout << "\tString operator=\n";
108: }
109:
```

```
110: // Nicht-konstanter Offset-Operator, gibt Referenz
111: // auf Zeichen zurück, das sich damit ändern lässt
112: char & String::operator[](int offset)
113: {
114:     if (offset > itsLen)
115:         return itsString[itsLen-1];
116:     else
117:         return itsString[offset];
118: }
119:
120: // Konstanter Offset-Operator für konstante
121: // Objekte (siehe Kopierkonstruktor!)
122: char String::operator[](int offset) const
123: {
124:     if (offset > itsLen)
125:         return itsString[itsLen-1];
126:     else
127:         return itsString[offset];
128: }
129:
130: // Erzeugt einen neuen String durch Anfügen des
131: // aktuellen Strings an rhs
132: String String::operator+(const String& rhs)
133: {
134:     int  totalLen = itsLen + rhs.GetLen();
135:     String temp(totalLen);
136:     int i,j;
137:     for (i = 0; i<itsLen; i++)
138:         temp[i] = itsString[i];
139:     for (j = 0; j<rhs.GetLen(); j++, i++)
140:         temp[i] = rhs[j];
141:     temp[totalLen]='\0';
142:     return temp;
143: }
144:
145: // Ändert aktuellen String, gibt nichts zurück
146: void String::operator+=(const String& rhs)
147: {
148:     int rhsLen = rhs.GetLen();
149:     int totalLen = itsLen + rhsLen;
150:     String  temp(totalLen);
151:     int i,j;
152:     for ( i = 0; i<itsLen; i++)
153:         temp[i] = itsString[i];
154:     for ( j = 0; j<rhs.GetLen(); j++, i++)
155:         temp[i] = rhs[i-itsLen];
156:     temp[totalLen]='\0';
157:     *this = temp;
158: }
```

```
159:
160: // int String::ConstructorCount = 0;
161:
162: std::ostream& operator<<(
163:                           std::ostream& theStream,
164:                          String& theString)
165: {
166:     theStream << theString.GetString();
167:     return theStream;
168: }
169:
170: class pAddress
171: {
172: public:
173:     pAddress(SERVICE theService,
174:              const String& theAddress,
175:              const String& theDisplay):
176:         itsService(theService),
177:         itsAddressString(theAddress),
178:         itsDisplayString(theDisplay)
179:     {}
180:     // pAddress(String, String);
181:     // pAddress();
182:     // pAddress (const pAddress&);
183:     ~pAddress(){}
184:     friend std::ostream& operator<<(
185:   std::ostream& theStream, pAddress& theAddress);
186:     String& GetDisplayString()
187: { return itsDisplayString; }
188: private:
189:     SERVICE itsService;
190:     String itsAddressString;
191:     String itsDisplayString;
192: };
193:
194: std::ostream& operator<<
195:   ( std::ostream& theStream, pAddress& theAddress)
196: {
197:     theStream << theAddress.GetDisplayString();
198:     return theStream;
199: }
200:
201: class PostMasterMessage
202: {
203: public:
204: //  PostMasterMessage();
205:
```

```
206:     PostMasterMessage(const pAddress& Sender,
207:                       const pAddress& Recipient,
208:                       const String& Subject,
209:                       const pDate& creationDate);
210:
211:     ~PostMasterMessage(){}
212:
213:     void Edit(); // Ruft Editor auf dieser Nachricht auf
214:
215:     pAddress& GetSender()  { return itsSender; }
216:     pAddress& GetRecipient()  { return itsRecipient; }
217:     String& GetSubject()  { return itsSubject; }
218:     //  void SetSender(pAddress& );
219:     // Andere Zugriffsfunktionen auf Datenelemente
220:
221:     // Hier Operator-Methoden, u.a. Gleich-Operator
222:     // und Routinen zur Umwandlung der PostMaster-
223:     // Nachrichten in andere Formate.
224:
225: private:
226:     pAddress itsSender;
227:     pAddress itsRecipient;
228:     String  itsSubject;
229:     pDate itsCreationDate;
230:     pDate itsLastModDate;
231:     pDate itsReceiptDate;
232:     pDate itsFirstReadDate;
233:     pDate itsLastReadDate;
234: };
235:
236: PostMasterMessage::PostMasterMessage(
237:     const pAddress& Sender,
238:     const pAddress& Recipient,
239:     const String& Subject,
240:     const pDate& creationDate):
241:     itsSender(Sender),
242:     itsRecipient(Recipient),
243:     itsSubject(Subject),
244:     itsCreationDate(creationDate),
245:     itsLastModDate(creationDate),
246:     itsFirstReadDate(0),
247:     itsLastReadDate(0)
248: {
249:     std::cout << "PostMaster-Nachricht erzeugt. \n";
250: }
251:
```

```
252: void PostMasterMessage::Edit()
253: {
254:     std::cout << "PostMasterMessage Bearbeitungsfunktion
aufgerufen.\n";
255: }
256:
257:
258: int main()
259: {
260:     pAddress Sender(
261:         PostMaster, "jliberty@PostMaster", "Jesse Liberty");
262:     pAddress Recipient(
263:         PostMaster, "sliberty@PostMaster","Stacey Liberty");
264:     PostMasterMessage PostMasterMessage(
265:         Sender, Recipient, "Saying Hello", 0);
266:     std::cout << "Nachricht ueberpruefen... \n";
267:     std::cout << "Von:\t\t"
268:         << PostMasterMessage.GetSender() << std::endl;
269:     std::cout << "An:\t\t"
270:         << PostMasterMessage.GetRecipient() << std::endl;
271:     std::cout << "Betreff:\t"
272:         << PostMasterMessage.GetSubject() << std::endl;
273:     return 0;
274: }
```

```
PostMaster-Nachricht erzeugt.
Nachricht ueberpruefen...
Von:            Jesse Liberty
An:             Stacey Liberty
Betreff:        Begruessung
```

Zeile 5 definiert `pDate` mit `typedef` als `unsigned long`. Es ist durchaus üblich, dass man Datumswerte als lange Ganzzahlen speichert, normalerweise als Anzahl der Sekunden seit einem willkürlich gewählten Startzeitpunkt wie zum Beispiel dem 1. Januar 1900 um 00:00 Uhr. In diesem Programm handelt es sich um einen Platzhalter. Man geht davon aus, dass `pDate` schließlich in eine echte Klasse umgewandelt wird.

Zeile 6 definiert die Aufzählungskonstante `SERVICE`, damit Adressobjekte den jeweiligen Adresstyp aufzeichnen können, beispielsweise PostMaster oder CompuServe.

In den Zeilen 9 bis 168 finden Sie die bereits aus vorherigen Kapiteln bekannte Klasse `String` wieder. Die Deklaration (das heißt, die Schnittstelle) und die Implementierung wurden fast unverändert übernommen. Die `String`-Klasse kommt für eine Reihe von Elementvariablen in allen `Message`-Klassen sowie in verschiedenen anderen Klassen, auf die sich die `Message`-Klassen stützen, zum Einsatz. In dieser Hinsicht bildet die `String`-Klasse den Dreh- und Angelpunkt Ihres Programms. Eine ausgefeilte `String`-Klasse ist für die Komplettierung der `Message`-Klassen lebenswichtig.

Die Zeilen 170 bis 192 zeigen die Deklaration der Klasse `pAddress`. Momentan repräsentiert sie nur die grundlegende Funktionalität. Diese Klasse müssen Sie also weiter ausbauen, wenn die entsprechenden Fragen zu Ihrem Programm geklärt sind. Die Objekte repräsentieren wesentliche Komponenten jeder Nachricht: sowohl die Adresse des Absenders als auch des Empfängers. Ein voll funktionsfähiges `pAddress`-Objekt kann das Versenden von Nachrichten, Antworten auf Nachrichten usw. behandeln.

Das `pAddress`-Objekt verfolgt für den jeweiligen Dienst sowohl den String für die Anzeige als auch für die interne Weiterleitung. Es bleibt noch offen, ob ein gemeinsames `pAddress`-Objekt genügt oder ob man für jeden Dienst ein Objekt als Unterklasse ableiten sollte. Momentan wird der Dienst in Form einer Aufzählungskonstanten protokolliert, die als Elementvariable zu jedem `pAddress`-Objekt gehört.

Die Zeilen 201 bis 233 zeigen die Schnittstelle zur Klasse `PostMaster-Message`. In diesem Listing ist die Klasse eigenständig, bald aber werden Sie diese zum Bestandteil ihrer Vererbungshierarchie machen. Wenn Sie den Entwurf überarbeiten, um von `Message` zu erben, lassen sich einige Elementvariablen in die Basisklasse verschieben und einige Elementfunktionen werden zu redefinierten Methoden der Basisklasse.

Damit die Klasse voll funktionsfähig wird, sind noch zahlreiche Konstruktoren, Zugriffsfunktionen und andere Elementfunktionen erforderlich. Aus diesem Listing lässt sich aber erkennen, dass eine Klasse noch nicht hundertprozentig fertig sein muss, damit man einige der im Entwurf getroffenen Annahmen mithilfe eines Rahmenprogramms testen kann.

In den Zeilen 252 bis 255 ist die `Edit()`-Funktion als Stub ausgeführt und zeigt gerade noch so viele Details, dass man auf die vorgesehene Funktionalität nach Fertigstellung der Klasse schließen kann.

Die Zeilen 258 bis 274 repräsentieren das Rahmenprogramm. Momentan spielt dieses Programm lediglich ein paar Zugriffsfunktionen und das Überladen des Operators `operator<<` durch. Trotzdem bildet es einen geeigneten Ausgangspunkt für Experimente mit `PostMasterMessage`-Nachrichten und stellt ein Gerüst bereit, in dem man diese Klassen modifizieren und die Wirkungen untersuchen kann.

22.4 Fragen und Antworten

F: In welchen Punkten unterscheidet sich der objektorientierte Ansatz bei Analyse und Entwurf von Projekten grundsätzlich von anderen Lösungen?

A: Vor der Entwicklung der objektorientierten Techniken stellten sich Analytiker und Programmierer die Programme als Funktionen vor, die auf Daten arbeiten. Die objektorientierte Programmierung konzentriert sich auf die Integration von Daten und Funktionalität in selbstständigen Einheiten, die sowohl über Wissen (Daten) als auch über Fähigkeiten (Funktionen) verfügen. Prozedurale Programme konzentrieren sich andererseits auf Funktionen und deren Einwirkung auf die Daten. Man kann sagen, dass Pascal- und C-Programme aus Sammlungen von Prozeduren bestehen, während C++-Programme als Sammlungen von Klassen anzusehen sind.

F: Ist die objektorientierte Programmierung schließlich das Nonplusultra und lösen sich damit alle Programmierungsprobleme?

A: Nein, das war gar nicht beabsichtigt. Bei großen Projekten liefert allerdings der objektorientierte Ansatz bei Analyse, Entwurf und Programmierung die Werkzeuge, mit denen der Programmierer die gestiegene Komplexität der Programme in einer früher nicht möglichen Art und Weise in den Griff bekommen kann.

22.5 Workshop

Nachdem Sie sich mit der objektorientierten Analyse und dem Entwurf beschäftigt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu festigen.

22.5.1 Kontrollfragen

1. Was ist zu tun, bevor man mit der Kodierung beginnt?
2. Was ist eine Anwendungsprogrammierschnittstelle (API)?
3. Welche Aufgabe hat ein Rahmenprogramm?
4. Worin besteht der Vorteil, große Anwendungen in kleinere Blöcke aufzuteilen?

22.5.2 Übungen

1. Greifen Sie sich eine kleinere Aufgabe aus dem geschäftlichen oder privaten Bereich heraus und führen Sie die in dieser Stunde beschriebenen Schritte aus.
2. Modifizieren Sie das in Listing 22.1 (`simpleevent.cpp`) gezeigte Programm, indem Sie in jeden Destruktor eine `std::cout`-Nachricht einbauen. Führen Sie das Programm aus und stellen Sie fest, wann Destruktoren aufgerufen werden. Die Aufrufe der Konstruktoren kennen Sie bereits, da die Protokollmeldung erscheint.
3. Sehen Sie sich im Internet nach einigen C++-Klassen und Bibliotheken um. Überlegen Sie sich, ob Sie diese Klassen lieber kaufen oder eher selbst schreiben würden.

22.5.3 Antworten zu den Kontrollfragen

1. Bevor man die erste Codezeile schreibt, ist eine ganze Menge zu tun: Zusammentragen der Anforderungen (was will der Endbenutzer?), Analyse (was ist notwendig, um diese Anforderungen zu erfüllen?) und Entwurf (wie lassen sich diese Anforderungen erfüllen?) Leider verfolgen viele Firmen die Philosophie »Wir haben hier so viele Programmierer herumsitzen – warum sollen sie nicht programmieren? Egal, ob die Anforderungen schon alle vollständig vorliegen!«
2. Eine API ist ein Satz von Dokumentationen und Routinen für die Nutzung eines Dienstes. Man muss nicht wissen, wie der Code im Einzelnen funktioniert, sondern nur, wie man ihn anwendet. Die Implementierungsdetails bleiben dem Benutzer der API verborgen.
3. Mit einem Rahmenprogramm testet man andere Funktionen oder Klassen während der Entwicklung. Auch wenn das Programm nur einfach gestrickt ist, kann man sehen, wie sich der Code verhält und interagiert.
4. Vor allem lässt sich die Arbeit auf mehrere Team-Mitglieder (oder sogar mehrere Teams) aufteilen. Während Sie an der ersten Klasse arbeiten, kann schon jemand anderes die nächste Klasse fertig stellen.

KAPITEL 23

jetzt lerne ich

Templates

In dieser Stunde lernen Sie,

- was Templates sind und wie man sie verwendet,
- warum Templates eine bessere Alternative zu Makros darstellen,
- wie man Templates für Klassen erzeugt.

23.1 Was sind Templates?

In Stunde 19 haben Sie gelernt, wie man eine verkettete Liste erzeugt. Diese verkettete Liste war gut gekapselt: Die Liste kannte nur ihren Kopfzeiger, der Kopfzeiger delegierte seine Arbeit an interne Zeiger weiter usw.

Das gravierende Problem der verketteten Liste bestand darin, dass sie ausschließlich Datenobjekte behandeln konnte, für die die Liste programmiert war. Etwas anderes ließ sich nicht in der Liste speichern. Man konnte beispielsweise keine Liste mit `Car`-, `Cat`- oder anderen Objekten erstellen, die nicht vom selben Typ wie diejenigen der Originalliste waren.

Nach traditioneller Programmierung würde man neue Versionen des Objekts für jeden neuen Datentyp erzeugen. Bei einem mehr objektorientierten Ansatz erstellt man eine Basisklasse `List` und leitet davon die Klassen `CarList` und `CatsList` ab. Dann schneidet man soviel wie möglich aus der Klasse `LinkedList` aus und fügt das Ganze in die neue `CatsList`-Deklaration ein. Sobald es Ihnen aber in den Sinn kommt, eine Liste mit `Car`-Objekten zu erstellen, müssen Sie eine neue Klasse anlegen und wieder mit Ausschneiden und Einfügen arbeiten.

Dass dies keine zufrieden stellende Lösung ist, versteht sich von selbst. Mit der Zeit machen sich Erweiterungen der `List`-Klasse und ihrer abgeleiteten Klassen erforderlich. Dann wird es zum Albtraum, alle Änderungen in allen verwandten Klassen auf den gleichen Stand zu bringen.

Templates – eine relativ neue Erweiterung von C++ – bieten einen Ausweg für dieses Problem. Im Gegensatz zu den guten alten Makros sind Templates darüber hinaus Teil der Sprache, sie sind typsicher und sehr flexibel.

Templates bieten die Möglichkeit, eine allgemeine Klasse zu erzeugen. Durch die Übergabe von Typen als *Parameter* an diese Klasse lassen sich spezifische Instanzen erstellen.

23.2 Instanzen der Templates

Statt typenspezifische Listen zu erzeugen, teilt man dem Compiler mit Templates einfach mit, wie eine Liste eines beliebigen Typs zu erzeugen ist. Eine `PartsList` ist eine Liste mit Bauteilen, eine `CatsList` eine Liste mit Katzen. Diese Listen unterscheiden sich einzig und allein durch das konkrete Objekt in der Liste. Bei Templates wird der Typ des Objekts zu einem Parameter in der Definition der Klasse.

Das Erzeugen eines Objekts (von einer Klasse) oder eines spezifischen Typs von einer Template nennt man *Instanzbildung* oder *Instanzierung*, die einzelnen Klassen heißen *Instanzen* der Template.

23.3 Definition von Templates

Ein parametrisiertes `List`-Objekt (eine Template für eine Liste) erzeugt man wie folgt:

```
template <class T>    // die Template und den Parameter deklarieren
class List            // die zu parametrisierende Klasse
{
public:
   List();
   // hier steht die vollständige Klassendeklaration
};
```

Jede Deklaration und Definition einer Template-Klasse wird mit dem Schlüsselwort `template` eingeleitet. Die Parameter der Template stehen nach diesem Schlüsselwort. Es sind die Elemente, die sich mit jeder Instanz ändern. Im obigen Codefragment ändert sich zum Beispiel der Typ der Objekte, die in der Liste gespeichert sind. Eine Instanz speichert vielleicht eine Liste von Integer-Werten, während eine andere eine Liste von `Animal`-Objekten aufnimmt.

Im Beispiel ist für den Parameter das Schlüsselwort `class` mit dem nachfolgenden Bezeichner `T` angegeben. Das Schlüsselwort `class` kennzeichnet den Parameter als Typ. Der Bezeichner `T` steht als Platzhalter für den parametrisierten Typ im restlichen Teil der Template-Definition. Eine Instanz dieser Klasse ersetzt `T` jeweils durch `int`, während eine andere Instanz den Platzhalter `T` durch ein `Cat`-Objekt ersetzt.

Um eine `int`- und eine `Cat`-Instanz der parametrisierten Listenklasse zu erzeugen, schreibt man zum Beispiel:

```
List<int> anIntList;
List<Cat> aCatList;
```

Das Objekt `anIntList` ist vom Typ ganzzahlige Liste, das Objekt `aCatList` vom Typ `ListOfCats`. Man kann jetzt den Typ `List<int>` an allen Stellen verwenden, wo normalerweise eine Typangabe steht – beispielsweise beim Rückgabewert einer Funktion oder als Parameter an eine Funktion.

Listing 23.1 parametrisiert das schon bekannte `List`-Objekt. Diese Technik eignet sich hervorragend zum Erstellen von Templates: Man bringt das Objekt mit einem einzelnen Typ zum Laufen, wie es zum Beispiel in Stunde 19 erfolgt ist. Dann parametrisiert man das Objekt und nimmt damit eine Verallgemeinerung des Objekts für die Behandlung beliebiger Typen vor.

Listing 23.1: parmlist.cpp – Parametrisierte Listen (mit verketteten Listen)

```
// ************************************************
//      Listing 23.1
//
//      Demonstriert parametrisierte Liste
//
//
//
//
//
// Zeigt objektorientierte Lösung für parametrisierte verkettete
// Listen. Die Liste delegiert die eigentliche Arbeit an die
// Knoten. Die Knoten sind abstrakte Objekttypen. Es gibt
// drei Knotentypen: Kopfknoten, Endeknoten und interne
// Knoten. Nur die internen Knoten nehmen Objekte auf.
//
// Die erzeugte Objekt-Klasse dient als Objekt, das
// in der verketteten Liste gespeichert wird.
//
// ************************************************
#include <iostream>

enum { kIsSmaller, kIsLarger, kIsSame};

```

```
24: // Objekt-Klasse für die Speicherung in der Liste
25: // Jede Klasse in dieser verketteten Liste muss zwei Methoden
26: // unterstützen: Show (zeigt den Wert an) und
27: // Compare (gibt relative Position zurück)
28: class Data
29: {
30: public:
31:     Data(int val):myValue(val){}
32:     ~Data()
33:     {
34:         std::cout << "Loesche Data-Objekt mit Wert: ";
35:         std::cout << myValue << "\n";
36:     }
37:     int Compare(const Data &);
38:     void Show() { std::cout << myValue << std::endl; }
39: private:
40:     int myValue;
41: };
42:
43: // Compare entscheidet, wohin ein bestimmtes Objekt
44: // in der Liste gehört.
45: int Data::Compare(const Data & theOtherObject)
46: {
47:     if (myValue < theOtherObject.myValue)
48:         return kIsSmaller;
49:     if (myValue > theOtherObject.myValue)
50:         return kIsLarger;
51:     else
52:         return kIsSame;
53: }
54:
55: // Andere Klasse zur Aufnahme in die verkettete Liste.
56: // Jede Klasse in dieser verketteten Liste muss zwei Methoden
57: // unterstützen:
58: // Show (zeigt den Wert an) und
59: // Compare (gibt relative Position zurück)
60: class Cat
61: {
62: public:
63:     Cat(int age): myAge(age){}
64:     ~Cat()
65:     {
66:         std::cout << "Loeschen einer ";
67:         std::cout << myAge << " Jahre alten Katze.\n";
68:     }
69:     int Compare(const Cat &);
70:     void Show()
71:     {
```

```
 72:         std::cout << "Diese Katze ist ";
 73:         std::cout << myAge << " Jahre alt.\n";
 74:     }
 75: private:
 76:     int myAge;
 77: };
 78:
 79:
 80: // Compare entscheidet, wohin ein bestimmtes Objekt
 81: // in der Liste gehört.
 82: int Cat::Compare(const Cat & theOtherCat)
 83: {
 84:     if (myAge < theOtherCat.myAge)
 85:         return kIsSmaller;
 86:     if (myAge > theOtherCat.myAge)
 87:         return kIsLarger;
 88:     else
 89:         return kIsSame;
 90: }
 91:
 92:
 93: // ADT, der das Knotenobjekt in der Liste darstellt.
 94: // Alle abgeleiteten Klassen müssen Insert und Show
redefinieren.
 95: template <class T>
 96: class Node
 97: {
 98: public:
 99:     Node(){}
100:     virtual ~Node(){}
101:     virtual Node * Insert(T * theObject)=0;
102:     virtual void Show() = 0;
103: private:
104: };
105:
106: template <class T>
107: class InternalNode: public Node<T>
108: {
109: public:
110:     InternalNode(T * theObject, Node<T> * next);
111:     ~InternalNode(){ delete myNext; delete myObject; }
112:     virtual Node<T> * Insert(T * theObject);
113:     virtual void Show()
114:     {
115:         myObject->Show();
116:         myNext->Show();
117:     } // Delegieren!
118: private:
```

```
119:     T * myObject;        // das Objekt selbst
120:     Node<T> * myNext;    // zeigt auf nächsten Knoten in der
verketteten Liste
121: };
122:
123: // Der Konstruktor führt nur Initialisierungen aus.
124: template <class T>
125: InternalNode<T>::InternalNode(T * theObject, Node<T> * next):
126: myObject(theObject),myNext(next)
127: {
128: }
129:
130: // Der Kern der Listenkonstruktion. Stellt man ein
131: // neues Objekt in die Liste, wird es an den Knoten
132: // weitergereicht, der ermittelt, wohin das Objekt
133: // gehört, und es in die Liste einfügt.
134: template <class T>
135: Node<T> * InternalNode<T>::Insert(T * theObject)
136: {
137:     // Ist das neue Objekt größer oder kleiner als ich?
138:     int result = myObject->Compare(*theObject);
139:
140:     switch(result)
141:     {
142:     // Ist das neue Objekt gleich groß, kommt es vor das aktuelle.
143:     case kIsSame:        // zum nächsten case-Zweig durchfallen
144:     case kIsLarger:      // neues Objekt vor mir einordnen
145:         {
146:             InternalNode<T> * ObjectNode =
147:             new InternalNode<T>(theObject, this);
148:             return ObjectNode;
149:         }
150:     // Größer als ich, also an den nächsten Knoten
151:     // weiterreichen. ER soll sich darum kümmern.
152:     case kIsSmaller:
153:         myNext = myNext->Insert(theObject);
154:         return this;
155:     }
156:     return this;  // Tribut an Compiler
157: }
158:
159: // Endeknoten ist einfach eine Markierung
160: template <class T>
161: class TailNode : public Node<T>
162: {
163: public:
164:     TailNode(){}
165:     virtual ~TailNode(){}
```

```
166:     virtual Node<T> * Insert(T * theObject);
167:     virtual void Show() { }
168: private:
169: };
170:
171: // Wenn Objekt zu mir kommt, muss es vor mir eingefügt werden,
172: // da ich der Endeknoten bin und NICHTS nach mir kommt.
173: template <class T>
174: Node<T> * TailNode<T>::Insert(T * theObject)
175: {
176:     InternalNode<T> * ObjectNode =
177:     new InternalNode<T>(theObject, this);
178:     return ObjectNode;
179: }
180:
181: // Kopfknoten hat kein Objekt, sondern zeigt einfach
182: // auf den Beginn der Liste.
183: template <class T>
184: class HeadNode : public Node<T>
185: {
186: public:
187:     HeadNode();
188:     virtual ~HeadNode() { delete myNext; }
189:     virtual Node<T> * Insert(T * theObject);
190:     virtual void Show() { myNext->Show(); }
191: private:
192:     Node<T> * myNext;
193: };
194:
195: // Nach Erzeugen des Kopfknotens erzeugt dieser
196: // sofort den Endeknoten.
197: template <class T>
198: HeadNode<T>::HeadNode()
199: {
200:     myNext = new TailNode<T>;
201: }
202:
203: // Vor dem Kopf kommt nichts, also das Objekt einfach
204: // an den nächsten Knoten weiterreichen
205: template <class T>
206: Node<T> * HeadNode<T>::Insert(T * theObject)
207: {
208:     myNext = myNext->Insert(theObject);
209:     return this;
210: }
211:
212: // Ich stehe im Mittelpunkt, mache aber gar nichts.
213: template <class T>
```

```
214: class LinkedList
215: {
216: public:
217:     LinkedList();
218:     ~LinkedList() { delete myHead; }
219:     void Insert(T * theObject);
220:     void ShowAll() { myHead->Show(); }
221: private:
222:     HeadNode<T> * myHead;
223: };
224:
225:  // Bei Geburt erzeuge ich den Kopfknoten.
226:  // Er erzeugt den Endeknoten.
227:  // Eine leere Liste zeigt damit auf den Kopf, dieser
228:  // zeigt auf das Ende, dazwischen ist nichts.
229: template <class T>
230: LinkedList<T>::LinkedList()
231: {
232:     myHead = new HeadNode<T>;
233: }
234:
235: // Delegieren, delegieren, delegieren
236: template <class T>
237: void LinkedList<T>::Insert(T * pObject)
238: {
239:     myHead->Insert(pObject);
240: }
241:
242: // Rahmenprogramm zum Testen
243: int main()
244: {
245:     Cat * pCat;
246:     Data * pData;
247:     int val;
248:     LinkedList<Cat>  ListOfCats;
249:     LinkedList<Data> ListOfData;
250:
251:     // Benutzer zum Erzeugen von Werten auffordern.
252:     // Diese Werte in die Liste stellen.
253:     for (;;)
254:     {
255:         std::cout << "Welcher Wert? (0 zum Beenden): ";
256:         std::cin >> val;
257:         if (!val)
258:             break;
259:         pCat = new Cat(val);
260:         pData= new Data(val);
```

```
261:         ListOfCats.Insert(pCat);
262:         ListOfData.Insert(pData);
263:     }
264:
265:     // Jetzt Liste durchlaufen und Objekt anzeigen.
266:     std::cout << "\n";
267:     ListOfCats.ShowAll();
268:     std::cout << "\n";
269:     ListOfData.ShowAll();
270:     std::cout << "\n ************ \n\n";
271:     return 0;  // Die Listen verlieren den Gültigkeitsbereich
                    // und werden abgebaut!
272: }
```

```
Welcher Wert? (0 zum Beenden): 5
Welcher Wert? (0 zum Beenden): 13
Welcher Wert? (0 zum Beenden): 2
Welcher Wert? (0 zum Beenden): 9
Welcher Wert? (0 zum Beenden): 7
Welcher Wert? (0 zum Beenden): 0

Diese Katze ist 2 Jahre alt.
Diese Katze ist 5 Jahre alt.
Diese Katze ist 7 Jahre alt.
Diese Katze ist 9 Jahre alt.
Diese Katze ist 13 Jahre alt.

2
5
7
9
13

 ************

Loesche Data-Objekt mit Wert: 13
Loesche Data-Objekt mit Wert: 9
Loesche Data-Objekt mit Wert: 7
Loesche Data-Objekt mit Wert: 5
Loesche Data-Objekt mit Wert: 2
Loeschen einer 13 Jahre alten Katze.
Loeschen einer 9 Jahre alten Katze.
Loeschen einer 7 Jahre alten Katze.
Loeschen einer 5 Jahre alten Katze.
Loeschen einer 2 Jahre alten Katze.
```

Als Erstes fällt die Ähnlichkeit zum Listing in Stunde 19 auf. Geändert wurden vor allem die Klassendeklarationen und Methoden. Diese enthalten jetzt den Vorsatz:

```
template <class T>
```

Damit parametrisiert man die Liste auf einen Typ, den man später bei der Instanzierung der Liste festlegt. Zum Beispiel sieht dann die Deklaration der Klasse `Node` wie folgt aus:

```
template <class T>
class Node
```

Damit zeigt man, dass `Node` nicht als Klasse an sich existiert, sondern dass man beispielsweise Knoten (`Nodes`) von `Cat`- oder von `Data`-Objekten instanzieren will. Das `T` steht als Platzhalter für den zu übergebenden Typ.

Der interne Knoten `InternalNode` wird damit zu `InternalNode<T>` (zu lesen als »`InternalNode` von `T`«). `InternalNode<T>` zeigt auch nicht mehr auf ein `Data`-Objekt und einen anderen `Node`, sondern auf ein `T` (ein Objekt von welchem Typ auch immer) und einen `Node<T>`. Die Zeilen 119 und 120 zeigen diese Konstruktion.

Sehen Sie sich genau die Definition von `Insert` in den Zeilen 134 bis 157 an. Die Logik ist gleich geblieben, anstelle des spezifischen Typs (`Data`) steht aber jetzt `T`. Der Parameter in Zeile 135 ist demzufolge als Zeiger auf ein `T` deklariert. Wenn wir später die konkrete Liste instanzieren, ersetzt der Compiler das `T` durch den richtigen Typ (zum Beispiel `Data` oder `Cat`).

Vor allem aber ist der interne Knoten `InternalNode` weiterhin arbeitsfähig, unabhängig vom tatsächlichen Typ. Der Knoten fordert die Objekte auf, sich selbst zu vergleichen, und kümmert sich nicht darum, ob der Vergleich von `Cat`-Objekten genauso abläuft wie der Vergleich von `Data`-Objekten. In der Tat lässt sich das `Cat`-Objekt neu schreiben, sodass es nicht mehr das Alter, sondern das Geburtsdatum speichert und das jeweilige Alter dynamisch berechnet. `InternalNode` kümmert das herzlich wenig.

23.4 Template-Elemente

Template-Elemente kann man genauso wie jeden anderen Typ behandeln. Sie lassen sich bei Parametern und Rückgabewerten von Funktionen sowohl als Referenz als auch als Wert übergeben. Listing 23.2 zeigt die Übergabe von Template-Objekten. Vergleichen Sie dieses Listing mit dem Code in Listing 23.1.

Listing 23.2: passparmlist.cpp – Parametrisierte Listen (Übergabe als Referenz, verkettete Listen)

```
// **********************************************
//      Listing 23.2
//
//     Demonstriert parametrisierte Liste
//
//
//
//
//
// Zeigt objektorientierte Lösung für parametrisierte verkettete
// Listen. Die Liste delegiert die eigentliche Arbeit an die
// Knoten. Die Knoten sind abstrakte Objekttypen. Es gibt
// drei Knotentypen: Kopfknoten, Endeknoten und interne
// Knoten. Nur die internen Knoten nehmen Objekte auf.
//
// Die erzeugte Objekt-Klasse dient als Objekt, das
// in der verketteten Liste gespeichert wird.
//
// **********************************************

#include <iostream>

enum { kIsSmaller, kIsLarger, kIsSame};

// Objekt-Klasse für die Speicherung in der Liste
// Jede Klasse in dieser verketteten Liste muss zwei Methoden
// unterstützen: Show (zeigt den Wert an) und
// Compare (gibt relative Position zurück)
class Data
{
public:
    Data(int val):myValue(val){}
    ~Data()
    {
        std::cout << "Loesche Data-Objekt mit Wert: ";
        std::cout << myValue << "\n";
    }
    int Compare(const Data &);
    void Show() { std::cout << myValue << std::endl; }
private:
    int myValue;
};

// Compare entscheidet, wohin ein bestimmtes Objekt
// in der Liste gehört.
int Data::Compare(const Data & theOtherObject)
{
    if (myValue < theOtherObject.myValue)
```

```
49:         return kIsSmaller;
50:     if (myValue > theOtherObject.myValue)
51:         return kIsLarger;
52:     else
53:         return kIsSame;
54: }
55:
56: // Andere Klasse zur Aufnahme in die verkettete Liste.
57: // Jede Klasse in dieser verketteten Liste muss zwei Methoden
58: // unterstützen:
59: // Show (zeigt den Wert an) und
60: // Compare (gibt relative Position zurück)
61: class Cat
62: {
63: public:
64:     Cat(int age): myAge(age){}
65:     ~Cat()
66:     {
67:         std::cout << "Loeschen einer " << myAge
68:             << " Jahre alten Katze.\n";
69:     }
70:     int Compare(const Cat &);
71:     void Show()
72:     {
73:         std::cout << "Diese Katze ist " << myAge
74:             << " Jahre alt.\n";
75:     }
76: private:
77:     int myAge;
78: };
79:
80:
81: // Compare entscheidet, wohin ein bestimmtes Objekt
82: // in der Liste gehört.
83: int Cat::Compare(const Cat & theOtherCat)
84: {
85:     if (myAge < theOtherCat.myAge)
86:         return kIsSmaller;
87:     if (myAge > theOtherCat.myAge)
88:         return kIsLarger;
89:     else
90:         return kIsSame;
91: }
92:
93:
94: // ADT, der das Knotenobjekt in der Liste darstellt.
95: // Alle abgeleiteten Klassen müssen Insert und Show redefinieren.
```

```
 96: template <class T>
 97: class Node
 98: {
 99: public:
100:     Node(){}
101:     virtual ~Node(){}
102:     virtual Node * Insert(T * theObject)=0;
103:     virtual void Show() = 0;
104: private:
105: };
106:
107: template <class T>
108: class InternalNode: public Node<T>
109: {
110: public:
111:     InternalNode(T * theObject, Node<T> * next);
112:     virtual ~InternalNode(){ delete myNext; delete myObject; }
113:     virtual Node<T> * Insert(T * theObject);
114:     virtual void Show() // Delegieren!
115:     {
116:         myObject->Show(); myNext->Show();
117:     }
118: private:
119:     T * myObject;        // das Objekt selbst
120:     Node<T> * myNext;    // zeigt auf nächsten Knoten in der
                             // verketteten Liste
121: };
122:
123: // Der Konstruktor führt nur Initialisierungen aus.
124: template <class T>
125: InternalNode<T>::InternalNode(T * theObject, Node<T> * next):
126: myObject(theObject),myNext(next)
127: {
128: }
129:
130: // Der Kern der Listenkonstruktion. Stellt man ein
131: // neues Objekt in die Liste, wird es an den Knoten
132: // weitergereicht, der ermittelt, wohin das Objekt
133: // gehört, und es in die Liste einfügt.
134: template <class T>
135: Node<T> * InternalNode<T>::Insert(T * theObject)
136: {
137:     // Ist das neue Objekt größer oder kleiner als ich?
138:     int result = myObject->Compare(*theObject);
139:
140:     switch(result)
141:     {
142:     // Ist das neue Objekt gleich groß, kommt es vor das aktuelle.
```

```
143:     case kIsSame:        // zum nächsten case-Zweig "durchfallen"
144:     case kIsLarger:      // neues Objekt vor mir einordnen
145:         {
146:             InternalNode<T> * ObjectNode =
147:             new InternalNode<T>(theObject, this);
148:             return ObjectNode;
149:         }
150:     // Größer als ich, also an den nächsten Knoten
151:     // weiterreichen. ER soll sich darum kümmern.
152:     case kIsSmaller:
153:         myNext = myNext->Insert(theObject);
154:         return this;
155:     }
156:     return this;  // Tribut an Compiler
157: }
158:
159:
160: // Endeknoten ist einfach eine Markierung
161: template <class T>
162: class TailNode : public Node<T>
163: {
164: public:
165:     TailNode(){}
166:     virtual ~TailNode(){}
167:     virtual Node<T> * Insert(T * theObject);
168:     virtual void Show() { }
169: private:
170: };
171:
172: // Wenn Objekt zu mir kommt, muss es vor mir eingefügt werden,
173: // da ich der Endeknoten bin und NICHTS nach mir kommt.
174: template <class T>
175: Node<T> * TailNode<T>::Insert(T * theObject)
176: {
177:     InternalNode<T> * ObjectNode =
178:         new InternalNode<T>(theObject, this);
179:     return ObjectNode;
180: }
181:
182: // Kopfknoten hat kein Objekt, sondern zeigt einfach
183: // auf den Beginn der Liste.
184: template <class T>
185: class HeadNode : public Node<T>
186: {
187: public:
188:     HeadNode();
189:     virtual ~HeadNode() { delete myNext; }
190:     virtual Node<T> * Insert(T * theObject);
```

```
191:     virtual void Show() { myNext->Show(); }
192: private:
193:     Node<T> * myNext;
194: };
195:
196: // Nach Erzeugen des Kopfknotens erzeugt dieser
197: // sofort den Endeknoten.
198: template <class T>
199: HeadNode<T>::HeadNode()
200: {
201:     myNext = new TailNode<T>;
202: }
203:
204: // Vor dem Kopf kommt nichts, also das Objekt einfach
205: // an den nächsten Knoten weiterreichen
206: template <class T>
207: Node<T> * HeadNode<T>::Insert(T * theObject)
208: {
209:     myNext = myNext->Insert(theObject);
210:     return this;
211: }
212:
213: // Ich stehe im Mittelpunkt, mache aber gar nichts.
214: template <class T>
215: class LinkedList
216: {
217: public:
218:     LinkedList();
219:     ~LinkedList() { delete myHead; }
220:     void Insert(T * theObject);
221:     void ShowAll() { myHead->Show(); }
222: private:
223:     HeadNode<T> * myHead;
224: };
225:
226:  // Bei Geburt erzeuge ich den Kopfknoten.
227:  // Er erzeugt den Endeknoten.
228:  // Eine leere Liste zeigt damit auf den Kopf, dieser
229:  // zeigt auf das Ende, dazwischen ist nichts.
230: template <class T>
231: LinkedList<T>::LinkedList()
232: {
233:     myHead = new HeadNode<T>;
234: }
235:
236: // Delegieren, delegieren, delegieren
237: template <class T>
238: void LinkedList<T>::Insert(T * pObject)
```

```
239: {
240:     myHead->Insert(pObject);
241: }
242:
243: void myFunction(LinkedList<Cat>& ListOfCats);
244: void myOtherFunction(LinkedList<Data>& ListOfData);
245:
246: // Rahmenprogramm zum Testen
247: int main()
248: {
249:     LinkedList<Cat>  ListOfCats;
250:     LinkedList<Data> ListOfData;
251:
252:     myFunction(ListOfCats);
253:     myOtherFunction(ListOfData);
254:
255:     // Jetzt Liste durchlaufen und Objekt anzeigen.
256:     std::cout << "\n";
257:     ListOfCats.ShowAll();
258:     std::cout << "\n";
259:     ListOfData.ShowAll();
260:     std::cout << "\n ************ \n\n";
261:     return 0;  // Die Listen verlieren den Gültigkeitsbereich und
                   // werden abgebaut!
262: }
263:
264: void myFunction(LinkedList<Cat>& ListOfCats)
265: {
266:     Cat * pCat;
267:     int val;
268:
269:     // Benutzer zum Erzeugen von Werten auffordern.
270:     // Diese Werte in die Liste stellen.
271:     for (;;)
272:     {
273:         std::cout << "\nWie alt ist Ihre Katze? (0 zum Beenden): ";
274:         std::cin >> val;
275:         if (!val)
276:             break;
277:         pCat = new Cat(val);
278:         ListOfCats.Insert(pCat);
279:     }
280:
281: }
282:
```

```
283: void myOtherFunction(LinkedList<Data>& ListOfData)
284: {
285:     Data * pData;
286:     int val;
287:
288:     // Benutzer zum Erzeugen von Werten auffordern.
289:     // Diese Werte in die Liste stellen.
290:     for (;;)
291:     {
292:         std::cout << "\nWelcher Wert? (0 zum Beenden): ";
293:         std::cin >> val;
294:         if (!val)
295:             break;
296:         pData = new Data(val);
297:         ListOfData.Insert(pData);
298:     }
299:
300: }
```

```
Wie alt ist Ihre Katze? (0 zum Beenden): 12

Wie alt ist Ihre Katze? (0 zum Beenden): 2

Wie alt ist Ihre Katze? (0 zum Beenden): 14

Wie alt ist Ihre Katze? (0 zum Beenden): 6

Wie alt ist Ihre Katze? (0 zum Beenden): 0

Welcher Wert? (0 zum Beenden): 3

Welcher Wert? (0 zum Beenden): 9

Welcher Wert? (0 zum Beenden): 1

Welcher Wert? (0 zum Beenden): 5

Welcher Wert? (0 zum Beenden): 0

Diese Katze ist 2 Jahre alt.
Diese Katze ist 6 Jahre alt.
Diese Katze ist 12 Jahre alt.
Diese Katze ist 14 Jahre alt.
```

```
1
3
5
9

 ************

Loesche Data-Objekt mit Wert: 9
Loesche Data-Objekt mit Wert: 5
Loesche Data-Objekt mit Wert: 3
Loesche Data-Objekt mit Wert: 1
Loeschen einer 14 Jahre alten Katze.
Loeschen einer 12 Jahre alten Katze.
Loeschen einer 6 Jahre alten Katze.
Loeschen einer 2 Jahre alten Katze.
```

Dieser Code entspricht fast dem vorherigen Beispiel. Jetzt haben wir aber die `LinkedLists` als Referenz an die betreffenden Funktionen zur Verarbeitung übergeben. Dieser Mechanismus ist sehr leistungsfähig. Nach der Instanzierung der Listen kann man sie als normale Typen behandeln, an Funktionen übergeben und als Werte zurückgeben.

23.5 Die Standard Template Library

Der C++-Standard definiert eine *Standard Template Library* (STL). Jeder Compiler, der mit dem Standard kompatibel sein will, sollte die STL einbinden. Es handelt sich um eine Bibliothek mit Template-basierten Container-Klassen, einschließlich Vektoren, Listen, Warteschlangen und Stacks. Die STL enthält auch eine Reihe allgemeiner Algorithmen, beispielsweise Routinen zum Sortieren und Suchen.

Die STL ist als Alternative zur »Neuerfindung des Rades« bei üblichen Standardroutinen gedacht. Die Bibliothek ist ausgetestet und fehlerbereinigt, bietet eine hohe Leistung und ist kostenlos! Der wichtigste Punkt ist die Wiederverwendbarkeit. Wenn man einmal den Umgang mit einem STL-Container beherrscht, kann man ihn in allen Programmen verwenden, ohne ihn neu erfinden zu müssen.

23.6 Fragen und Antworten

F: Warum verwendet man Templates, wenn sich Makros ebenfalls eignen?

A: Templates sind typsicher und integraler Bestandteil der Sprache.

F: Worin liegt der Unterschied zwischen dem parametrisierten Typ einer Template-Funktion und den Parametern an eine normale Funktion?

A: Eine reguläre Funktion (keine Template) übernimmt Parameter, mit denen sie direkt arbeitet. Bei einer Template-Funktion lässt sich der Typ eines bestimmten Parameters an die Funktion parametrisieren.

F: Wann verwendet man Templates und wann Vererbung?

A: Verwenden Sie Templates, wenn das gesamte – oder nahezu das gesamte – Verhalten gleich bleibt und sich nur der Typ des Elements, mit dem die Klasse arbeitet, verändert. Wenn man laufend Klassen kopiert und lediglich den Typ von Elementen der Klasse ändert, sollte man den Einsatz einer Template in Betracht ziehen.

23.7 Workshop

Nachdem Sie Templates kennen gelernt haben, können Sie einige Fragen beantworten und eine Reihe von Übungen absolvieren, um Ihr Wissen zu diesem Thema zu festigen.

23.7.1 Kontrollfragen

1. Woran erkennt der Compiler, dass Sie eine Template und keine reguläre Klasse definieren?
2. Wie ermittelt die verkettete Liste in Listing 23.1, in welcher Reihenfolge das Objekt innerhalb der konstruierten verketteten Liste zu speichern ist?
3. Wie deklarieren Sie bei einer Template ein Objekt dieses Typs und die Klasse, die die Template enthält?
4. Woher weiß der Compiler, wann die in der verketteten Liste enthaltenen Objekte abzubauen sind?

23.7.2 Übungen

1. Vergleichen Sie Listing 23.1 (`parmlist.cpp`) mit Listing 19.1 (`linkedlist.cpp`). Dabei werden Sie feststellen, wie ähnlich sich die beiden Listings tatsächlich sind!
2. Informieren Sie sich in der Dokumentation für Ihren Compiler, welche Klassen der Standard Template Library verfügbar sind. Beim Borland-Compiler wählen Sie HILFE | HILFETHEMEN und klicken dann im linken Fensterbereich auf *Referenz*.
3. Vergleichen Sie die Template für eine verkettete Liste in Listing 23.1 mit der Dokumentation für die STL-Listenklasse Ihres Compilers. Wo gibt es Ähnlichkeiten und wo Unterschiede?

23.7.3 Antworten zu den Kontrollfragen

1. Der Compiler erkennt das am Präfix `template <class T>`.
2. Zeile 138 ruft die Funktion `Compare()` für das jeweilige Objekt auf. Es ist Ihre Aufgabe als Programmierer, das geeignete Verhalten in dieser Funktion zu realisieren (sehen Sie sich als Beispiel die Funktionen an, die auf den Zeilen 45 und 82 beginnen). Hätten wir eine `Dog`-Klasse in der verketteten Liste, würden wir dafür auch eine Vergleichsfunktion benötigen.
3. Sehen Sie sich in Listing 23.1 die Zeilen 248 und 249 an. Die einfache Form lautet `template_name <contained_class_name> template_object_name`.
4. Wenn die `main()`-Funktion endet, verliert die verkettete Liste ihren Gültigkeitsbereich und der Destruktor für diese Klasse wird aufgerufen. Da der Speicherplatz für die einzelnen (enthaltenen) Objekte dynamisch zugewiesen wird, muss ihn die verkettete Liste auch wieder freigeben. Das geschieht in Zeile 111 von Listing 23.1.

KAPITEL 24

jetzt lerne ich

Exceptions und Fehlerbehandlung

In dieser letzten Stunde lernen Sie,

- was Exceptions sind,
- wie man Exceptions verwendet und welche Probleme sie aufwerfen,
- wie man fehlerfreien Code erzeugt,
- wie es in Zukunft weitergeht.

24.1 Fehler

Der Code, den Sie bisher in diesem Buch kennen gelernt haben, diente nur Demonstrationszwecken. Fehlerbehandlungen waren nicht vorgesehen, um nicht von den jeweiligen Themen abzulenken. In echten Programmen muss man allerdings Fehlerbedingungen in Betracht ziehen. In der Tat kann das Abfangen möglicher Fehler und die Fehlerbehandlung den größten Teil des Codes ausmachen!

Spötter behaupten, dass Städte wie Kartenhäuser zusammenbrechen würden, wenn sie genauso wie Software gebaut wären. Tatsache ist, dass viele kommerzielle Programme selbst namhafter Anbieter mit Fehlern gespickt sind – und zwar mit schwerwiegenden.

Das ist natürlich kein Freibrief für die eigene Programmierung. Das Schreiben robuster, fehlerfreier Programme sollte höchste Priorität für jeden ernsthaften Programmierer haben. Fehlerhafter, instabiler Code gehört zu den größten

Problemen der Software-Industrie und die Kosten für Testen, Suchen und Beseitigen von Fehlern verschlingen einen Großteil des Budgets.

Programmfehler lassen sich in verschiedene Kategorien gliedern. Die erste betrifft die Logik: Das Programm läuft zwar, die Algorithmen sind aber nicht ausreichend durchdacht. Zur zweiten Kategorie gehören die Syntaxfehler: falsche Anweisungen, Funktionen oder Strukturen. Die meisten Fehler fallen in diese beiden Gruppen und hier suchen die Programmierer auch zuerst. Wesentlich schwerer sind Fehler zu finden, die sich erst bei unerwarteten Benutzerhandlungen zeigen. Diese kleinen logischen Bomben liegen auf der Lauer und warten nur darauf, dass jemand »auf die falsche Stelle tritt«. Bis dahin lief alles wunderbar, nun aber explodiert das Programm.

Untersuchungen und praktische Erfahrungen beweisen, dass die Beseitigung eines Problems um so mehr kostet, je später man im Entwicklungsprozess darauf stößt. Am billigsten ist es, Fehler von vornherein zu vermeiden. Auf der Kostenskala kommen dann Fehler, die der Compiler bemängelt. Entsprechend der C++-Standards werden in die Compiler immer mehr Mechanismen eingebaut, um Fehler schon zur Kompilierzeit aufzudecken.

Fehler in einem Programm, das sich zwar kompilieren lässt, aber bereits beim ersten Test abstürzt, sind wesentlich leichter und damit kostengünstiger zu beseitigen, als verborgene Fehler, die erst nach gewisser Zeit zum Crash führen.

Neben logischen und syntaktischen Fehlern stellt die Stabilität der Programme eines der größeren Probleme dar. Solange sich der Benutzer gegenüber dem Programm »anständig« verhält, läuft alles wie erwartet. Gibt er aber zum Beispiel bei einer angeforderten Zahl nicht nur Ziffern, sondern auch Buchstaben ein, hängt sich das Programm auf. Andere Abstürze sind beispielsweise auf Speicherüberlauf, die fehlende Diskette im Laufwerk oder eine gestörte Modemverbindung zurückzuführen.

Gegen diese Instabilitäten kämpfen Programmierer mit *kugelsicheren* Programmen an, die alle Eventualitäten zur Laufzeit des Programms behandeln können – angefangen bei seltsamen Benutzereingaben bis hin zur Speicherknappheit. Diese Form der Programmierung ist so etwas wie eine »defensive Fahrweise«, bei der man rücksichtsloses Verhalten der anderen Verkehrsteilnehmer einkalkuliert, auf außergewöhnliche Vorkommnisse vorbereitet ist und annimmt, durch umsichtiges Verhalten einen Unfall vermeiden zu können.

Programmfehler sind zu differenzieren in Syntaxfehler, die der Programmierer durch Verwendung syntaktisch falscher Strukturen produziert hat, logische Fehler, die ebenfalls auf den Programmierer zurückgehen, weil er das Problem missverstanden hat oder nicht genau weiß, wie es zu lösen ist, und Exceptions (Ausnahmen), die ihre Ursache in ungewöhnlichen aber vorhersehbaren Problemen wie zum Beispiel knapp werdenden Ressourcen (Speicher oder Festplattenplatz) haben.

24.1.1 Das Unerwartete behandeln

Programmierer arbeiten mit leistungsfähigen Compilern und durchsetzen ihren Code mit `assert`-Anweisungen, um Programmierfehler aufzuspüren. Logischen Fehlern rücken sie mit der Überprüfung des Entwurfs und ausgiebigen Testphasen zu Leibe.

Exceptions (Ausnahmen) sind etwas ganz anderes. Man kann unerwartete Umstände nicht ausschließen, muss sich aber darauf vorbereiten. Von Zeit zu Zeit kommt es beim Benutzer zur Speicherknappheit. Was unternehmen Sie dann in Ihrem Programm? Die Auswahl ist ziemlich begrenzt:

- das Programm abstürzen lassen,
- den Benutzer informieren und das Programm ordnungsgemäß herunterfahren,
- den Benutzer informieren und ihm die Möglichkeit bieten, den Fehler zu beheben und die Arbeit fortzusetzen,
- korrigierend eingreifen und das Programm fortsetzen, ohne den Benutzer zu belästigen.

Es ist zwar nicht erforderlich und auch nicht wünschenswert, für jeden Ausnahmezustand eine automatische Fehlerkorrektur vorzusehen, dennoch muss man sich etwas Besseres einfallen lassen, als einfach einen Programmabsturz in Kauf zu nehmen.

Die Exception-Behandlung in C++ bietet eine typensichere, in die Sprache integrierte Methode, um vorhersehbare aber ungewöhnliche Umstände während eines Programmlaufs zu meistern.

24.2 Exceptions

In C++ ist eine Exception (Ausnahme) ein Objekt, das aus einem Codebereich, wo das Problem auftritt, an den Teil des Codes übergeben wird, der das Problem behandeln soll. Der Typ der Exception bestimmt, welcher Code die Behandlung übernimmt, und der Inhalt des ausgelösten Objekts (falls vorhanden) lässt sich für Rückmeldungen an den Benutzer einsetzen.

Exceptions beruhen auf einem einfachen Konzept:

- Die eigentliche Zuweisung von Ressourcen (beispielsweise die Zuweisung von Speicher oder das Sperren einer Datei) erfolgt in der Regel auf einer systemnahen Ebene des Programms.
- Die Programmlogik für die Reaktion auf eine gescheiterte Operation (Speicher lässt sich nicht zuweisen, die Datei lässt sich nicht sperren) ist

normalerweise in einer höheren Ebene des Programms angesiedelt, die auch den Code für den Dialog mit dem Benutzer enthält.

- Exceptions verkürzen den Weg vom Code, der die Zuweisung der Ressourcen übernimmt, zum Code, der den Fehlerzustand behandelt. Dazwischen liegende Funktionsebenen erhalten die Möglichkeit, Speicherzuweisungen zu bereinigen, müssen sich aber um die Weiterleitung der Fehlerbedingung in keiner Weise kümmern.

24.2.1 Komponenten einer Exception-Behandlung

Codebereiche, die ein Problem hervorrufen und damit Exceptions auslösen können, schließt man in `try`-Blöcke ein.

Zum Beispiel:

```
try
{
  SomeDangerousFunction(); // eine gefährliche Funktion
}
```

`catch`-Blöcke behandeln die vom `try`-Block initiierten Exceptions. Zum Beispiel:

```
try
{
  SomeDangerousFunction();
}
catch(OutOfMemory)
{
  // Maßnahmen gegen aufgetretenen Speichermangel unternehmen
}
catch(FileNotFound)
{
  // Maßnahme, wenn eine Datei nicht gefunden wurde
}
```

Beim Einsatz von Exceptions sind die folgenden grundlegenden Schritte auszuführen:

1. Ermitteln der Programmbereiche, in denen Operationen zu einer Exception führen können. Diese Bereiche in `try`-Blöcke einschließen.
2. `catch`-Blöcke erzeugen, um eventuell ausgelöste Exceptions abzufangen, den zugewiesenen Speicher aufzuräumen und den Benutzer in geeigneter Weise zu informieren. Listing 24.1 demonstriert die Verwendung von `try`- und `catch`-Blöcken.

Exceptions auslösen und abfangen

Nach dem Auslösen einer Exception geht die Programmsteuerung an den `catch`-Block über, der unmittelbar auf den aktuellen `try`-Block folgt.

Listing 24.1: exceptions.cpp – Eine Exception auslösen

```
 1: // Listing 24.1 Exceptions auslösen
 2: #include <iostream>
 3:
 4: const int DefaultSize = 10;
 5:
 6: // Exception-Klasse definieren
 7: class xBoundary
 8: {
 9: public:
10:     xBoundary() {}
11:     ~xBoundary() {}
12: private:
13: };
14:
15: class Array
16: {
17: public:
18:     // Konstruktoren
19:     Array(int itsSize = DefaultSize);
20:     Array(const Array &rhs);
21:     ~Array() { delete [] pType;}
22:
23:     // Operatoren
24:     Array& operator=(const Array&);
25:     int& operator[](int offSet);
26:     const int& operator[](int offSet) const;
27:
28:     // Zugriffsfunktionen
29:     int GetitsSize() const { return itsSize; }
30:
31:     // Friend-Funktion
32:     friend std::ostream& operator<< (std::ostream&, const Array&);
33:
34: private:
35:     int *pType;
36:     int  itsSize;
37: };
38:
39:
40: Array::Array(int size):
41: itsSize(size)
42: {
43:     pType = new int[size];
```

```
44:     for (int i = 0; i<size; i++)
45:         pType[i] = 0;
46: }
47:
48:
49: Array& Array::operator=(const Array &rhs)
50: {
51:     if (this == &rhs)
52:         return *this;
53:     delete [] pType;
54:     itsSize = rhs.GetitsSize();
55:     pType = new int[itsSize];
56:     for (int i = 0; i<itsSize; i++)
57:         pType[i] = rhs[i];
58:     return *this;
59: }
60:
61: Array::Array(const Array &rhs)
62: {
63:     itsSize = rhs.GetitsSize();
64:     pType = new int[itsSize];
65:     for (int i = 0; i<itsSize; i++)
66:         pType[i] = rhs[i];
67: }
68:
69:
70: int& Array::operator[](int offSet)
71: {
72:     int size = GetitsSize();
73:     if (offSet >= 0 && offSet < size)
74:         return pType[offSet];
75:     throw xBoundary();
76:     return pType[offSet]; // Tribut an Compiler!
77: }
78:
79:
80: const int& Array::operator[](int offSet) const
81: {
82:     int mysize = GetitsSize();
83:     if (offSet >= 0 && offSet < mysize)
84:         return pType[offSet];
85:     throw xBoundary();
86:     return pType[offSet]; // Tribut an Compiler!
87: }
88:
89: std::ostream& operator<< (std::ostream& output,
90:                           const Array& theArray)
91: {
```

```
 92:     for (int i = 0; i<theArray.GetitsSize(); i++)
 93:         output << "[" << i << "] " << theArray[i] << std::endl;
 94:     return output;
 95: }
 96:
 97: int main()
 98: {
 99:     Array intArray(20);
100:     try
101:     {
102:         for (int j = 0; j< 100; j++)
103:         {
104:             intArray[j] = j;
105:             std::cout << "intArray[" << j
106:                 << "] OK..." << std::endl;
107:         }
108:     }
109:     catch (xBoundary)
110:     {
111:         std::cout << "Kann Ihre Eingabe nicht verarbeiten.\n";
112:     }
113:     std::cout << "Fertig.\n";
114:     return 0;
115: }
```

```
intArray[0] OK...
intArray[1] OK...
intArray[2] OK...
intArray[3] OK...
intArray[4] OK...
intArray[5] OK...
intArray[6] OK...
intArray[7] OK...
intArray[8] OK...
intArray[9] OK...
intArray[10] OK...
intArray[11] OK...
intArray[12] OK...
intArray[13] OK...
intArray[14] OK...
intArray[15] OK...
intArray[16] OK...
intArray[17] OK...
intArray[18] OK...
intArray[19] OK...
Kann Ihre Eingabe nicht verarbeiten.
Fertig.
```

Listing 24.1 zeigt eine abgespeckte Version der `Array`-Klasse, die nur zur Demonstration von Exceptions erzeugt wurde. Die Zeilen 7 bis 13 deklarieren die einfache Ausnahmeklasse `xBoundary`. In dieser Klasse gibt es keinerlei Hinweise darauf, dass es sich um eine Exception-Klasse handelt. In der Tat eignet sich jede beliebige Klasse mit beliebigem Namen und einer beliebigen Anzahl von Methoden und Variablen als Exception-Klasse. Zur Exception wird diese Klasse lediglich dadurch, dass sie mit `throw` (wie zum Beispiel in Zeile 75) ausgelöst und mit `catch` (wie in Zeile 109) abgefangen wird!

Die Offset-Operatoren lösen `xBoundary` aus, wenn der Klient der Klasse auf Daten außerhalb des Arrays zuzugreifen versucht. Dieses Vorgehen ist der normalen Behandlung einer Anforderung von Array-Elementen – die einfach das zurückgibt, was gerade an der betreffenden Stelle im Speicher vorhanden ist, und damit das Programm fast sicher ins Nirwana schickt – weit überlegen.

In Zeile 100 leitet das Schlüsselwort `try` einen `try`-Block ein, der in Zeile 108 endet. Die Anweisungen im `try`-Block schreiben 100 Integer-Werte in das in Zeile 99 deklarierte Array.

Der in Zeile 109 deklarierte `catch`-Block fängt die `xBoundary`-Exceptions ab.

`try`-Blöcke

Ein `try`-Block ist eine Gruppe von Anweisungen, die mit dem Schlüsselwort `try` beginnt und in geschweifte Klammern eingefasst ist.

Zum Beispiel:

```
try
{
    Function();
};
```

`catch`-Blöcke

Ein `catch`-Block beginnt mit dem Schlüsselwort `catch`, dem ein Exception-Typ in Klammern folgt. Daran schließt sich der in geschweifte Klammern gefasste Anweisungsblock an.

Zum Beispiel:

```
try
{
    Function();
};
catch (OutOfMemory)
{
    // Aktion unternehmen
}
```

24.3 try-Blöcke und catch-Blöcke einsetzen

Da es nicht immer ohne weiteres klar ist, welche Aktionen eine Exception auslösen könnten, gehört die Suche nach den geeigneten Stellen für `try`-Blöcke mit zu den schwierigsten Aufgaben beim Einsatz von Exceptions. Als Nächstes steht die Frage, wo die Exception abzufangen ist. Zum Beispiel kann man alle Speicher-Exceptions dort auslösen, wo man den Speicher zuweist, und die Exceptions in einer höheren Ebene des Programms abfangen, wo die Routinen für die Benutzeroberfläche angesiedelt sind.

Kandidaten für `try`-Blöcke sind Routinen, die Speicher oder andere Ressourcen reservieren. Weiterhin sind Codebereiche ins Auge zu fassen, die Fehler bei Bereichsüberschreitungen, unzulässigen Eingaben oder ähnlichen Aktionen bewirken können.

24.3.1 Exceptions abfangen

Das Abfangen von Exceptions funktioniert folgendermaßen: Beim Eintreten eines Exception-Zustands wird der Aufruf-Stack untersucht. Der *Aufruf-Stack* verzeichnet die Funktionsaufrufe, wenn ein Programmabschnitt zu einer anderen Funktion verzweigt.

Über den Aufruf-Stack lässt sich der Aufrufpfad verfolgen. Wenn `main()` die Funktion `Animal::GetFavoriteFood()` aufruft, die Funktion `GetFavoriteFood()` ihrerseits in die Funktion `Animal::LookupPreferences()` verzweigt und diese Funktion schließlich `fstream::operator>>()` aufruft, stehen Informationen zu diesen Funktionen auf dem Aufruf-Stack. Eine rekursive Funktion kann auch mehrmals dort erscheinen.

Die Exception wird im Aufruf-Stack zum jeweils nächsten geschlossenen Block hinaufgereicht. Bei dieser Auflösung des Stacks werden die Destruktoren für lokale Objekte auf dem Stack aufgerufen und die Objekte abgebaut.

Nach jedem `try`-Block stehen eine oder mehrere `catch`-Anweisungen. Entspricht die Exception einer dieser `catch`-Anweisungen, wird die Exception mit der zugehörigen Anweisung behandelt. Stimmt die Exception mit keiner `catch`-Anweisung überein, setzt sich die Auflösung des Stacks fort.

Wenn die Exception den ganzen Weg bis zum Anfang des Programms (`main()`) zurückgelegt hat und immer noch nicht abgefangen wurde, wird die Funktion `terminate()`aufgerufen, die ihrerseits mit `abort()`das Programm abbricht.

Das Weiterreichen einer Exception lässt sich mit einer Einbahnstraße vergleichen. In ihrem Verlauf findet die Auflösung des Stacks und der Abbau der be-

treffenden Objekte statt. Es gibt kein Zurück: Nachdem eine Exception behandelt wurde, setzt das Programm nach dem `try`-Block mit der `catch`-Anweisung fort, die die Exception behandelt hat.

In Listing 24.1 springt das Programm demnach zu Zeile 112, der ersten Zeile nach dem `try`-Block mit der `catch`-Anweisung, die die `xBoundary`-Exception behandelt hat. Denken Sie daran, dass sich der Programmablauf beim Auslösen einer Exception nach dem `catch`-Block fortsetzt und nicht nach dem Punkt, an dem die Exception ausgelöst wurde.

24.3.2 Mehrere catch-Spezifikationen

Zu einer Exception können mehrere Bedingungen führen. In diesem Fall lassen sich die `catch`-Anweisungen nacheinander anordnen, wie man es von der `switch`-Anweisung kennt. Das Äquivalent der `default`-Anweisung ist die Anweisung »Fang alles ab«, die durch `catch(...)` gekennzeichnet ist.

Exceptions bei abgeleiteten Klassen

Achten Sie bei Konstruktionen mit mehreren Exceptions darauf, wohin die einzelnen `catch`-Anweisungen gehören. So ist es denkbar, dass sich bei zwei `catch`-Anweisungen die eine auf eine Basisklasse wie zum Beispiel `MyException` bezieht und die andere für eine von `MyException` abgeleitete Klasse gilt, aber spezifischer ist. Der Code durchläuft tatsächlich beide `catch`-Anweisungen. Manchmal ist dieses Verhalten erwünscht, aber nicht immer.

24.3.3 Abfangen als Referenz und Polymorphie

Man kann sich den Umstand zunutze machen, dass Exceptions einfach Klassen sind, die sich polymorph verwenden lassen. Übergibt man die Exception als Referenz, lässt sich gemäß der Vererbungshierarchie die geeignete Aktion entsprechend dem Laufzeittyp der Exception ausführen. Listing 24.2 demonstriert die polymorphe Verwendung von Exceptions.

Listing 24.2: polyexceptions.cpp – Polymorphe Exceptions

```
 1: // Listing 24.2 Exceptions polymorph abfangen
 2: #include <iostream>
 3:
 4: const int DefaultSize = 10;
 5:
 6: // Exception-Klassen definieren
 7: class xBoundary {};
 8:
 9: class xSize
10: {
11: public:
12:     xSize(int size):itsSize(size) {}
13:     ~xSize(){}
```

```
14:     virtual int GetSize() { return itsSize; }
15:     virtual void PrintError()
16:     { std::cout << "Groessenfehler. Empfangen: "
17:         << itsSize << std::endl; }
18: protected:
19:     int itsSize;
20: };
21:
22: class xTooBig : public xSize
23: {
24: public:
25:     xTooBig(int size):xSize(size){}
26:     virtual void PrintError()
27:     {
28:         std::cout << "Zu gross! Empfangen: ";
29:         std::cout << xSize::itsSize << std::endl;
30:     }
31: };
32:
33: class xTooSmall : public xSize
34: {
35: public:
36:     xTooSmall(int size):xSize(size){}
37:     virtual void PrintError()
38:     {
39:         std::cout << "Zu klein! Empfangen: ";
40:         std::cout << xSize::itsSize << std::endl;
41:     }
42: };
43:
44: class xZero  : public xTooSmall
45: {
46: public:
47:     xZero(int size):xTooSmall(size){}
48:     virtual void PrintError()
49:     {
50:         std::cout << "Null!!. Empfangen: ";
51:         std::cout << xSize::itsSize << std::endl;
52:     }
53: };
54:
55: class xNegative : public xSize
56: {
57: public:
58:     xNegative(int size):xSize(size){}
59:     virtual void PrintError()
60:     {
61:         std::cout << "Negativ! Empfangen: ";
62:         std::cout << xSize::itsSize << std::endl;
```

```
63:     }
64: };
65:
66: class Array
67: {
68: public:
69:     // Konstruktoren
70:     Array(int itsSize = DefaultSize);
71:     Array(const Array &rhs);
72:     ~Array() { delete [] pType;}
73:
74:     // Operatoren
75:     Array& operator=(const Array&);
76:     int& operator[](int offSet);
77:     const int& operator[](int offSet) const;
78:
79:     // Zugriffsfunktionen
80:     int GetitsSize() const { return itsSize; }
81:
82:     // Friend-Funktion
83:     friend std::ostream& operator<< (std::ostream&, const Array&);
84:
85:
86: private:
87:     int *pType;
88:     int  itsSize;
89: };
90:
91: Array::Array(int size):
92: itsSize(size)
93: {
94:     if (size == 0)
95:         throw xZero(size);
96:
97:     if (size < 0)
98:         throw xNegative(size);
99:
100:    if (size < 10)
101:        throw xTooSmall(size);
102:
103:    if (size > 30000)
104:        throw xTooBig(size);
105:
106:    pType = new int[size];
107:    for (int i = 0; i<size; i++)
108:        pType[i] = 0;
109: }
110:
```

```
111: int& Array::operator[] (int offset)
112: {
113:     int size = GetitsSize();
114:     if (offset >= 0 && offset < size)
115:         return pType[offset];
116:     throw xBoundary();
117:     return pType[offset];   // Tribut an MSC, Borland gibt
                                 // Warnung aus.
118: }
119:
120: const int& Array::operator[] (int offset) const
121: {
122:     int size = GetitsSize();
123:     if (offset >= 0 && offset < size)
124:         return pType[offset];
125:     throw xBoundary();
126:     return pType[offset];   // Tribut an MSC, Borland gibt
                                 // Warnung aus.
127: }
128:
129: int main()
130: {
131:     try
132:     {
133:         int choice;
134:         std::cout << "Bitte Array-Groesse eingeben: ";
135:         std::cin >> choice;
136:         Array intArray(choice);
137:         for (int j = 0; j< 100; j++)
138:         {
139:             intArray[j] = j;
140:             std::cout << "intArray[" << j << "] OK..."
141:                 << std::endl;
142:         }
143:     }
144:     catch (xBoundary)
145:     {
146:         std::cout << "Kann Ihre Eingabe nicht verarbeiten.\n";
147:     }
148:     catch (xSize& theException)
149:     {
150:         theException.PrintError();
151:     }
152:     catch (...)
153:     {
154:         std::cout << "Etwas ist schief gelaufen. "
155:             << "Keine Ahnung, was!" << std::endl;
156:     }
```

```
157:     std::cout << "Fertig.\n";
158:     return 0;
159: }
```

```
Bitte Array-Groesse eingeben: 5
Zu klein! Empfangen: 5
Fertig.

Bitte Array-Groesse eingeben: 50000
Zu gross! Empfangen: 50000
Fertig.

Bitte Array-Groesse eingeben: 12
intArray[0] OK...
intArray[1] OK...
intArray[2] OK...
intArray[3] OK...
intArray[4] OK...
intArray[5] OK...
intArray[6] OK...
intArray[7] OK...
intArray[8] OK...
intArray[9] OK...
intArray[10] OK...
intArray[11] OK...
Kann Ihre Eingabe nicht verarbeiten.
Fertig.
```

Listing 24.2 deklariert die virtuelle Methode `PrintError()`in der Klasse `xSize`. Diese Methode gibt eine Fehlermeldung und die aktuelle Größe der Klasse aus. Jede abgeleitete Klasse redefiniert diese Methode.

Zeile 148 deklariert das Exception-Objekt als Referenz. Bei Aufruf von `PrintError()` mit einer Referenz auf ein Objekt bewirkt die Polymorphie den Aufruf der korrekten Version von `PrintError()`. Beim ersten Durchlauf geben wir eine Array-Größe von 5 an. Das führt zur Auslösung der `TooSmall`-Exception. Auf Zeile 148 wird diese `xSize`-Exception abgefangen. Das zweite Mal geben wir ein Array mit 50000 Elementen an, was zur Auslösung der `TooBig`-Exception führt. Abgefangen wird diese Ausnahme ebenfalls in Zeile 148, wobei aber die Polymorphie für die Ausgabe der richtigen Fehlermeldung sorgt. Geben wir schließlich eine Array-Größe von 12 an, füllt das Programm das Array, bis es zur Auslösung der `xBoundary`-Exception (Bereichsüberschreitung) kommt, die in Zeile 144 abgefangen wird.

24.4 Wie geht es weiter?

Mit Templates und Exceptions beherrschen Sie nun auch kompliziertere Aspekte von C++. Bevor Sie das Buch weglegen, reißen wir noch einige Punkte zum Schreiben von professionellem Code an. Sobald Sie den Hobby-Bereich verlassen und als Mitglied eines Entwicklerteams arbeiten, muss Ihr Code nicht nur funktionieren, sondern auch von anderen verstanden werden. Der Code muss sich leicht warten und unterstützen lassen. Das trifft sowohl auf Sie zu, wenn Sie auf neue Kundenforderungen reagieren, als auch auf andere Mitarbeiter, nachdem Sie aus dem Projekt ausgeschieden sind.

Der Aneignung eines einheitlichen Stils kommt besondere Bedeutung zu, wenn es auch in vielerlei Hinsicht keine Rolle spielt, welchen Stil Sie wählen. Ein einheitlicher Stil lässt die Bedeutung bestimmter Codeabschnitte klar hervortreten. Beim Schreiben des Programms braucht man dann nicht ständig nachzusehen, ob man eine Funktion beim letzten Mal mit einem großen Anfangsbuchstaben geschrieben hat oder nicht.

Die folgenden Richtlinien sind willkürlich zusammengestellt. Sie basieren auf den Konzepten, die der Autor in eigenen Projekten realisiert hat und die sich als geeignet erwiesen haben. An diesen Richtlinien können Sie sich bei der Entwicklung eines eigenen Stils orientieren. Nachdem Sie sich auf einen Stil festgelegt haben, sollten Sie ihn strikt einhalten und so behandeln, als wäre er von den Programmiergöttern vorgegeben worden. Damit erleichtern Sie sich und Ihren Mitarbeitern die Arbeit im Team. Das heißt aber nicht, dass der Stil auf immer und ewig unverändert bleiben muss (im Laufe der Zeit werden Verbesserungen und neue Ideen einfließen) – sofern man immer konsequent einen einheitlichen Stil anwendet, vereinfacht sich die Zusammenarbeit.

24.4.1 Geschweifte Klammern

Die Ausrichtung der geschweiften Klammern ist eines der kontroversesten Themen unter C- und C++-Programmierern. Hierzu einige Vorschläge:

- Zueinander gehörende Klammern sollten untereinander stehen.
- Der äußerste Satz geschweifter Klammern in einer Definition oder Deklaration sollte am linken Rand stehen. Die zugehörigen Anweisungen rückt man ein. Alle anderen Sätze von geschweiften Klammern sollten linksbündig zur jeweils führenden Anweisung stehen.
- Bei wirklich langen Blöcken empfiehlt sich ein Kommentar nach der schließenden geschweiften Klammer, um den Zweck des Blocks zu kennzeichnen. Wenn Sie die schließende geschweifte Klammer sehen und nicht sicher sind, wo die öffnende Klammer steht, haben Sie einen »wirklich langen Block« vor sich. Zum Beispiel:

```
if (condition==true)
{
    // Hier stehen viele Codezeilen, einschließlich anderer Blöcke
    // Hier stehen viele Codezeilen, einschließlich anderer Blöcke
    // Hier stehen viele Codezeilen, einschließlich anderer Blöcke
}   // if (condition==true)
```

- Abgesehen von Inline-Funktionen sollte kein Code zusammen mit einer geschweiften Klammer auf derselben Zeile stehen. Zum Beispiel:

```
if (condition==true)
{
    j = k;
    SomeFunction();
}
m++;
```

24.4.2 Lange Zeilen

Schreiben Sie nur so viel auf eine Zeile, wie sich auf einem normalen Bildschirm anzeigen lässt. Code, der über den rechten Bildschirmrand hinausreicht, wird oft übersehen und der horizontale Bildlauf ist lästig. Teilt man eine Zeile, sollten die folgenden Zeilen mit einem Einzug beginnen. Nehmen Sie Zeilentrennungen an sinnvollen Stellen vor und belassen Sie einen dazwischen liegenden Operator am Ende der vorhergehenden Zeile (setzen Sie ihn nicht an den Beginn der folgenden Zeile). Damit ist klar, dass diese Zeile nicht allein steht und noch etwas folgt.

In C++ sind Funktionen oftmals zwar kürzer als in C. Trotzdem gilt nach wie vor der gute alte Rat: Halten Sie Funktionen so kurz wie möglich, damit die gesamte Funktion auf eine Seite passt.

Die Tabulatorgröße sollte drei oder vier Leerzeichen betragen. Vergewissern Sie sich, dass Ihr Editor jedes Tabulatorzeichen in die entsprechende Anzahl von Leerzeichen umwandelt.

24.4.3 switch-Anweisungen

Damit `switch`-Anweisungen die Breite einer Seite nicht überschreiten, können Sie sie wie folgt einrücken:

```
switch(variable)
{
case ValueOne:
    ActionOne();
    break;
case ValueTwo:
    ActionTwo();
    break;
```

```
default:
    assert("bad Action");
    break;
}
```

24.4.4 Programmtext

Leicht zu erfassender Code lässt sich auch leichter warten. Beherzigen Sie folgende Tipps:

- Gestalten Sie den Quelltext mit Whitespaces.
- Objekte und Arrays beziehen sich praktisch auf ein Ding. Verwenden Sie keine Leerzeichen innerhalb von Objektreferenzen (`.`, `->`, `[]`).
- Unäre Operatoren sind mit ihren Operanden verbunden. Schreiben Sie also kein Leerzeichen dazwischen. Sehen Sie ein Leerzeichen auf der dem Operanden abgewandten Seite des Operators vor. Zu den unären Operatoren gehören `!`, `~`, `++`, `-`, `*` (für Zeiger), `&` (Typumwandlungen) und `sizeof`.
- Binäre Operatoren sollten auf beiden Seiten ein Leerzeichen erhalten: `+`, `=`, `*`, `/`, `%`, `>>`, `<<`, `<`, `>`, `==`, `!=`, `&`, `|`, `&&`, `||`, `?:`, `=`, `+=` usw.
- Kennzeichnen Sie den Operatorvorrang nicht durch fehlende Leerzeichen (`4+ 3*2`).
- Setzen Sie ein Leerzeichen *nach* Kommas und Semikolons, nicht davor.
- Klammern sollten auf keiner Seite Leerzeichen erhalten.
- Schlüsselwörter wie `if` sollte man durch ein Leerzeichen absetzen: `if (a == b)`.
- Der eigentliche Kommentar sollte von den Zeichen // durch ein Leerzeichen getrennt sein.
- Setzen Sie das Kennzeichen für einen Zeiger oder eine Referenz neben den Typennamen und nicht neben den Variablennamen. Schreiben Sie also

  ```
  char* foo;
  int& theInt;
  ```

  und nicht

  ```
  char *foo;
  int &theInt;
  ```

- Deklarieren Sie nicht mehr als eine Variable auf derselben Zeile, sofern die Variablen nicht zusammengehören.

24.4.5 Namen von Bezeichnern

Für Bezeichner sollte man folgende Regeln einhalten:

- Namen von Bezeichnern sollten aussagekräftig und nicht zu kurz sein.
- Vermeiden Sie kryptische Abkürzungen.
- Nehmen Sie sich die Zeit, alles auszuschreiben.
- Kurze Namen (`i`, `p`, `x` usw.) sollte man nur verwenden, wenn sich durch die Kürze der Code besser lesen lässt und wo der Gebrauch so offensichtlich ist, dass man auf einen aussagekräftigen Namen verzichten kann.
- Die Länge eines Variablennamens sollte proportional zum Gültigkeitsbereich der Variablen sein.
- Aussehen und Klang von Bezeichnern sollten sich deutlich von anderen unterscheiden, um Verwechslungen zu vermeiden.
- Namen von Funktionen (oder Methoden) sind normalerweise Verben oder Kombinationen aus Verb und Substantiv: `Search()`, `Reset()`, `FindParagraph()`, `ShowCursor()`. Namen von Variablen sind gewöhnlich abstrakte Substantive, eventuell mit einem ergänzenden Substantiv: `count`, `state`, `windSpeed`, `windowHeight`. Boolesche Variablen sollten einen passenden Namen erhalten: `windowIconized`, `fileIsOpen`.

24.4.6 Schreibweise von Namen

Die Groß-/Kleinschreibung sollte man nicht übersehen, wenn man einen eigenen Stil entwickelt. Dazu folgende Hinweise:

- Bezeichner sollten einheitlich geschrieben sein – verwenden Sie möglichst die gemischte Schreibweise. Namen von Funktionen, Methoden, Klassen, `typedef` und `struct` sollten mit einem Großbuchstaben beginnen (beispielsweise `MyFunction`). Datenelemente oder lokale Variablen sollten mit einem Kleinbuchstaben beginnen (auch als »Kamelnotation« bezeichnet wie zum Beispiel `myVariable`).
- Aufzählungskonstanten sollten mit wenigen Kleinbuchstaben als Abkürzung für die Aufzählung (`enum`) beginnen. Zum Beispiel:

```
enum TextStyle
{
    tsPlain,
    tsBold,
    tsItalic,
    tsUnderscore,
};
```

24.4.7 Kommentare

Mit Kommentaren lässt sich ein Programm wesentlich verständlicher gestalten. Wenn man an einem Programm mehrere Tage oder sogar Monate arbeitet, kann man leicht vergessen, was bestimmte Codeabschnitte bewirken sollen oder warum man sie eingebunden hat. Probleme mit dem Verständnis des Codes können auch auftreten, wenn irgendjemand anderes Ihren Code liest. Einheitlich angewandte Kommentare in einem gut durchdachten Stil sind immer die Mühe wert. In Bezug auf Kommentare gibt es folgende Hinweise:

- Wo immer es möglich ist, sollten Sie die C++-Kommentare // und nicht die Form /* */ verwenden.
- Kommentare, die das Wesen eines Codeabschnitts beschreiben, sind unendlich wichtiger als Prozessdetails. Fügen Sie Aussagen von Wert hinzu und wiederholen Sie nicht einfach den Code. Verzichten Sie also auf Kommentare wie:

  ```
  n++; // n wird um eins inkrementiert
  ```

  Dieser Kommentar ist nicht einmal die Zeit wert, die er für die Eingabe braucht. Konzentrieren Sie sich auf die Semantik von Funktionen und Codeblöcken. Sagen Sie, was eine Funktion bewirkt. Kennzeichnen Sie Seiteneffekte, Parametertypen und Rückgabewerte. Beschreiben Sie alle Annahmen, die Sie treffen (oder nicht treffen), wie etwa »angenommen n ist nicht negativ« oder »liefert –1, wenn x ungültig ist«. In komplizierten Logikabschnitten kennzeichnen Sie mit Kommentaren die Zustände, die an diesem Punkt im Code existieren.
- Schreiben Sie vollständige Sätze mit korrekten Satzzeichen und mit Groß-/Kleinschreibung. Die zusätzlichen Eingaben lohnen sich. Seien Sie nicht zu kryptisch und kürzen Sie nicht allzu sehr ab. Was beim Schreiben des Codes noch klar auf der Hand zu liegen scheint, kann in wenigen Monaten schon völlig im Dunklen liegen.
- Verwenden Sie Leerzeilen, um dem Leser die Abläufe verständlich zu machen. Trennen Sie Anweisungen in logische Gruppen.

24.4.8 Zugriff

Die Zugriffsmethoden in Ihrem Programm sollten ebenfalls einheitlich sein. Einige Tipps für den Zugriff:

- Verwenden Sie immer die Bezeichner `public:`, `private:`, und `protected:`. Verlassen Sie sich nicht auf Standardwerte.
- Führen Sie zuerst die öffentlichen Elemente, dann die geschützten und erst danach die privaten auf. Gruppieren Sie die Datenelemente nach den Methoden.

- Schreiben Sie den/die Konstruktor(en) gefolgt vom Destruktor in den passenden Codeabschnitt. Führen Sie überladene Methoden mit demselben Namen unmittelbar nacheinander auf. Fassen Sie Zugriffsfunktionen wenn möglich zu einer Gruppe zusammen.
- Ordnen Sie die Methodennamen innerhalb jeder Gruppe und die Variablennamen alphabetisch. Dateinamen in `include`-Anweisungen sollten Sie ebenfalls nach dem Alphabet ordnen.
- Selbst wenn das Schlüsselwort `virtual` beim Redefinieren optional ist, verwenden Sie es trotzdem. Es erinnert Sie daran, dass das Element virtuell ist und führt zu einer einheitlichen Deklaration.

24.4.9 Klassendefinitionen

Geben Sie die Definitionen von Methoden möglichst in derselben Reihenfolge wie die Deklarationen an. Man findet sie dann leichter.

Bei der Definition einer Funktion setzen Sie den Rückgabetyp und alle anderen Modifizierer auf eine vorhergehende Zeile, damit der Klassenname und der Funktionsname am linken Rand beginnen. So lassen sich Funktionen leichter auffinden.

24.4.10 include-Dateien

Binden Sie nach Möglichkeit keine anderen Dateien in Header-Dateien ein. Im Idealfall geben Sie lediglich die Header-Datei für die hier abgeleitete Klasse an. Zwingend erforderlich sind auch `include`-Anweisungen für Objekte, die Elemente der deklarierten Klasse sind.

Lassen Sie keine `include`-Datei in einem Header weg, nur weil Sie annehmen, dass irgendeine `.cpp`-Datei die benötigte `include`-Datei einbindet.

Mehrfachdeklarationen

Alle Header-Dateien sollten mit Schutzmechanismen gegen Mehrfachdeklaration ausgestattet sein.

24.4.11 assert()-Anweisungen

Verwenden Sie `assert()`-Anweisungen großzügig. Diese Anweisungen helfen Ihnen bei der Fehlersuche, zeigen dem Leser aber auch deutlich, von welchen Annahmen auszugehen ist. Außerdem denkt man beim Schreiben des Codes genauer darüber nach, was gültig ist und was nicht.

24.4.12 const-Deklarationen

Verwenden Sie `const`, wo immer es angebracht ist: für Parameter, Variablen und Methoden. Häufig ist sowohl eine konstante als auch eine nicht konstante Version einer Methode erforderlich. Nehmen Sie das nicht als Entschuldigung eine auszulassen. Achten Sie genau auf Typumwandlungen von konstant in nicht-konstant und umgekehrt. Manchmal ist es die einzige Lösung. Prüfen Sie aber, ob es sinnvoll ist, und geben Sie einen Kommentar an.

24.5 Wie geht es weiter?

Nach harter Arbeit sind Sie nun ein kompetenter C++-Programmierer. Der Lernprozess hört aber nicht einfach an dieser Stelle auf. Auf Ihrem Weg vom Einsteiger in C++ zu einem Experten sollten Sie weitere Wissensquellen ausschöpfen.

Die folgenden Abschnitte bringen einige Quellenhinweise. Diese Empfehlungen spiegeln die persönliche Erfahrung und Meinung des Autors wider. Zu allen Themen gibt es Dutzende Werke. Informieren Sie sich eingehend, bevor Sie sich zum Kauf entschließen.

24.5.1 Hilfe und Rat

Als Allererstes sollten Sie als C++-Programmierer in die eine oder andere C++-Diskussionsrunde in einem Online-Dienst einsteigen. Diese Gruppen gestatten direkten Kontakt mit Hunderten oder Tausenden von C++-Programmierern, die Ihre Fragen beantworten, Rat bieten und ein Sprachrohr für Ihre Ideen darstellen. Es gibt auch E-Mail-Diskussionsgruppen, Webbasierte Diskussions-Boards und die unschätzbaren Newsgroups (wie zum Beispiel `comp.std.C++` u.a.).

24.5.2 Literatur

Zu C/C++ gibt es ein fast nicht zu überblickendes Literaturangebot. Wir empfehlen zum Beispiel folgende Bücher:

- *Effektiv C++ programmieren, 50 Möglichkeiten zur Verbesserung Ihrer Programme* von Scott Meyers (Addison-Wesley, ISBN: 3-8273-13058, dritte Auflage, 1997)
- *C++ Unleashed* von Mickey Williams et al. (Sams, ISBN: 0672312417, 1999). Stellt kompliziertere C++-Techniken vor.

- *Clouds To Code* von Jesse Liberty (Wrox Press Ltd, ISBN: 1-8610-00952, 1997). Das Buch zeigt, wie man eine praxisgerechte Anwendung mit C++ mithilfe objektorientierter Analyse und Entwurf erstellt.
- *Teach Yourself C++ for LINUX in 21 Days* von Jesse Liberty et al. (Sams, ISBN 0672318954, 2000). Zeigt mehr zu C++ und der Programmierung in der Linux-Umgebung.

Darüber hinaus empfehlen wir für Fortgeschrittene:

- *Die C++-Programmiersprache. Deutsche Übersetzung der Special Edition* von Bjarne Stroustrup (Addison-Wesley, ISBN: 3-8273-1660X, 2000)
- *The Elements of Programming Style* von Brian W. Kernighan und P. J. Plauger (McGraw-Hill, ISBN: 0070342075, 1988)
- *Praktische C++ Programmierung* von Steve Oualline (O'Reilly, ISBN: 3897213583, 2003)
- *The Art of Computer Programming* (3 Bände): *Fundamental Algorithms*, *Seminumerical Algorithms* und *Sorting and Searching* von Donald E. Knuth (Addison-Wesley, ISBN: 0201485419, 1998)
- *Programmierpraxis, Prinzipien zur effizienten Programmierung* von Brian W. Kernighan und Rob Pike (Addison-Wesley, 3827315832, 1999)
- *Programmierwerkzeuge* von Brian W. Kernighan und P. J. Plauger (Springer, ISBN: 3540104194, 1980)
- *C/C++ Users Journal* (monatlich erscheinend) *http://www.cuj.com*

Und wenn Sie mehr darüber wissen wollen, wie Sie Ihre Kenntnisse anwenden, um einen interessanten Job zu bekommen, sollten Sie sich folgendes Buch ansehen:

- *The Complete Idiot's Guide to a Career In Computer Programming* von Jesse Liberty (Que, ISBN: 0789719959, 1999)

24.6 Fragen und Antworten

F: Warum verwendet man `try`-Blöcke, wenn man auch auf die allgemeine Exception-Behandlung zurückgreifen kann?

A: Mit einem `try`-Block kann man eine für den jeweiligen Codeabschnitt spezifische Exception-Behandlung realisieren. Die allgemeine Behandlung gilt für den gesamten Code, die spezifische Routine für den Code innerhalb des `try`-Blocks.

F: Wie geht es weiter, nachdem ich nun das Ende dieses Buches erreicht habe?

A: Lernen, lernen, nochmals lernen! Probieren Sie Beispiele aus, schreiben Sie Code und informieren Sie sich in anderen Quellen, die online und in gedruckter Form verfügbar sind.

24.7 Workshop

Nachdem Sie alle 24 Stunden abgeschlossen haben, können Sie einige letzte Fragen beantworten und mit einer Übung abschließen, um Ihre Kenntnisse zu C++ zu festigen.

24.7.1 Kontrollfragen

1. Welchen Zweck erfüllt ein `try`-Block?
2. Wie definiert man, dass eine Exception ausgelöst werden soll?
3. Wie unterscheiden sich `throw` und `try`?
4. Wie deklarieren Sie einen `catch`-Standardblock?

24.7.2 Übung

Dieses Kapitel beschränkt sich auf eine einzige Übung: Programmieren Sie drauflos! Probieren Sie die Programme in diesem Buch und die gegebenenfalls im Lieferumfang des Compilers enthaltenen Beispielprogramme aus.

24.7.3 Antworten zu den Kontrollfragen

1. Der `try`-Block enthält einen Satz von Anweisungen, für die die Exceptions im zugehörigen `catch`-Block behandelt werden.
2. Eigene Exceptions können Sie auslösen, indem Sie eine Klasse erstellen und diesen Klassennamen in einer `throw`-Anweisung angeben. Sehen Sie eine korrespondierende `catch`-Anweisung vor, um diese Exception zu behandeln.
3. Mit `throw` lösen Sie eine Exception explizit – d.h. im Programm selbst – aus (»Houston, wir haben ein Problem«). Ein `try`-Block sagt dem Compiler, dass Sie alle Exceptions, die in diesem Block auftreten (so auch die von Bibliotheksroutinen, vom Betriebssystem oder auch von `throw`), im korrespondierenden `catch`-Block behandeln möchten.

4. Um einen `catch`-Block zu definieren, der alle noch nicht in einem anderen `catch`-Block genannten Exceptions auffängt (in der Art des `default`-Zweiges einer `switch`-Anweisung), schreiben Sie drei Punkte in der Argumentliste der `catch`-Anweisung, statt eines Exception-Namens: `catch(...)`.

Anhang A

Zahlensysteme

Mathematische Kenntnisse sind im Computer-Bereich unverzichtbar. In diesem Anhang beschäftigen wir uns mit der Darstellung von Zahlen in den verschiedenen Zahlensystemen.

Die Zahl 145 verbinden wir ohne großes Nachdenken mit der Vorstellung von »einhundertfünfundvierzig«. Um die Zahlensysteme zu verstehen, mit denen der Computer arbeitet, müssen wir die Zahl 145 unter einem anderen Blickwinkel untersuchen und sie nicht als Zahl, sondern als Code für eine Zahl betrachten.

Beispielsweise ist das Zahlzeichen »3« nur ein Symbol, das wir gedanklich in die abstrakte Vorstellung der Zahl drei umsetzen. Mit dem Zahlzeichen stellt man die Zahl dar. Das Zahlzeichen »3« ist aber nicht die einzige Darstellungsform, aus der wir den Begriff *drei* abstrahieren. Man kann diese Zahl auch in der Form »drei«, »3«, »III« oder »***« ausdrücken.

Im Dezimalsystem (zur Basis 10) lassen sich alle Zahlen mit den Zahlzeichen 0, 1, 2, 3, 4, 5, 6, 7, 8 und 9 schreiben. Wie wird nun die Zahl 10 dargestellt?

Nehmen wir zum Beispiel eine Strategie an, in der wir die Zahl 10 durch den Buchstaben A ausdrücken oder als IIIIIIIIII schreiben. Die alten Römer verwendeten das Zeichen X. In dem bei uns gebräuchlichen arabischen Zahlensystem benutzen wir eine Kombination aus Zahlzeichen und dessen Position in der Zahl zur Darstellung von Werten. Die erste (ganz rechts stehende) Stelle ist für die »Einer« vorgesehen, die nächste für die Zehner. Die Zahl fünfzehn wird demnach als 15 geschrieben (zu lesen als »eins, fünf«). Das heißt: ein Zehner und 5 Einer.

Es lassen sich bestimmte Regeln aufstellen, die man verallgemeinern kann:

1. Das Zahlensystem zur Basis 10 verwendet die Ziffern 0 bis 9.
2. Die Stellen sind Potenzen von zehn: Einer, Zehner, Hunderter usw.
3. Bedeutet die dritte Stelle 100, lässt sich mit zwei Stellen der Wert 99 als größte Zahl darstellen. Allgemeiner formuliert: Bei *n* Stellen kann man die Werte 0 bis $(10^n - 1)$ ausdrücken. Mit drei Stellen lassen sich demnach die Werte 0 bis $(10^3 - 1)$ oder 0 bis 999 darstellen.

A.1 Andere Basen

Die Basis 10 verwenden wir nicht aus purem Zufall – wir haben 10 Finger. Allerdings sind auch andere Basen möglich. Die zur Basis 10 aufgestellten Regeln können wir für die Beschreibung der Basis 8 wie folgt abändern:

1. Das Zahlensystem zur Basis 8 verwendet die Ziffern 0 bis 7.
2. Die Stellen sind Potenzen von 8: Einer, Achter, Vierundsechziger usw.
3. Mit *n* Stellen kann man 0 bis $(8^n - 1)$ Werte darstellen.

Zur Unterscheidung der Zahlen in den jeweiligen Basen schreibt man die Basis als Index neben die Zahl. Die Zahl fünfzehn zur Basis 10 wird dann als 15_{10} geschrieben und als »eins, fünf, Basis zehn« gelesen.

Die Zahl 15_{10} schreibt man im Zahlensystem zur Basis 8 als 17_8 und liest »eins, sieben, Basis acht«. Man kann auch »fünfzehn« sagen, da sich am eigentlichen Wert der Zahl nichts geändert hat.

Warum 17? Die 1 bedeutet 1 Achter, und die 7 steht für 7 Einer. Ein Achter plus sieben Einer ergibt fünfzehn.

Schreiben wir diese Zahl in Form von fünfzehn Sternchen:

```
*****     *****
*****
```

Normalerweise bildet man zwei Gruppen – eine Gruppe mit zehn Sternchen und eine weitere mit fünf. Als Dezimalzahl wird das als 15 (1 Zehner und 5 Einer) dargestellt. Die Sternchen kann man auch wie folgt anordnen:

```
****      *******
****
```

Das heißt, acht Sternchen und sieben. Zur Basis 8 schreibt man das als 17_8 (ein Achter und sieben Einer).

A.2 Betrachtungen zu Basen

Die Zahl fünfzehn kann man zur Basis zehn als 15_{10}, zur Basis neun als 16_9, zur Basis 8 als 17_8 und zur Basis 7 als 21_7 darstellen. Warum 21_7? In der Zahlenbasis 7 gibt es kein Zahlzeichen 8. Um den Wert fünfzehn auszudrücken, braucht man zwei Siebener und einen Einer.

Wie kann man dieses Verfahren verallgemeinern? Um eine Zahl aus der Basis zehn in die Basis 7 umzuwandeln, geht man von den einzelnen Stellen aus: In der Basis 7 gibt es Einer, Siebener, Neunundvierziger, Dreihundertdreiundvierziger usw. Diese Stellenwerte ergeben sich aus 7^0, 7^1, 7^2, 7^3 usw. Bauen Sie sich eine Tabelle auf:

4	3	2	1
7^3	7^2	7^1	7^0
343	49	7	1

Zahlen zur 0-ten Potenz

In der Mathematik besagt eine der Potenzregeln, dass jeder beliebige Wert zur Potenz 0 das Ergebnis 1 hat. So ist $7^0 = 1$, genau wie 10^0 oder $1.217.549.343^0$ gleich 1 ist.

In der ersten Zeile steht die Stellennummer. Die zweite Zeile gibt die Potenzen zu 7 wieder. In der dritten Zeile finden Sie die dazugehörenden Werte als Dezimalzahl.

Um einen dezimalen Wert in die Basis 7 zu konvertieren, geht man folgendermaßen vor: Zunächst ermittelt man für die umzuwandelnde Zahl die höchstwertige Stelle (das heißt Spalte in der obigen Tabelle). Ist zum Beispiel 200 zu konvertieren, ist der Ergebniswert für Spalte 4 (343) gleich 0, da 343 nicht in 200 passt.

Als Nächstes bestimmt man, wie oft der Wert aus Spalte 3 (49) in 200 enthalten ist. Dazu dividiert man 200 durch 49. Das – ganzzahlige – Ergebnis lautet 4. Diesen Wert schreibt man in Stelle 3 der Zielzahl und arbeitet mit dem Rest der Division weiter: 4. Dieser Wert ist nicht durch 7 (Spalte 1 der Tabelle) teilbar, die Siebener-Stelle wird also zu 0. Der Wert 4 lässt sich mit 4 Einern darstellen. In die Einer-Stelle kommt demnach eine 4. Die vollständige Antwort lautet 404_7.

Um die Zahl 968_{10} in eine Zahl zur Basis 6 zu konvertieren, stellt man zunächst folgende Tabelle auf:

5	4	3	2	1
6^4	6^3	6^2	6^1	6^0
1296	216	36	6	1

Der Wert 1296 passt nicht in 968. Stelle 5 erhält also eine 0. Die Division von 968 durch 216 liefert 4 mit einem Rest von 104. Stelle 4 ist demnach 4. Dividiert man den Rest 104 durch 36, erhält man 2 Rest 32. Die Stelle 3 ist also 2. Bei Division von 32 durch 6 erhält man 5 mit einem Rest von 2. Die umgewandelte Zahl lautet schließlich 4252_6.

5	4	3	2	1
6^4	6^3	6^2	6^1	6^0
1296	216	36	6	1
0	4	2	5	2

Die Konvertierung von einer Basis in eine andere (zum Beispiel Basis 6 in Basis 10) lässt durch Multiplikationen wie folgt schneller durchführen:

```
4 * 216 = 864
2 * 36  =  72
5 * 6   =  30
2 * 1   =   2

Summe:    968
```

A.2.1 Binärsystem

Die Basis 2 ist die ultimative Erweiterung dieses Konzepts. Hier gibt es nur zwei Ziffern: 0 und 1. Die Stellen besitzen folgende Wertigkeiten:

Stelle:	8	7	6	5	4	3	2	1
Potenz:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Wert:	128	64	32	16	8	4	2	1

Um die Dezimalzahl 88 in eine Zahl zur Basis 2 umzuwandeln, verfährt man wie oben: Die Zahl ist nicht durch 128 teilbar, Stelle 8 ist daher 0. In 88 passt einmal 64, Stelle 7 wird zu 1. Es bleibt ein Rest von 24. Dieser ist nicht durch 32 teilbar, sodass Stelle 6 zu 0 wird. 16 passt einmal in 24, die Stelle 5 ist

demnach 1. Der Rest ist 8. Das entspricht genau dem Wert von Stelle 4, die somit eine 1 erhält. Da bei Division durch 8 kein Rest bleibt, sind die übrigen Stellen 0.

```
0 1 0 1 1 0 0 0
```

Zur Kontrolle konvertieren wir die Zahl zurück:

```
1 * 64 = 64
0 * 32 =  0
1 * 16 = 16
1 *  8 =  8
0 *  4 =  0
0 *  2 =  0
0 *  1 =  0

Summe:   88
```

A.2.2 Warum die Basis 2?

Zahlen zur Basis 2 lassen sich von einem Computer problemlos verarbeiten. Eigentlich wissen Computer überhaupt nicht, ob es sich um Buchstaben, Ziffern, Anweisungen oder Programme handelt. Der Hauptteil eines Computers besteht aus elektronischen Schaltungen, die nur zwei Zustände unterscheiden: entweder es fließt viel Strom oder sehr wenig.

Um die Logik einfach zu halten, verwendet man in der digitalen Schaltungstechnik keine relative Skala (wenig Strom, etwas Strom, mehr Strom, viel Strom, sehr viel Strom), sondern eine binäre Skala (»genug Strom« oder »nicht genug Strom«). Statt der Ausdrücke »genug« und »nicht genug« sagt man einfach »Ja« oder »Nein«. Die Werte Ja oder Nein bzw. `true` (wahr) oder `false` (falsch) lassen sich als 1 oder 0 darstellen. Es gilt meist die Zuordnung 1 für `true` oder Ja. Allerdings handelt es sich nur um eine Konvention. Man könnte genauso gut dem Wert 1 die Bedeutungen `false` oder Nein zuordnen.

Aus diesen Erläuterungen wird die Leistung des Binärsystems deutlich: Mit Einsen und Nullen kann man die logischen Funktionen jedes Schaltkreises darstellen (es fließt Strom oder keiner). Nur diese beiden Zustände hat der Computer zu unterscheiden.

A.2.3 Bits, Bytes und Nibbles

Die Darstellung der logischen Zustände mit Einsen und Nullen erfolgt durch so genannte Binärziffern (oder Bits). Die ersten Computer konnten 8 Bits gleichzeitig verarbeiten. Daher lag es nahe, den Code mithilfe von 8-Bit-Zahlen – den so genannten *Bytes* – zu schreiben.

Ein Halbbyte (4 Bit) bezeichnet man als *Nibble*.

Mit 8 Binärziffern lassen sich 256 verschiedene Werte darstellen. Sehen Sie sich dazu die Stellen an: Wenn alle 8 Bits gesetzt (1) sind, beträgt der Wert 255. Sind keine Bits gesetzt (alle gelöscht oder Null), ergibt sich der Wert 0. Der Bereich 0 bis 255 umfasst 256 mögliche Werte.

A.2.4 Was ist ein Kbyte?

Der Wert 2^{10} (1024) liegt in der Nähe von 10^3 (1000). Diese auffallende Übereinstimmung führte dazu, dass die Informatiker den Wert 2^{10} Bytes als 1 Kbyte oder 1 Kilobyte bezeichneten und sich damit an den Einheitenvorsatz von kilo für den Faktor 1000 anlehnten.

Analog dazu liegt 1024 * 1024 (1.048.576) nahe genug an einer Million, sodass man dafür 1 Mbyte oder 1 Megabyte schreibt. 1024 Mbyte sind dementsprechend 1 Gbyte.

A.2.5 Binärzahlen

Computer kodieren alle Vorgänge in Mustern aus Einsen und Nullen. Sämtliche Maschinenanweisungen sind in dieser Form verschlüsselt und werden durch Logikschaltkreise interpretiert. Prinzipiell lassen sich diese binär kodierten Anweisungen als Zahlen ausdrücken. Es wäre aber falsch, diesen Zahlen eine spezielle Bedeutung zuzuschreiben.

Beispielsweise interpretiert der x86-Chipsatz von Intel das Bitmuster 1001 0101 als Befehl. Man kann diese Bitfolge zwar in die Dezimalzahl 149 umwandeln, aber diese Zahl hat per se keine Bedeutung.

Manchmal sind diese Bitmuster Anweisungen, manchmal stellen sie Werte dar und ein andermal sind es Zeichencodes. In diesem Zusammenhang ist der standardisierte ASCII-Zeichensatz von Bedeutung. In ASCII (American Standard Code for Information Interchange – amerikanischer Standardcode für den Informationsaustausch) ist jedem Buchstaben und Satzzeichen eine binäre Darstellung aus 7 Bit zugeordnet. Beispielsweise wird der Kleinbuchstabe »a« durch 0110 0001 dargestellt. Das ist keine Zahl, obwohl man diese Bitfolge in die Zahl 97_{10} (64 + 32 + 1) umwandeln kann. In der Praxis sagt man, dass der Buchstabe »a« den ASCII-Wert 97 besitzt. Im engeren Sinne ist die binäre Darstellung von 97 (0110 0001) die Kodierung des Buchstabens »a« – bequemer ist es allerdings, den Dezimalwert 97 zu verwenden.

A.3 Hexadezimalsystem

Da Binärzahlen recht unübersichtlich sind, stellt man dieselben Werte in einer einfacheren Form dar. Die Umwandlung von Binärzahlen in das Dezimalsystem ist relativ umständlich. Dagegen lassen sich Zahlen zur Basis 2 sehr einfach in ein Zahlensystem zur Basis 16 konvertieren.

Dieses so genannte Hexadezimalsystem kennt 16 Zahlzeichen: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E und F. Den dezimalen Werten 10 bis 15 wurden willkürlich die Buchstaben A bis F zugeordnet. Die Stellen im Hexadezimalsystem lauten:

4	3	2	1
16^3	16^2	16^1	16^0
4096	256	16	1

Die Umwandlung vom hexadezimalen in das dezimale System erfolgt durch Multiplikation. Die Zahl $F8C_{16}$ stellt damit den folgenden dezimalen Wert dar:

```
F * 256 = 15 * 256 = 3840
8 * 16             =  128
C * 1    = 12 * 1  =   12

Summe:                3980
```

Die Umwandlung der Zahl FC_{16} in eine Binärzahl lässt sich am besten erledigen, indem man sie zuerst in das Dezimalsystem und von dort aus in das Binärsystem konvertiert:

```
F * 16 = 15 * 16 = 240
C * 1  = 12 *  1 =  12

Summe:             252
```

Die Umwandlung von 252_{10} in das Binärsystem nimmt man über die folgende Tabelle vor:

Stelle:	9	8	7	6	5	4	3	2	1
Potenz:	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Wert:	256	128	64	32	16	8	4	2	1

Die Zahl 252 lässt sich nicht durch 256 teilen.

252 dividiert durch 128 ergibt 1 Rest 124.

124 dividiert durch 64 ergibt 1 Rest 60.

60 dividiert durch 32 ergibt 1 Rest 28.

28 dividiert durch 16 ergibt 1 Rest 12.

12 dividiert durch 8 ergibt 1 Rest 4.

4 dividiert durch 4 ergibt 1 Rest 0.

Die restlichen Stellen sind damit 0 und als Ergebnis erhält man:

```
0 1 1 1 1 1 0 0
```

Die Lösung in der Binärdarstellung lautet also 1111 1100.

Gliedert man die Binärzahl in zwei Gruppen zu je 4 Ziffern, lässt sich die Zahl sehr einfach umwandeln.

Die rechte Gruppe lautet 1100. Der Dezimalwert ist 12, als Hexadezimalzahl ausgedrückt C.

Die linke Gruppe lautet 1111. Der Wert als Dezimalzahl ist 15 oder hexadezimal F.

Damit lässt sich schreiben:

```
1111 1100
F    C
```

Schreibt man beide Hexadezimalzahlen zusammen, erhält man FC, das den eigentlichen Wert von $1111\ 1100_2$ repräsentiert. Dieses Kurzverfahren funktioniert immer. Eine Binärzahl beliebiger Länge kann man in Gruppen zu je 4 Stellen einteilen, jede Gruppe einzeln in eine Hexadezimalzahl umwandeln und die sich ergebenden Ziffern hintereinander schreiben. Das Ergebnis ist die Hexadezimaldarstellung der Binärzahl.

Mit diesem Kurzverfahren lassen sich umgekehrt auch Hexadezimalzahlen in das Binärsystem konvertieren. Man teilt die Hexadezimalzahl in einzelne Ziffernstellen auf und wandelt jede von ihnen in vier Binärstellen (Bits) um. Merken Sie sich einfach, dass die ersten vier Bits die Wertigkeiten 8, 4, 2 und 1 haben. Dann können Sie leicht vom Hexadezimalsystem zum Binärsystem übergehen – jede Hexadezimalziffer lässt sich als selbstständige 4-Bit-Gruppe behandeln.

Das folgende Beispiel zeigt die Umwandlung einer längeren Zahl:

```
1011 0001 1101 0111
```

Die dezimale Wertigkeit der Stellen lautet von rechts nach links: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 und 32768.

```
1 *     1 =     1
1 *     2 =     2
1 *     4 =     4
0 *     8 =     0

1 *    16 =    16
0 *    32 =     0
1 *    64 =    64
1 *   128 =   128

1 *   256 =   256
0 *   512 =     0
0 *  1024 =     0
0 *  2048 =     0

1 *  4096 =  4096
1 *  8192 =  8192
0 * 16384 =     0
1 * 32768 = 32768

Summe:      45527
```

Die Umwandlung dieser Zahl in eine Hexadezimalzahl erfordert eine Tabelle mit den hexadezimalen Werten.

Stelle	5	4	3	2	1
Potenz	16^4	16^3	16^2	16^1	16^0
Wert	65535	4096	256	16	1

Die Zahl 45527 lässt sich nicht durch 65536 teilen. Als erste Stelle kommt demnach 4096 in Betracht. Der Wert 4096 passt 11 Mal (45056) in 45527, mit einem Rest von 471. Dieser Rest liefert bei Division durch 256 den Wert 1 und einen weiteren Rest von 215. Die Division von 215 durch 16 liefert 13 mit einem Rest von 7. Als Hexadezimalzahl ergibt sich B1D7.

Die Probe sieht folgendermaßen aus:

```
B * 4096 = 11 * 4096 = 45056
1 * 256                = 256
D * 16   = 13 * 16     = 208
7 * 1                    = 7

Summe: 45527
```

Beim abgekürzten Verfahren nimmt man die Binärzahl, 1011000111010111, und teilt sie in Gruppen zu je vier Stellen auf: 1011 0001 1101 0111. Dann rechnet man jede der vier Gruppen einzeln in eine Hexadezimalzahl um:

```
1011 =
1 * 1 =  1
1 * 2 =  2
0 * 4 =  0
1 * 8 =  8
Summe:  11
Hex:     B

0001 =
1 * 1 =  1
0 * 2 =  0
0 * 4 =  0
0 * 8 =  0
Summe:   1
Hex: 1

1101 =
1 * 1 =  1
0 * 2 =  0
1 * 4 =  4
1 * 8 =  8
Summe:  13
Hex: D

0111 =
1 * 1 =  1
1 * 2 =  2
1 * 4 =  4
0 * 8 =  0
Summe:   7
Hex: 7
```

Gesamtergebnis in Hex-Darstellung: `B1D7`

ANHANG B

Glossar

Dieser Anhang fasst die Schlüsselbegriffe zusammen, denen Sie das ganze Buch hindurch begegnen. Die Stichworte sind in der Reihenfolge ihres Auftretens nach Stunden geordnet aufgeführt. Wenn Sie auf einen unbekannten Begriff stoßen, können Sie zum entsprechenden Abschnitt gehen, um dort mehr zu erfahren.

Kapitel 1

Bibliothek: Eine Sammlung verknüpfungsfähiger Dateien, die zum Lieferumfang des Compilers gehören, die separat erworben werden oder die Sie selbst erstellen.

Funktion: Ein Codeblock, der eine bestimmte Aufgabe ausführt, beispielsweise das Addieren zweier Zahlen oder die Ausgabe auf dem Bildschirm.

Klasse: Die Definition eines neuen Typs. Klassen werden als Daten mit den zugehörigen Funktionen implementiert.

Compiler: Ein Programm, das den Quelltext eines (selbst geschriebenen) Programms in Maschinencode übersetzt und dabei eine Objektdatei erstellt, die sich mit anderen Objektdateien zu einem ausführbaren Programm linken (siehe **Linker**) lässt.

Linker: Ein Programm, das eine ausführbare Datei (ein ausführbares Programm) aus den vom Compiler erzeugten Objektcodedateien erstellt.

Objektorientierte Programmierung: Ein Programmierkonzept, das den nächsten Schritt nach prozeduraler und strukturierter Programmierung dar-

stellt. Die objektorientierte Programmierung nutzt das Verhalten von Objekten (das in Klassen definiert ist). Manchmal wird dieser Begriff auch werbewirksam für andere Arten von Software missbraucht.

ANSI (American National Standards Institute): Eine gemeinnützige Organisation mit der Aufgabe, die Standards in den Vereinigten Staaten zu pflegen. Die meisten Länder und viele Regionen (wie zum Beispiel die Europäische Union) haben ähnliche – zum Teil staatliche – Organisationen. Das ANSI ist keine Regierungsbehörde. Weitere Informationen finden Sie unter *http://www.ansi.org*.

ISO (International Organization for Standardization): Eine internationale nicht staatliche Standardisierungsbehörde ähnlich dem ANSI. Weitere Informationen finden Sie unter *http://www.iso.org*.

Kapitel 2

Kompilieren: Der erste Schritt beim Erstellen einer ausführbaren Datei. Der Compiler übersetzt den Quellcode in den so genannten Objektcode und schreibt ihn in eine Objektdatei (mit der Erweiterung `.obj`).

Linken: Der zweite Schritt beim Erstellen einer ausführbaren Datei. Der Linker verknüpft die von einem Compiler erstellten Objektdateien zu einem ausführbaren Programm.

Ausführbares Programm: Ein Programm, das sich unter dem jeweiligen Betriebssystem ausführen lässt.

Interpreter: Ein Programm, das den Quellcode während der Programmausführung interpretiert und dabei in den ausführbaren Maschinencode übersetzt.

Prozedurale Programmierung: Ein Programmierkonzept, bei dem eine Folge von Aktionen auf einer Menge von Daten ausgeführt wird.

Strukturierte Programmierung: Ein Programmierkonzept, das Programme in Prozeduren gliedert.

Kapselung: Das Erstellen in sich geschlossener Objekte.

Daten verbergen: Den Zustand einer Klasse in privaten Elementvariablen verbergen.

Vererbung: Einen neuen Typ erzeugen, der die Eigenschaften und Verhaltensweisen eines vorhandenen Typs erweitern kann.

Polymorphie: Die Fähigkeit, mehrere Untertypen so zu behandeln, als hätten sie denselben Basistyp.

Präprozessor: Ein Programm, das vor dem eigentlichen Compiler-Lauf ausgeführt wird und die Zeilen verarbeitet, die mit einem Nummernzeichen (#) beginnen.

Kommentare: Abschnitte im Quelltext, die sich nicht auf die Arbeitsweise des Programms auswirken, sondern als Erläuterung oder Dokumentation für den Programmierer gedacht sind.

Signatur: Der Name einer Funktion zusammen mit ihren Parametern (das heißt, der Funktionskopf ohne den Rückgabewert).

Kapitel 3

Variable: Eine benannte Stelle im Hauptspeicher des Computers, an der man einen Wert ablegen und später wieder abrufen kann.

RAM: Abkürzung für Random Access Memory – Speicher mit wahlfreiem Zugriff. Für den Arbeitsspeicher eines Computers verwendeter Speichertyp.

Typ: Die Größe und die Charakteristika eines Objekts.

vorzeichenbehaftet: Ein Variablentyp, der sowohl negative als auch positive Werte speichern kann.

vorzeichenlos: Ein Variablentyp, der nur positive Werte speichern kann.

ASCII (American Standard Code for Information Interchange): Ein System für die Kodierung von Buchstaben, Ziffern und Satzzeichen, das viele Computer als so genannten Zeichensatz verwenden.

Groß-/Kleinschreibung: Ein Compiler ist von der Groß-/Kleinschreibung abhängig, wenn er Klein- und Großbuchstaben in Bezeichnern als unterschiedliche Zeichen betrachtet (beispielsweise bezeichnet `myVal` nicht dasselbe wie `Myval`).

Typdefinition: Mit dem Schlüsselwort `typedef` gekennzeichnetes Synonym für einen vordefinierten (integrierten) Typ.

Konstanten: Datenspeicherstellen, deren Inhalt sich während der Programmausführung nicht ändert.

literale Konstanten: Ein Wert, der direkt (d.h. »buchstäblich«) im Quelltext des Programms eingetippt wird, beispielsweise `35`.

symbolische Konstanten: Ein typisierter und benannter Wert, der als Konstante gekennzeichnet ist, beispielsweise `SiedePunkt`.

Aufzählungskonstanten: Eine benannte Menge von Konstanten.

Kapitel 4

Anweisung: Ein Sprachkonstrukt, das die Reihenfolge der Ausführung steuert, einen Ausdruck auswertet oder auch nichts bewirkt (die `null`-Anweisung).

Whitespaces: Sammelbezeichnung für Leerzeichen, Tabulatoren und Zeichen für neue Zeile (Wagenrücklauf, Zeilenschaltung).

Verbundanweisung: Eine in geschweifte Klammern eingeschlossene Folge von mehreren Einzelanweisungen. An allen Stellen, wo eine einzelne Anweisung stehen kann, sind auch Verbundanweisungen zulässig.

Ausdruck: Jede Anweisung, die einen Wert zurückgibt.

Operator: Ein Symbol, das den Compiler zur Ausführung einer Aktion veranlasst.

Operand: Ein mathematischer Begriff, der den Teil eines Ausdrucks bezeichnet, auf den ein Operator wirkt.

Zuweisungsoperator (=): Bewirkt, dass der Wert des Operanden auf der linken Seite des Operators in den Wert auf der rechten Seite des Operators geändert wird.

L-Wert: Ein Operand, der auf der linken Seite eines Zuweisungsoperators zulässig ist.

R-Wert: Ein Operand, der auf der rechten Seite eines Zuweisungsoperators zulässig ist.

Vergleichsoperatoren (relationale Operatoren): Bestimmen, ob zwei Werte gleich sind oder ob der eine größer oder kleiner als der andere ist.

Inkrementieren: Einen Wert um 1 erhöhen (bezogen auf den Operator ++).

Dekrementieren: Einen Wert um 1 erniedrigen (bezogen auf den Operator --).

Präfix-Operator: Inkrementiert oder dekrementiert einen Wert, *bevor* der Ausdruck ausgewertet wird (beispielsweise `++myAge`).

Postfix-Operator: Inkrementiert oder dekrementiert einen Wert, *nachdem* der Ausdruck ausgewertet wurde (beispielsweise `myAge++`).

Rangfolge (Vorrang): Die Reihenfolge, nach der der Compiler die Operatoren eines Ausdrucks auswertet.

Kapitel 5

Stack: Von einem Programm eingerichteter spezieller Bereich im Hauptspeicher zur Aufnahme von Daten, die für die einzelnen Funktionen im Programm erforderlich sind. Der Stack arbeitet nach dem LIFO-Prinzip (Last In, First Out) einer Warteschlange: Das jeweils zuletzt in die Warteschlange (bzw. auf den Stack) gestellte Element wird zuerst wieder entnommen.

Funktionsdeklaration: Teilt dem Compiler den Namen, den Rückgabetyp und die Parameter der Funktion mit.

Prototyp: Die Deklaration einer Funktion.

Funktionsdefinition: Teilt dem Compiler die Arbeitsweise der Funktion mit. Die Definition enthält den eigentlichen Code – den Rumpf – der Funktion.

Parameterliste: Die Liste aller – jeweils durch Komma getrennten – Parameter und ihrer Typen.

Lokale Variablen: Variablen, die nur innerhalb einer Funktion existieren.

Gültigkeitsbereich: Bestimmt, wo eine Variable in einem Programm sichtbar ist und wo man auf sie zugreifen kann.

Globale Variablen: Variablen, die von jeder Stelle innerhalb eines Programms zugänglich sind.

Kapitel 6

Iteration: Die wiederholte Ausführung einer Aktion. Die Hauptmethode der Iteration ist die Schleife.

Endlosschleife: Eine Schleife, bei der die Abbruchbedingung nie erfüllt wird und die deshalb endlos läuft. Normalerweise ist der Programmierer darauf bedacht, solche Schleifen zu vermeiden.

Spagetti-Code: Programme, die in einem unübersichtlichen und schwer zu lesenden Format geschrieben und kaum strukturiert sind. Man vergleicht derartigen Programmcode mit einem Teller Spagetti, da sich die internen Abläufe kaum entwirren lassen.

Kapitel 7

Klienten: Andere Klassen oder Funktionen, die von der jeweiligen Klasse Gebrauch machen.

Elementvariablen (auch Datenelemente): Die Variablen einer Klasse.

Datenelemente: siehe **Elementvariablen**.

Elementfunktionen (auch Elementmethoden): Die Funktionen einer Klasse.

Objekt: Eine Instanz einer Klasse.

öffentlich (`public`): Mit dem Schlüsselwort `public` deklarierte Elemente sind allen Methoden aller Klassen zugänglich.

privat (`private`): Mit dem Schlüsselwort `private` deklarierte Elemente sind nur Methoden der Klasse selbst oder Methoden, die von dieser Klasse abgeleitet sind, zugänglich.

Zugriffsmethoden: Von einer Klasse bereitgestellte (öffentliche) Methoden, über die man auf private Elementvariablen zugreifen kann.

Methodendefinition: Eine Definition bestehend aus dem Namen der Klasse gefolgt von zwei Doppelpunkten, dem Namen der Methode und ihren Parametern.

Standardkonstruktor: Ein Konstruktor ohne Parameter.

Kapitel 8

Konstante Elementfunktion: Eine als konstant deklarierte Elementfunktion verspricht, dass sie nicht den Wert irgendeines Elements der Klasse ändert.

Schnittstelle (auch Interface, Klassenschnittstelle): Die Definition der Daten und Methoden, auf die andere Klassen und anderer Code zugreifen können. Die Schnittstelle ist nicht der eigentliche Code der Klasse, sondern sagt nur aus, wie der Code verwendet wird. Diese Informationen speichert man normalerweise in einer Header-Datei und bindet diese in das Modul ein, das die Klasse verwendet.

Implementierung (auch Klassenimplementierung): Der Code und die Deklarationen der Daten innerhalb einer Klasse. Dieser Code ist über die Schnittstelle zugänglich. Die Implementierung wird normalerweise in einer `.cpp`-Datei gespeichert und in das Objekt- oder Bibliotheksformat kompiliert. In vielen Fällen wird die Schnittstelle (in Form einer Header-Datei) und der kompilierte Code (Objekt- oder Bibliotheksdatei) aber nicht der eigentliche Klassencode bereitgestellt, was Änderungen am Klassencode verhindert.

Kapitel 9

Zeiger: Eine Variable, die eine Speicheradresse aufnimmt.

Umleitung (auch Indirektion): Zugriff auf den Wert an der im Zeiger gespeicherten Adresse.

Heap: Der noch freie Speicher, nachdem der Codebereich, der globale Namensraum und der Stack zugewiesen wurden. Den Heap bezeichnet man auch als »Freispeicher«, da er die Quelle für den mit `new` oder `malloc` dynamisch reservierten Speicher bildet.

Kapitel 10

wilde Zeiger: Bezeichnung für einen Zeiger, auf dem man `delete` aufgerufen hat und über den man dann versucht, auf den freigegebenen Speicher zuzugreifen. Dieser häufig auftretende Fehler ist schwer einzukreisen, da die fehlerauslösende Aktion (der ungeeignete Speicherzugriff) in der Regel erst viel später nach dem `delete` stattfindet.

Kapitel 11

Referenz: Ein Alias für ein Objekt.

Kapitel 12

Effizienz: Ein überholtes Thema, bei dem man größere Änderungen am Programmcode vornimmt, um winzige Zeiteinsparungen zu erreichen. Man kann zwar bereits beim Schreiben des Programms die Ausführungsgeschwindigkeit im Hinterkopf behalten, doch sofern es sich nicht um Echtzeit-Systeme handelt (die beispielsweise medizinische Apparaturen oder Marschflugkörper steuern), sollte man es dem Compiler überlassen, sich um die Details von effizientem Code zu kümmern.

Kapitel 13

flache Kopie: Kopiert die genauen Werte der Elementvariablen des einen Objekts in ein anderes Objekt. Auch als **elementweise Kopie** bezeichnet.

tiefe Kopie: Kopiert die Werte von Elementvariablen und erzeugt Kopien der Objekte, auf die Elementzeiger verweisen.

Kapitel 14

unärer Operator: Ein Operator, der nur auf einen Term wirkt, beispielsweise `a++`.

binärer Operator: Ein Operator, der zwei Terme verknüpft, beispielsweise `a + b`.

ternärer Operator: Ein Operator, der drei Terme einbezieht. In C++ gibt es einen ternären Operator (`? :`), den man wie folgt verwendet:

```
a < b ? true : false;
```

Dieser Ausdruck liefert `true`, wenn `a` kleiner als `b` ist, und andernfalls `false`.

Kapitel 15

Array (Datenfeld): Eine Sammlung von Objekten, die alle den gleichen Typ haben.

Index: Der Abstand zum ersten Element eines Arrays. Die Zählung der Elemente beginnt in C++ immer bei `0`, sodass man beispielsweise mit `myArray[3]` auf das vierte Element des Arrays `myArray` zugreift.

String: Ein Array mit Elementen vom Typ `char`, das mit einem Null-Zeichen (geschrieben als `'\0'`) endet.

Kapitel 16

Stub-Routine: Eine Funktion oder Methode, die nur so viel Code enthält, dass sie sich kompilieren lässt. Die eigentlichen Implementierungsdetails werden später ergänzt.

Überschreiben (Redefinieren): In der abgeleiteten Klasse eine Methode mit demselben Rückgabetyp und derselben Signatur wie eine Elementfunktion in der Basisklasse erstellen, aber mit einer neuen Implementierung versehen. Die redefinierte Methode muss bezüglich Rückgabetyp und Signatur mit der Methode in der Basisklasse übereinstimmen.

Kapitel 17

virtuelle Methode: Eines der Instrumente, mit denen C++ die **Polymorphie** implementiert. Damit lassen sich abgeleitete Objekte so behandeln, als wären sie Objekte der Basisklasse.

V-Tabelle: Der interne Mechanismus, der die virtuellen Funktionen verfolgt, die in den einzelnen Objekten erzeugt werden.

Kapitel 18

abstrakter Datentyp (ADT): Repräsentiert ein Konzept (wie Form, `Shape`) und nicht ein Objekt (wie Kreis, `Circle`). Wie der Name verrät, handelt es sich um eine abstrakte Form.

abstrakte Funktion: Eine virtuelle Funktion, die in der abgeleiteten Klasse überschrieben werden muss, weil ihr der eigentliche Code fehlt. Eine abstrakte Funktion hat keine Implementierung in der Basisklasse und heißt deshalb auch **rein virtuelle Funktion**.

Kapitel 19

verkettete Liste: Eine Datenstruktur, die aus miteinander verketteten Knoten besteht.

einfach verkettete Liste: Eine verkettete Liste, bei der jeder Knoten auf den jeweils nächsten Knoten verweist, jedoch nicht auf den vorhergehenden.

doppelt verkettete Liste: Eine verkettete Liste, bei der jeder Knoten sowohl auf den nächsten als auch den vorhergehenden Knoten in der Liste verweist.

Baum: Eine komplexe Datenstruktur, die aus Knoten aufgebaut ist, wobei jeder Knoten auf zwei oder mehr andere Knoten verweist.

Kapitel 20

Friend: Schlüsselwort, um einer anderen Klasse den Zugriff auf die privaten Elementvariablen und Methoden der aktuellen Klasse zu gewähren.

statische Datenelemente: Im Unterschied zu den meisten Datenelementen innerhalb einer Klasse werden statische Datenelemente nicht für jedes erzeugte Objekt repliziert. Es existiert jeweils nur ein Exemplar dieser Elemente und alle Objekte dieser Klasse können darauf zugreifen. Mit statischen Datenelementen verfolgt man beispielsweise die Anzahl der erzeugten Objekte oder etwas anderes, was für alle Objekte dieser Klasse relevant ist.

statische Elementfunktionen: Diese Funktionen existieren analog zu statischen Datenelementen im Gültigkeitsbereich der Klasse – nicht in einzelnen Objekten – und lassen sich aufrufen, ohne ein konkretes Objekt zu referenzieren.

Kapitel 21

#define: Ein Befehl, der eine String-Ersetzung definiert.

Token: Eine Zeichenfolge, die sich überall dort einsetzen lässt, wo man einen String, eine Konstante oder eine andere Buchstabengruppe verwenden könnte.

Kapitel 22

Simulation: Ein Computer-Modell für einen Teil eines realen Systems.

konzeptionelle Phase: Das Kernkonzept des Softwareprojekts.

Anwendungsfall: Eine Beschreibung wie das System verwendet wird.

UML (Unified Modeling Language): Eine standardisierte, grafisch orientierte Beschreibungssprache für objektorientierte Modelle (Anforderungen, Entwurf).

Problemraum: Die Menge der Probleme und Aufgaben, die das Programm zu lösen versucht.

Lösungsraum: Die Menge der möglichen Lösungen für das Problem.

Rahmenprogramm: Ein Testprogramm.

Kapitel 23

Template: Ein Instrument, um eine allgemeine Klasse oder Methode zu erzeugen und Typen als Parameter zu übergeben.

Instanzierung: Ein Objekt von einer Klasse oder einen Typ von einer Template erzeugen.

Kapitel 24

Exception (Ausnahme): Ein Objekt, das von dem Codebereich, in dem ein Problem auftritt, zum dem Codebereich, der das Problem behandelt, übergeben wird.

Stichwortverzeichnis

-- (Dekrement) 81
- (Subtraktion) 79
 48, 87
! (NOT-Operator) 95
!= (Ungleich) 87
(Präprozessor) 46, 427
(Zeichenkettenbildung) 436
(Verkettung) 436
#define 71, 428, 552
– Konstanten 428
– Makros 433
– Testen 429
– Vergleich zu const 428
#endif 429
#ifdef 429
#ifndef 429
#include 46, 427
#undef 432
% (Modulo) 79
%= (Modulo mit Zuweisung) 81
& (Adressoperator) 184
– Zeiger 188
& (Referenz) 218
&& (AND-Operator) 95
(Index) 292
(Semikolon) 106
* (Indirektion) 189
* (Multiplikation) 79, 190
* (Zeiger) 188
*= (Multiplikation mit Zuweisung) 81
+ (Addition) 79
++ (Inkrement) 81
+= (Addition mit Zuweisung) 81
/ (Division) 79
/* (Kommentare) 49
// (Kommentare) 49
/= (Division mit Zuweisung) 81
-= (Subtraktion mit Zuweisung) 81
= (Zuweisung) 77, 276
== (Gleich) 87
-> (Elementverweis) 205
> (Größer als) 87
>= (Größer oder gleich) 87
| | (OR-Operator) 95
~ (Destruktor) 161
80/80-Regel 474

A
Ableitung
– ADTs von anderen ADTs 367
– Syntax 314
– Vererbung 312
abort() 517
Add() 53, 273
Addition (+) 79
Adressen 57, 183
– mit Zeigern manipulieren 192
– Zeiger 187, 190

Adressoperator 184
– Referenzen 219
ADT (abstrakter Datentyp) 359, 460, 551
Alarmsysteme
– Analyse und Anforderungen der Simulation 459
– Condition-Objekte simulieren 462
– Ereignisschleifen simulieren 463
– konzeptionelle Phase der Simulation 459
– Objekte simulieren 461
– simulieren 458
Alias (typedef) 66
Analysephase
– Programmentwurf 459
AND 95
Anführungszeichen
– Zeichenkettenbildung 436
ANSI C++ 24
ANSI (American National Standards Institute) 25, 544
Anweisungen 75, 546
– #define 428
– #else 429
– #endif 429
– #ifdef 429
– #ifndef 429
– #include 427
– Ausdrücke 77
– break 131
– continue 131
– default 145
– else 88
– Funktionen 107
– goto 127
– if 88
– Label 127
– Präprozessor 46
– return 106
– switch 145
– zusammengesetzte 76
– Zuweisungen 75
Anwendungsfälle 459, 552
API (Application Programming Interface) 475
Architektur
– Prototypen 474
AreaCube() 118
Argumente
– als Referenz übergeben 224
– Funktionen 52, 110
– Klammern in Makros 434
– Konstruktoren der Basisklasse 321
– Standardwerte 116
– Übergabe als Wert 112
– Zeiger auf Funktionen 412
Arrays 287, 550
– Anzahl der Elemente 291
– Bereichsüberschreitung 289
– char 303
– deklarieren 287
– Dimensionen 293
– Elemente 288
– Elemente gruppieren 295
– Fehlersuche 289
– Größe 291
– Heap 297, 299
– Index 287
– initialisieren 291
– löschen auf Heap 302
– mehrdimensionale 293
– mehrdimensionale initialisieren 295
– Namen 300
– Objekte 292
– Offset 288
– Platzbedarf 294
– Speicher 296
– Standardkonstruktoren 292
– Strings 303
– Zeiger 297
– Zeiger auf 300
– Zeiger auf Elementfunktionen 421
– Zeiger auf Funktionen 409
– Zeigerarithmetik 300
– Zugriff 288
ASCII 61, 545
– American Standard Code for Information Interchange 538
assert() 438, 528
– Ebenen der Fehlersuche 448
– Fehlersuche 440
– Seiteneffekte 440
– Zwischenwerte ausgeben 447
Aufrufen
– Destruktoren 318
– Konstruktoren 318
– Methoden 330
Aufruf-Stack 517
Aufzählungskonstanten 71, 545
Ausdrücke 77, 546
– Klammern verschachteln 85

– Testergebnis 88
– true vs. false 97
Ausführung
– Programme 50
Ausgaben
– Bildschirm 48
– Zwischenwerte 447
Ausnahmen 511, 552
Autoren
– Adressen 530

B
Basis
– Zahlensysteme 534
Basisklassen 312
Bäume 551
– verkettete Listen 376
Bearbeitung 467
Bedingungen
– testen 88, 129
– unveränderliche 441
Befehlszeile
– Definitionen 432
– Fehlersuche aktivieren 439
Begleit-CD 22
Begrenzer
– Strings 305
Begriffe 543
– #define 552
– abgeleiteter Typ 45
– abstrakte Funktion 360, 551
– abstrakter Datentyp 359, 551
– ANSI 544
– ANSI C++ 24
– Anweisung 75, 546
– Anwendungsfall 459, 552
– Array 287, 550
– ASCII 545
– ASCII-Zeichensatz 61
– Aufzählungskonstanten 71, 545
– Ausdruck 77, 546
– ausführbare Programme 544
– Ausnahme 511, 552
– Basisklasse 312
– Baum 376, 551
– Bibliothek 32, 543
– binärer Operator 272, 550
– Bit 537
– Byte 537
– Clone-Methode 344
– Compiler 42, 543
– Daten verbergen 544
– Datenelemente 153, 548
– Datenfeld 287, 550
– Definition (Funktion) 102
– Dekrementieren 81, 546
– doppelt verkettete Liste 551
– Effizienz 549
– einfach verkettete Liste 551
– Elementfunktionen 153, 548
– Elementmethoden 548
– Elementvariablen 153, 548
– elementweise Kopie 260, 549
– Endlosschleife 133, 547
– Exception 511, 552
– flache Kopie 260, 549
– Freispeicher 195, 549
– Friend 404, 551
– Funktion 32, 101, 543
– Funktionsdefinition 547
– Funktionsdeklaration 102, 547
– Gigabyte 538
– globale Variablen 110, 547
– Groß-/Kleinschreibung 62, 545
– Gültigkeitsbereich 110, 547
– Heap 195, 549
– Implementierung 171, 548
– Index 287, 550
– Indirektion 188, 549
– Inkrementieren 81, 546
– Instanzbildung 490
– Instanzen 490
– Instanziierung 490, 552
– Interface 172
– Interpreter 42, 544
– ISO 544
– Iteration 127, 547
– Kamelnotation 63
– Kapselung 44, 153, 544
– Kilobyte 538
– Klasse 32, 152, 543
– Klienten 153, 548
– Kommentare 49, 545
– Kompilieren 42, 544
– konstante Elementfunktion 169, 548
– konstante Zeiger 211
– Konstanten 70, 545
– konzeptionelle Phase 459, 552
– Linken 42, 544
– Linker 42, 543
– literale Konstanten 70, 545

– lokale Variablen 107, 547
– Lösungsraum 460, 552
– L-Wert 79, 546
– Megabyte 538
– Methoden 153
– Methodendefinition 158, 548
– Nibble 537
– Objekt 548
– objektorientierte Programmierung 44, 543
– öffentlich 548
– Operand 77, 546
– Operator 79, 546
– Parameter (Templates) 490
– Parameterliste 547
– Polymorphie 45, 544
– Postfix 82, 546
– Präfix 82, 546
– Präprozessor 46, 545
– private 156, 548
– Problemraum 460, 552
– Prototyp 103, 547
– prozedurale Programmierung 43, 544
– public 156, 548
– Rahmenprogramm 479, 552
– RAM 58, 545
– Rangfolge 84, 546
– Redefinieren 550
– Referenz 217, 549
– rein virtuelle Funktion 360, 551
– relationale Operatoren 86, 546
– R-Wert 79, 546
– Schnittstelle 172, 548
– Signatur 52, 545
– Simulation 458, 552
– Spagetti-Code 547
– Stack 122, 547
– Standardkonstruktor 162, 548
– statische Datenelemente 391, 551
– statische Elementfunktionen 394, 551
– String 303, 550
– strukturierte Programmierung 43, 544
– Stub-Routine 314, 550
– symbolische Konstanten 70, 545
– Template 489, 552
– ternärer Operator 550
– Textzeichenfolge 48
– tiefe Kopie 260, 549
– Token 428, 552
– Typ 58, 151, 545
– Typdefinition 66, 545
– Überschreiben 550
– UML (Unified Modeling Language) 552
– Umleitung 188, 549
– unärer Operator 272, 550
– Variablen 57, 545
– Verbundanweisung 76, 546
– Vererbung 45, 311, 544
– Vergleichsoperatoren 86, 546
– verkettete Liste 375, 551
– verwurzelte Klassen 470
– virtuelle Methode 550
– Vorrang 84, 546
– Vorzeichen 60
– vorzeichenbehaftet 545
– vorzeichenlos 545
– V-Tabelle 340, 550
– Whitespaces 76, 546
– wilde Zeiger 210, 549
– Zeiger 183, 549
– Zeiger auf Konstanten 211
– Zugriffsmethoden 157, 548
– Zuweisungsoperator 77, 546
Benutzeroberfläche
– IDE (Integrated Development Environment) 25
Bereichsüberschreitung 289
Betriebssysteme
– Anforderungen 24
Bezeichner
– Konventionen 526
Bibliotheken 32, 543
– PostMaster 466
– Standard Template Library 506
– Strings 306
Bildschirm
– Ausgaben 48
Binäre Operatoren 272, 550
Binärsystem 536
Binden
– spätes 340
– zur Kompilierzeit 340
– zur Laufzeit 340
Bit 537
Blöcke
– catch 516
– try 516
bool 86

break 131
– switch 145
Bugs 33
Byte 58, 537

C
C# 20
C++
– Standard Template Library 506
C++BuilderX
– Compiler einrichten/installieren 25
– Editor 28
– Haltepunkte 29
– konfigurieren 27
– Projekte 28
catch 512
– abgeleitete Klassen 518
– Standardblock 518
catch(...) 518
C-Dateien 172
char
– Arrays 303
cin 89
– Begrenzer 305
– get() 305
class 153
Clone() 344
Code
– bedingt kompilieren 432
– Bezeichner 526
– catch-Bedingungen, mehrere 518
– catch-Blöcke 512, 516
– const 529
– Definition aufheben 432
– Definitionen auf der Befehlszeile 432
– Entwicklungszyklus 33
– Exceptions 511
– Fehler 509
– Funktionen 32
– geschweifte Klammern 523
– Groß-/Kleinschreibung 526
– include-Dateien 528
– Klassendefinitionen 528
– Kommentare 527
– lange Zeilen 524
– Mehrfachdeklaration 431
– Programmtext 525
– Stil 523
– try-Blöcke 512, 516
– Zugriff 527
Compiler 25, 42, 543
– Befehlszeile 432
– Begleit-CD 22
– boolesche Datentypen 86
– einrichten/installieren 25
– Fehlersuche 170
– Hersteller 31
– integrierte Entwicklungsumgebung 32
– konfigurieren 27
– Linken 42
– Präprozessor 46, 427
– Schalter 428
– Stack 122
– Typprüfung von Konstanten 71
– Vergleich mit Interpreter 42
– vordefinierte Makros 437
– Warnungen 67
– Zwischendateien 428
const 71
– Elementfunktionen 169
– Stil 529
– this-Zeiger 214
– Vergleich zu #define 428
– Zeiger 211
Containment 396
– Datenelemente enthaltener Klassen 402
– Kopieren als Referenz 403
– Kopieren als Wert 403
– Kopierkonstruktoren 403
– Nachteile 403
continue 131
Convert() 108
cout 48
CPP-Dateien 30, 171
CUBE 435

D
Dateien
– C 172
– cat.hpp 174
– CPP 171
– H 46
– Header 172
– HELLO.CPP 33
– HPP 172
– include 528
– Mehrfachdeklaration 431
– OBJ 32
– string.h 306

– temporäre 428
– Wiederverwendbarkeit 400
Dateierweiterungen
– CPP 172
– HPP 172
– Quelldateien 30
Daten
– mit Zeigern manipulieren 191
– statische Datenelemente 391
– verbergen 544
– Zugriff filtern 402
Datenbanken 467
Datenelemente 153, 548
– Containment 396
– geschützte 316
– im Heap 206
– initialisieren 161
– Kopieren als Referenz 403
– Kopieren als Wert 403
– private 157
– statische 391, 551
– Zugriff auf enthaltene Klassen 402
– Zugriff filtern 402
Datenfelder 287, 550
Datenteilung 342
Datentypen
– abstrakte 356, 460
– abstrakte bestimmen 372
– abstrakte Funktionen 360
– ADTs von anderen ADTs ableiten 367
– bool 86
– Klassen als abstrakte 363
Datum und Uhrzeit
– Makros 437
– Speicherung 485
default 145
Definition
– #undef 432
– auch Implementierung 160
– auf der Befehlszeile 432
– aufheben 432
– Funktionen 102
– Konstanten 71
– Konstanten mit #define 428
– Makros 433
– Methoden 171
– Objekte 155
– Variablen 62
Deklaration
– Arrays 287
– Arrays auf dem Heap 299
– Funktionen 102
– Header-Dateien 172
– Indirektion 189
– Inline-Implementierung 173
– Klassen 153, 171
– Klassen als abstrakte Datentypen 363
– parametrisierte Listenobjekte 490
– Schutz gegen mehrfache 431
– Zeiger 187
– Zeiger auf konstante Objekte 212
Dekrementieren 81, 546
delete 197
– aufrufen 200
Dereferenz
– Indirektion 189
Dereferenzierungsoperator 189
Destruktoren 161
– überladen 259
– Vererbung 318
– virtuelle 344
Dimensionen
– Arrays 293
Division (/) 79
do...while-Schleifen 135
Dokumentation 529
– Simulation 459
Doubler() 115, 121
DrawShape() 253
dynamic_cast
– Operator 353

E

Editoren
– Borland-IDE 28
– Nachrichten 468
– Quelldateien 30
Effizienz 235, 549
Eingaben
– Begrenzer 305
– newline 305
– Tastatur 48
Elementfunktionen 153, 548
– Arrays mit Zeigern auf 421
– konstante 169, 548
– Konstruktoren 161
– Namen 154
– Slicing 342
– Standardwerte 256

– statische 394
– this-Zeiger 396
– überladen 253, 258
– virtuelle 340
– Zeiger auf 418
– Zugriffsoperator 418
Elementmethoden 548
Elementvariablen 153, 548
– initialisieren 259
– Namen 154
– statische 391
Elementweise Kopie 260, 549
else 88
E-Mail 467
endl (Neue Zeile) 65
Endlosschleifen 133, 547
– Ereignisschleifen 463
Entwicklungsumgebung
– integrierte 32
Entwicklungszyklus 457
– 80/80-Regel 474
– Entwurfsentscheidungen 477
– Fallstudie PostMaster 466
– Hierarchien entwerfen 470
– Prototypen erstellen 473
– Rahmenprogramme 479
– Schnittstellen 472
– Team-Arbeit 477
Entwurf
– Analyse und Anforderungen der Simulation 459
– Entscheidungen 477
– Entwicklungszyklus 457
– Ereignisschleifen simulieren 463
– Fehler 477
– High-Level 460
– Lösungsraum 460
– Low-Level 460
– Menü 478
– Objekte simulieren 461
– Problemraum 460
– Programme 23
– Rahmenprogramme 479
– Schnittstellen 472
– Simulation 458
enum 71
Ereignisschleifen
– Programmentwurf 463
Erweiterbarkeit 467
Escape-Zeichen 185
Et-Zeichen 95
Exceptions 511, 552
– abfangen 516, 517
– abgeleitete Klassen 518
– abort() 517
– als Referenz abfangen 518
– Anwendung 512
– auslösen 516
– catch 512, 516
– catch(...) 518
– catch-Bedingungen, mehrere 518
– Klassen 516
– Komponenten 512
– Konzept 511
– Polymorphie 518
– Standardblock (catch) 518
– terminate() 517
– throw 516
– try 512, 516
Exponentialoperator 79

F

Factor() 230
Fallstudien
– 80/80-Regel 474
– Bibliotheken 466
– Entwurfsentscheidungen 477
– Hierarchien 470
– Klassenentwurf in PostMaster 468
– PostMaster 466
– PostMasterMessage-Klasse 475
– Prototypen erstellen 473
– Rahmenprogramme 479
– Schnittstellen 472
– Team-Arbeit 477
– Unterprojekte 467
– verkettete Liste 377
false (Wahrheitswert) 86
Fehler 509
– Kompilierzeit 36
– Laufzeitfehler 171
Fehlersuche
– #ifdef DEBUG 432
– Arrays 289
– assert() 440, 528
– Compiler 171
– default-Zweig in switch 146
– Definition statischer Elementvariablen 393
– delete 198
– Ebenen 448

– Fehlerarten 33
– Fence Post Error 290
– for-Anweisung 139
– Klammern in Makros 434
– konstante Funktionen 170
– Leerzeichen in Makro-Definitionen 433
– Makros 436
– Mehrfachdeklaration 431
– Methoden redefinieren 330
– Methoden verbergen 330
– new ohne delete 200
– Schlüsselwort const 330
– Seiteneffekte 440
– Speicher freigeben 198
– vordefinierte Makros 437
– wilde Zeiger 210
– Zeigervariablen 188
– Zeitbomben 211
– Zwischenwerte ausgeben 447
Fence Post Error 290
Filtern
– Zugriff auf enthaltene Datenelemente 402
flache Kopie 260, 549
for-Schleifen 137
– leere Anweisungen 140
– leerer Rumpf 142
– mehrere Anweisungen 139
– verschachtelte 143
free() 200
Freispeicher 195, 549
Freund 404
Friend 551
– Funktionen 405
– Kapselung 404
– Klassen 404
Funktionalität
– Prototypen 474
Funktionen 32, 101, 543
– abort() 517
– abstrakte 360, 551
– Add() 53, 273
– Additionsoperator 272
– als Argumente 111
– Anweisungen 107
– Arbeitsweise 122
– AreaCube() 118
– Argumente 52, 110, 112
– Argumente als Referenz übergeben 224
– Arrays mit Zeigern auf 409
– Arrays mit Zeigern auf Elementfunktionen 421
– assert()-Anweisungen 528
– aufrufen 51
– Convert() 108
– definieren 102, 105
– deklarieren 102
– Doubler() 115, 121
– DrawShape() 253
– Elementfunktionen 153
– Elementfunktionen überladen 253
– Factor() 230
– free() 200
– Friend 405
– geschützte 316
– get() 305
– GetAge() 160
– GetLength() 210
– Header 228
– Inkrement 268
– Inline 119
– Klammern in Makros 434
– konstante Zeiger übergeben 239
– Kopf 52, 105
– Kurzaufruf 408
– lokale Variablen 107
– main() 47, 50, 161
– Makros 433
– malloc() 200
– mehrere Werte zurückgeben 229, 231
– Methoden 153
– Parameter 52, 112
– Parameter mit Referenzen übergeben 226
– Parameter mit Zeigern übergeben 225
– Parameterliste 103
– Polymorphie 45, 119
– Postfix 270
– Prototypen 103, 228
– redefinieren 326
– rein virtuelle 360, 551
– Rückgabewerte 52, 106, 113
– Rückkehr 52
– Rumpf 52, 105
– SetAge() 160
– SetLength() 210
– Signatur 52, 326
– sizeof() 60, 291

– Slicing 342
– Stack 122, 196, 517
– Stack-Rahmen 123
– Standardparameter 116
– statische Elementfunktionen 394, 551
– strcpy() 306
– strncpy() 306
– Stub-Routinen 314
– swap() 113, 225
– terminate() 517
– TheFunction() 245
– this-Zeiger 210, 396
– typedef 415
– überladen 118, 258
– Variablen 107
– Vergleich zu Makros 435
– verlassen 52
– virtuelle Elementfunktionen 340
– void 52
– Zeiger 249, 405
– Zeiger an andere Funktionen 412
– Zeiger auf Elementfunktionen 418
Funktionsdefinition 102, 547
Funktionsdeklaration 102, 547

G
GB (Gigabyte) 538
Gbyte (Gigabyte) 538
Geschwindigkeit
– Inline-Funktionen 120
get() 305
GetAge() 160
GetLength() 210
Gigabyte (Gbyte) 538
Gleichheitsoperator 279
goto 127
Groß-/Kleinschreibung 62, 526, 545
– Kamelnotation 526
Größe
– Arrays 291
Gültigkeitsbereich 547
– lokale Variablen 196
– statische Datenelemente 391
– statische Elementfunktionen 394
– Variablen 110

H
Haltepunkte 29
H-Dateien 46
Header 228
Header-Dateien 172
– Mehrfachdeklaration 431
– string.h 306
Heap 195, 549
– Arrays 297, 299
– Arrays löschen 302
– Datenelemente 206
– delete 197
– Objekte erzeugen 203
– Objekte löschen 204
– Speicher freigeben 197
– Speicher mit new zuweisen 196
– Speicherlücken 199, 208
– Zugriff auf Datenelemente 205
Hexadezimalsystem 539
Hierarchien
– Klassenentwurf 468, 470
– nicht verwurzelte 470
– Vererbung 471
– verwurzelte 470
Hilfe
– Compiler 29
– kontextabhängige 29
HPP-Dateien 172

I
IDE (Integrated Development Environment) 25
– Kommentare 49
if 88
– else-Klausel 88
Implementierung 170, 548
– abstrakte Funktionen 364
– auch Definition 160
– Funktionen 171
– inline 173
– Klassenmethoden 158
– Methoden 171
– Methoden redefinieren 326
include-Dateien 528
Index 287, 550
– Dimensionen 293
– Operator 292
– Zahlensysteme 534
Indirektion 188, 549
Indirektionsoperator 189
Initialisieren
– Arrays 291
– Datenelemente 161
– Elementvariablen 259
– Konstanten 70

– mehrdimensionale Arrays 295
– Objekte 259
– Variablen 64
– Zeiger 187
Inkrementieren 81, 546
– Funktion 268
inline 119, 173
Inline-Funktionen 119
Insert() 384
Instanzbildung 490
Instanziierung 155, 166, 490, 552
Integrated Development Environment (IDE) 25
Interpreter 42, 544
Invarianten 441
Invariants() 441
ISO (International Organization for Standardization) 25, 544
– C++ 24
Iteration 127, 547

K

Kamelnotation 63, 526
Kapselung 44, 153, 544
– Friend 404
Kaufen
– Programme 472
KB (Kilobyte) 538
Kbyte (Kilobyte) 538
Kilobyte (Kbyte) 538
Klammern
– Array-Elemente gruppieren 295
– ausrichten 523
– geschweifte 47
– in verschachtelten if-Anweisungen 92
– Makros 434
– Rangfolge 85
– spitze 46
– verschachtelte 85
Klassen 32, 44, 151, 152, 543
– Ableitung 312
– als abstrakte Datentypen deklarieren 363
– als Datenelemente 175
– API 475
– aufbauen 175
– Basisklassen 312
– Containment 396
– Datenelemente 153
– Datenelemente enthaltener 402
– deklarieren 153
– Elementfunktionen 153
– Elementvariablen 153
– Exceptions 516
– Friend 404
– Header-Dateien 431
– Hierarchien 470
– Implementierung 170, 173
– Inline-Implementierung 173
– Invarianten 441
– Kapselung 153
– Klienten 153
– konstante Methoden 169
– Kopieren als Referenz 403
– Kopieren als Wert 403
– Mehrfachdeklaration 431
– Methoden 153
– Methoden implementieren 158
– Methoden in Basisklasse verbergen 328
– Namen 154
– nicht verwurzelte 470
– Polymorphie 45
– PostMaster 468
– PostMasterMessage 475
– private 156
– Programmentwurf 461
– public 156
– Schnittstellen 170, 172, 359, 404
– statische Elementfunktionen 394
– String 307, 400
– Templates 490
– unveränderliche Bedingungen 441
– Vererbung 312
– Vergleich mit Objekten 155
– verwurzelte 470
– Zugriff auf Elemente 155
– Zugriffsmethoden 157
Klassendefinitionen
– Stil 528
Klienten 153, 548
Knoten (verkettete Listen) 375
– Ende 377
– interne 377
– Kopf 377
Kommentare 48, 527, 545
– /* 49
– // 49
– Tipps 50
– verschachteln 49

Kommunikation 467
Kompilieren 31, 42, 544
– bedingt 432
– Fehlersuche 171
Kompilierzeit 36
– Binden 340
Komponenten
– verkettete Listen 377
Konstanten 70, 545
– #define 428
– Aufzählungen 71
– definieren 71
– initialisieren 70
– literale 70, 545
– symbolische 70, 545
– Typprüfung 71
– vorzeichenlose 130
Konstruktoren 161
– Basisklasse 321
– Kopierkonstruktor 260
– Standardkonstruktoren 162
– überladen 258
– Vererbung 318
– virtuelle Kopierkonstruktoren 344
Konventionen 21
– Bezeichner 526
– Bezeichner bei Mehrfachdeklarationen 432
– Groß-/Kleinschreibung 526
– Makronamen 455
– Quellcode 525
– Variablennamen 63
Konvertierung
– Operatoren 279
Kopf
– Funktionen 52, 105
Kopieren
– Clone-Methoden 344
– elementweise 260
– Strings 306
Kopierkonstruktoren
– Containment 403
– virtuelle 344
– Zuweisung 277

L
Label 127
Laufzeit
– Binden 340
Laufzeitfehler 171
Leeranweisung 75
Linken 31, 42, 544
– Bibliotheken 32
Linker 25, 543
Listen
– Bäume 376
– doppelt verkettete 376
– einfach verkettete 376
– Fallstudie 377
– Objekte deklarieren 490
– parametrisierte 490
– Templates 490
– verkettete 375
Listings
– 9.1 _ addressdemo.cpp _ Adressen von Variablen 184
– addfunction.cpp _ Die Funktion Add() 273
– addressreference.cpp _ Die Adresse einer Referenz ermitteln 219
– adt.cpp _ Abstrakte Datentypen 360
– adtshapeclass.cpp _ Die Klasse Shape mit abstrakten Datentypen 361
– arrayfilled.cpp _ Ein Array füllen 304
– arrayfilled2.cpp _ Ein Array füllen 305
– arrayonheap.cpp _ Ein Array auf dem Heap speichern 297
– arrayptrfunction.cpp _ Array mit Zeigern auf Elementfunktionen 421
– assignoperator.cpp _ Ein Zuweisungsoperator 278
– assignreference.cpp _ Neuzuweisung einer Referenz 221
– baseoverride.cpp _ Eine Methode der Basisklasse in einer abgeleiteten Klasse redefinieren 327
– bodyskip.cpp _ Den Rumpf der while-Schleife überspringen 135
– bracesif.cpp _ Mit Klammern verdeutlichen, welche else-Anweisung zu welcher if-Anweisung gehört 92
– breaker.cpp _ Die Anweisungen break und continue 131
– callfunc.cpp _ Aufruf einer Funktion 51
– callingbase.cpp _ Die Basismethode von der redefinierten Methode aufrufen 331

– comments.cpp _ demonstriert Kommentare 49
– complexwhile.cpp _ Komplexe while-Schleifen 129
– conanddest.cpp _ Aufgerufene Konstruktoren und Destruktoren 319
– constptr.cpp _ Zeiger auf const-Objekte 212
– constructive.cpp _ Konstruktoren und Destruktoren 163
– copyconstructors.cpp _ Kopierkonstruktoren 261
– counter.cpp _ Die Klasse Counter 267
– counterconvert2.cpp _ Einen int in ein Counter-Objekt konvertieren 281
– counterconvertint.cpp _ Konvertieren von Counter nach int() 282
– createreference.cpp _ Referenzen erzeugen und verwenden 218
– datamemberptr.cpp _ Zeiger als Datenelemente 207
– debugginglevels.cpp _ Ebenen für die Fehlersuche 449
– declarefunction.cpp _ Deklaration, Definition und Verwendung einer Funktion 104
– defaultparm.cpp _ Standardwerte für Parameter 117
– derivedobject.cpp _ Ein abgeleitetes Objekt 317
– derivingadt.cpp _ ADTs von anderen ADTs ableiten 368
– dowhile.cpp _ Demonstration einer do…while-Schleife 136
– driverprogram.cpp - Ein Rahmenprogramm für PostMasterMessage 479
– dynamiccast.cpp _ Dynamische Typumwandlung 354
– employeemain.cpp _ Die Klasse Employee und das Rahmenprogramm 400
– exceptions.cpp _ Eine Exception auslösen 513
– express.cpp _ Auswertung komplexer Ausdrücke 78
– forempty.cpp _ Eine leere for-Schleifenanweisung 141
– forloop.cpp _ Beispiel für eine for-Schleife 138
– formulti.cpp _ Mehrere Anweisungen in for-Schleifen 140
– fornested.cpp _ Verschachtelte for-Schleifen 143
– fornull.cpp _ Eine for-Schleife mit leeren Anweisungen 140
– fornullbody.cpp _ Darstellung einer leeren Anweisung als Rumpf einer for-Schleife 142
– func.cpp _ demonstriert eine einfache Funktion 53
– heapaccess.cpp _ Zugriff auf Datenelemente von Objekten auf dem Heap 206
– heapcreate.cpp _ Objekte auf dem Heap erzeugen und löschen 204
– HELLO.CPP 34
– hello.cpp _ Das Programm »Hello World« 34
– hello.cpp _ demonstriert die Teile eines C++-Programms 46
– hellobad.cpp _ Demonstration eines Compilerfehlers 36
– hidingbase.cpp _ Methoden verbergen 329
– ifelse.cpp _ Einsatz der else-Klausel 88
– implement.cpp _ Abstrakte Funktionen implementieren 364
– Implementierung von Cat in cat.cpp 174
– inliner.cpp _ Eine Inline-Funktion 120
– intarray.cpp _ Ein Array für Integer-Zahlen 288
– Klassendeklaration von Cat in cat.hpp 174
– leaky.cpp _ Speicherlücken 246
– linkedlist.cpp _ Eine verkettete Liste 378
– localvar.cpp _ Lokale Variablen und Parameter 107
– localvar2.cpp _ Lokale Variablen und Parameter (mit Variablennamen, die sich zwischen main und Convert unterscheiden) 109

– manyreturns.cpp _ Eine Funktion mit mehreren return-Anweisungen 114
– multidimarray.cpp _ Ein mehrdimensionales Array erzeugen 295
– multivirtual.cpp _ Mehrere virtuelle Elementfunktionen der Reihe nach aufrufen 338
– nestedif.cpp _ Verschachtelte if-Anweisung 90
– newarray.cpp _ Ein Array mit new erzeugen 301
– objarray.cpp _ Ein Array von Objekten erzeugen 292
– overloadderive.cpp _ Konstruktoren in abgeleiteten Klassen überladen 321
– overloadfunctions.cpp _ Überladen von Elementfunktionen 254
– overloadincrement.cpp _ Den Inkrement-Operator überladen 269
– parmlist.cpp _ Parametrisierte Listen (mit verketteten Listen) 491
– passbyptr.cpp _ Übergabe als Referenz mit Zeigern 225
– passbyref.cpp _ Die Funktion swap() mit Übergabe von Referenzen 227
– passbyvalue.cpp _ Übergabe von Argumenten als Wert 224
– passbyvar.cpp _ Übergabe als Wert 112
– passconstptr.cpp _ Übergabe von konstanten Zeigern 239
– passingptrfunction.cpp _ Übergabe von Zeigern auf Funktionen als Funktionsargumente 412
– passobjectsbyref.cpp _ Objekte als Referenz übergeben 236
– passparmlist.cpp _ Parametrisierte Listen (Übergabe als Referenz, verkettete Listen) 499
– passreftoobj.cpp _ Referenzen auf Objekte übergeben 242
– plusoperator.cpp _ Der operator+ 274
– pointerheap.cpp 198
– pointerstore.cpp 192
– pointeruse.cpp 191
– polyexceptions.cpp _ Polymorphe Exceptions 518
– postmasterinterface.cpp _ Die Schnittstelle der Klasse PostMasterMessage 476
– preandpostfix.cpp _ Präfix- und Postfix-Operatoren 271
– prepostfix.cpp _ Präfix- und Postfix-Operatoren 82
– printingvalues.cpp _ Ausgabe von Werten im DEBUG-Modus 447
– properbraces.cpp _ Richtige Verwendung von geschweiften Klammern bei einer if-Anweisung 94
– ptrarrayfunction.cpp _ Ein Array von Zeigern auf Funktionen 409
– ptrtofunction.cpp _ Zeiger auf Funktionen 406
– ptrtomember.cpp _ Zeiger auf Elementfunktionen 418
– rect.cpp _ Deklaration einer vollständigen Klasse 177
– rect.hpp _ Deklaration einer vollständigen Klasse (Header-Datei) 176
– returnref.cpp _ Rückgabe einer Referenz auf ein nicht mehr existierendes Objekt 244
– returnwithptr.cpp _ Rückgabe von Werten mit Zeigern 229
– returnwithref.cpp _ Neufassung von Listing 11.7 mit Übergabe von Referenzen 231
– shapeclass.cpp _ Shape-Klassen 356
– simpleassert.cpp _ Ein einfaches assert()-Makro 438
– simpleclass.cpp _ Methoden einer einfachen Klasse implementieren 158
– simpleevent.cpp _ Eine einfache Ereignisschleife 463
– simpleinherit.cpp _ Einfache Vererbung 314
– sizer.cpp _ Die Größe von Variablentypen für einen Computer bestimmen 59
– slicing.cpp _ Datenteilung bei Übergabe als Wert 342
– staticfunction.cpp _ Statische Elementfunktionen 394
– staticmember.cpp _ Statische Datenelemente 392

– string.hpp _ Die Klasse String 396
– switcher.cpp _ Einsatz der switch-Anweisung 146
– toobigs.cpp _ Addieren einer zu großen Zahl auf eine Zahl vom Typ signed int 69
– toobigu.cpp _ Speichern eines zu großen Wertes in einer Variablen vom Typ unsigned int 68
– typedefer.cpp _ Demonstration von typedef 66
– usevar.cpp _ Einsatz von Variablen 65
– usingdefaults.cpp _ Standardwerte verwenden 256
– usingdefine.cpp _ Einsatz von #define 429
– usinginvariants.cpp _ Die Methode Invariants() 442
– usingparen.cpp _ Klammern in Makros 434
– usingstrcpy.cpp _ Die Funktion strcpy() 306
– usingstrncpy.cpp _ Die Funktion strncpy() 307
– usingthis.cpp _ Den Zeiger this verwenden 209
– usingtypedef _ Einsatz von typedef, um Zeiger auf Funktionen verständlicher zu machen 415
– virtmethod.cpp _ Virtuelle Methoden 336
– virtualcopy.cpp _ Virtueller Kopierkonstruktor 345
– virtualmethods.cpp _ Virtuelle Methoden 351
– while.cpp _ while-Schleifen 128
– whileagain.cpp _ Untersuchung einer while-Schleife 137
– whileforever.cpp _ while(1)-Schleifen 133
– Zeilennummern in Ausgabe 324
Literatur 529
Löschen
– Arrays auf Heap 302
– Objekte 204
Lösungsraum 460, 552
L-Value 79
L-Wert 79, 546

M
main() 47, 161
– Besonderheiten 50
Makro-Erweiterung 435
Makros 433
– ## (Verkettung) 436
– __DATE__ 437
– __FILE__ 437
– __LINE__ 437
– __TIME__ 437
– assert() 438
– CUBE 435
– Datum und Uhrzeit 437
– Ebenen der Fehlersuche 448
– Erweiterung 435
– Fehlersuche 436
– Fehlersuche mit assert() 440
– fortsetzen auf neuer Zeile 435
– Klammern 434
– MAX 433
– MIN 433
– Nachteile 435
– Parameter 433
– Seiteneffekte 440
– String-Manipulation 436
– THREE 435
– Vergleich zu Funktionen 435
– Vergleich zu Templates 435
– vordefinierte 437, 440
– Zwischenwerte ausgeben 447
malloc() 200
Manipulieren
– Strings 436
Maschinensprache 42
MAX 433
MB (Megabyte) 538
Mbyte (Megabyte) 538
Megabyte (Mbyte) 538
Mehrdimensionale Arrays 293
Mehrfachdeklaration 431
Menüs
– PostMaster 478
Methoden 153
– aufrufen 330
– Basismethoden aufrufen 330
– Clone() 344
– const 529
– get() 305
– implementieren 158
– Insert() 384
– Invariants() 441

– konstante 169
– PrintError() 522
– redefinieren 326
– überladen vs. überschreiben (redefinieren) 328
– überschreiben 326
– verbergen 328
– virtuelle 335, 348
– V-Tabelle 340
Methodendefinitionen 158, 548
MIN 433
Modulo-Operation (%) 79
Multiplikation (*) 79

N
Nachrichten
– Editoren 468
– Formate 467
– Formate (PostMaster) 468
Namen
– Bezeichner 526
– Elementfunktionen 154
– Elementvariablen 154
– Groß-/Kleinschreibung 526
– Klassen 154
– Variablen 57
– Zeiger 189
– Zeiger auf Arrays 300
Namensbereiche 35
Neue Zeile
– endl 65
new
– aufrufen 200
– Speicher reservieren 196
newline
– Eingaben abschließen 305
Newsgroups 529
Nibble 537
NOT 95
Null
– Referenzen 223
Nummernzeichen 46
– Präprozessor 427

O
OBJ-Dateien 32
Objektcode 42
Objektdateien 32
Objekte 548
– als Referenz übergeben 235
– Arrays 292
– Clone-Methoden 344
– cout 48
– Datenelemente im Heap 205, 206
– definieren 155
– elementweise Kopie 260, 277
– erstellen 161
– flache Kopie 260, 277
– Gleichheit 279
– Größe 60
– im Heap erzeugen 203
– initialisieren 259
– instantiieren 155
– Kopien auf dem Stack 236
– kopieren 344
– Kopierkonstruktor 260
– löschen 161, 204
– parametrisierte Listen 490
– Referenzen 223
– Referenzen auf nicht existente 244
– simulieren 461
– this-Zeiger 208
– tiefe Kopie 260, 277
– Übergabe bei Templates 498
– Übergabe von Referenzen auf 241
– Vergleich mit Klassen 155
– virtuelle Elementfunktionen 340
Objektorientierte Programmierung 44
öffentlich 548
Offset
– Arrays 288
Online-Dienste 529
OOP
– Kapselung 44
– Merkmale 44
– Polymorphie 45
– Vererbung 45
Operanden 77, 546
– L-Wert 79
– R-Wert 79
operator 486
operator+
– überladen 274
operator=
– überladen 276
Operatoren 79, 546
– 48, 87, 87
– -- (Dekrement) 81
– - (Subtraktion) 79
– ! (logisches NOT) 96
– != (Ungleich) 87

– % (Modulo) 79
– %= (Modulo mit Zuweisung) 81
– & (Adressen) 184
– & (Referenz) 218
– && (logisches AND) 95
– * (Indirektion) 189
– * (Multiplikation) 79
– *= (Multiplikation mit Zuweisung) 81
– + (Addition) 79
– ++ (Inkrement) 81
– += (Addition mit Zuweisung) 81
– / (Division) 79
– /= (Division mit Zuweisung) 81
– -= (Subtraktion mit Zuweisung) 81
– = (Zuweisung) 87
– == (Gleich) 87
– -> (Elementverweis) 205
– > (Größer als) 87
– >= (Größer oder gleich) 87
– || (logisches OR) 95
– Addition (+) 79
– Additionsoperator überladen 272
– binäre 272
– Dereferenzierung 189
– Division (/) 79
– dynamic_cast 353
– Einschränkungen 276
– Exponential 79
– Index 292
– Inkrement überladen 268
– int 282
– logische 95
– mathematische 79
– Modulo (%) 79
– Multiplikation (*) 79
– operator 486
– operator+ überladen 274
– operator= überladen 276
– Postfix 82
– Postfix überladen 270
– Potenz 79
– Präfix 82
– Präfix im Vergleich zu Postfix 270
– Rangfolge 96
– relationale 86, 546
– Strings manipulieren 436
– Strings verketten 436
– Subtraktion (-) 79
– überladen 267
– Umwandlung 279, 282
– unäre 272
– Vergleich 86
– Vorrang 84
– Zeichenkettenbildung 436
– zusammengesetzte 80
– Zuweisung 77
OR 95
Organisation 467

P
Parameter
– const 529
– Funktionen 52
– Makros 433
– Standardparameter 116
– Templates 490
– Typen übergeben 490
– Übergabe als Wert 112
Parameterliste 547
– Funktionen 103
– überladene Funktionen 118
Pareto-Regel 475
Planung 467
Plattformen 467
– bedingte Kompilierung 432
Polymorphie 45, 544
– absteigende Typumwandlung 351
– Exceptions 518
– überladene Funktionen 119
– virtuelle Methoden 335
Postfix 82, 546
– überladen 270
– Unterschied zu Präfix 270
PostMaster
– API 475
– Bibliotheken 466
– Entwurfsentscheidungen 477
– Fallstudie 466
– Hierarchien 470
– Klassenentwurf 468
– Menü 478
– Nachrichtenformate 468
– PostMasterMessage-Klasse 475
– Prototypen erstellen 473
– Rahmenprogramm 479
– Schnittstellen 472
– Unterprojekte 467
Postmaster
– Vererbungshierarchie 471
Potenzoperator 79

Präfix 82, 546
– Unterschied zu Postfix 270
Präprozessor 46, 545
– #define 428
– #else 429
– #ifdef 429
– #ifndef 429
– #include 427
– assert() 438
– Compiler 427
– Klammern in Makros 434
– Konstanten 71
– Leerzeichen in Makro-Definitionen 433
– Makros 433
– Strings manipulieren 436
– Strings verketten 436
– Token 428
– Zeichenkettenbildung 436
PrintError() 522
private 316, 548
Problemraum 460, 552
Programme
– ausführbare 42, 544
– Ausführung 50
– Bibliotheken 472
– Entwicklungsschritte 32
– Geschwindigkeit 120
– Größe 120
– HELLO.CPP 33
– kaufen oder selbst schreiben 472
– Kommentare 48
– Komponenten 45
– Linken 32
– Rahmenprogramme 479
– Speicher freigeben 196
– Verzweigungen 88
Programmiersprachen
– Betriebssysteme 24
– Compiler 42
– Interpreter 42
– Überblick 41
Programmierung
– 80/80-Regel 474
– Analyse und Anforderungen der Simulation 459
– Anweisungen 75
– API 475
– assert()-Anweisungen 528
– const 529
– Dokumentation 459
– Endlosschleifen 134
– Entwicklungszyklus 33, 457
– Entwurf 23
– Fehler 509
– geschweifte Klammern 523
– goto-Anweisungen 128
– Groß-/Kleinschreibung 526
– High-Level-Entwurf 460
– include-Dateien 528
– Klassen simulieren 461
– Klassendefinitionen 528
– Kommentare 527
– konzeptionelle Phase 459
– lange Zeilen im Quellcode 524
– Low-Level-Entwurf 460
– Mehrfachdeklaration 431
– objektorientierte 44, 543
– Prototypen erstellen 473
– prozedurale 43, 387, 544
– Quellcode 525
– Schleifen 127
– Simulation 458
– Stil 523
– strukturierte 43, 544
– Team-Arbeit 477
– Zugriff 527
– Zustände simulieren 462
Projekte 28
protected 316
Prototyp 103, 228, 473, 547
– Architektur 474
– Funktionalität 474
– Funktionen 106
– Schnittstellen 474
public 316, 548
Punktoperator 155

Q

Quellcode
– Entwicklungszyklus 33
– Kommentare 50
– Kompilieren 31
– Linken 31
– Zeilennummern 437
Quelldateien
– Editoren 30
– Erweiterungen 30
Quelltext
– Kommentare 48
– Whitespaces 76
– Zeilennummern 34

R
Rahmenprogramme 479, 552
RAM 58, 545
Rangfolge 84, 546
– Vergleichsoperatoren 96
Redefinieren 550
– Funktionen 326
– Methoden 326
– vs. Überladen 328
Referenzen 217, 549
– Adressoperator 219
– Argumente an Funktionen 224
– auf nicht existente Objekte 244
– auf Objekte 223
– auf Objekte übergeben 241
– erzeugen 218
– Exceptions 518
– mehrere Werte zurückgeben 229, 231
– mit Zeigern übergeben 225
– Null 223
– Objekte übergeben 235
– Speicherlücken 246
– Übergabe mit Zeigern 225
– Vergleich mit Zeigern 243
– Ziel 217
Referenzoperator 218
Register 195
Ressourcen 529
– Speicher 197
return 52, 106, 113
Rückgabewerte 48
– Funktionen 52, 106
– Umwandlungsoperatoren 282
– void 113
Rumpf
– Funktionen 52, 105
R-Value 79
R-Wert 79, 546

S
Schalter
– Compiler 428
Schleifen 127
– break 131
– continue 131
– do...while 135
– Endlos 133
– for 137
– for ohne Rumpf 142
– goto 127
– Label 127
– leere Anweisungen 140
– Sprungmarken 127
– verlassen 131
– verschachtelte 143
– while 128
– while(1) 133
Schlüsselwörter 63
– class 153, 491
– const (this-Zeiger) 214
– const (Zeiger) 211
– delete 197, 200
– enum 71
– inline 120, 173
– new 196, 200
– private 156, 316
– protected 316
– public 156, 316
– return 113
– struct 181
– template 490
– typedef 66
– using 165
– using namespace 133
– void 113
Schnittstellen 170, 359, 548
– API 475
– Klassen 172, 404
– PostMaster 472
– Prototypen 474
Seiteneffekte 440
Semikolon 106
SetAge() 160
SetLength() 210
Signatur 52, 545
– Funktionen 326
Simulation 458, 552
– Analyse und Anforderungen 459
– Condition-Objekte 462
– Dokumentation 459
– Ereignisschleifen 463
– High-Level-Entwurf 460
– Klassen 461
– konzeptionelle Phase 459
– Low-Level-Entwurf 460
– Objekte 461
– Zustände 462
sizeof() 60, 291
Slicing 342
Sonderzeichen 185
Spagetti-Code 128, 547

Spätes Binden 340
Speicher
– Adressen 183
– Adressen mit Zeigern manipulieren 192
– anfordern 197
– Arrays 296
– Arrays auf dem Heap 299
– Daten mit Zeigern manipulieren 191
– Datenelemente im Heap 205
– freigeben 196
– Freispeicher 195
– Heap 195
– konstante Zeiger 211
– mit delete freigeben 197
– Objekte erzeugen 203
– Objekte im Heap löschen 204
– RAM 58
– Register 195
– Ressourcen 197
– Stack 122, 195
– statische Elementvariablen 393
– this-Zeiger 208
– Variablen 58, 186
– wilde Zeiger 210
– Zeiger 190, 249
– Zeiger auf Datenelemente 206
Speicheradressen 183
– new 196
Speicherlücken 197
– Arrays löschen 302
– Destruktor 208
– Heap 199, 208
– Referenzen 246
Sprungmarken 127
Stack 122, 195, 547
– auflösen 517
– Aufrufe 517
– Exceptions abfangen 517
– Funktionen 517
– Kopien von Objekten 236
– lokale Variablen 195
Stack-Pointer 122
Stack-Rahmen 123
Standard Template Library 506
Standardbibliothek
– using 165
Standardkonstruktoren 162, 548
– Arrays 292
– Kopierkonstruktor 260
Standardparameter 116
Standardwerte
– Elementfunktionen 256
– Vergleich zu überladenen Funktionen 258
Stapelspeicher 122
Stil 523
– assert()-Anweisungen 528
– geschweifte Klammern 523
– Groß-/Kleinschreibung 526
– include-Dateien 528
– Klassendefinitionen 528
– Kommentare 527
– lange Zeilen 524
– Namen von Bezeichnern 526
– Programmtext 525
– switch-Anweisungen 524
– Zugriff 527
strcpy() 306
String (Klasse) 400
string.h 306
stringizing operator 436
Strings 550
– #define 428
– abschließen 304
– Arrays 303
– Begrenzer 305
– Bibliothek 306
– Ersetzung 428
– Klassen 307
– kopieren 306
– Makros 436
– manipulieren 436
– Null-Zeichen 304
– strcpy() 306
– strncpy() 306
– Token 428
– verketten 436
strncpy() 306
Stroustrup 20
struct 181
Stub-Routinen 314, 550
Subtraktion (-) 79
swap
– Übergabe mit Zeigern 225
swap() 113, 225
– Übergabe mit Referenzen 226
switch-Anweisungen 145
– break 145
– default 145
– im Quellcode einrücken 524
Symbole, Makros 433

Syntax
– Ableitung 314
Syntaxfehler 33

T
Tabellen
– virtuelle Methoden 340
Tabulatorzeichen 185
– Quelltextgestaltung 524
Tastatur
– Eingaben 48
Team-Arbeit 477
Telefon
– erweitertes 45
Templates 489, 552
– erstellen 491
– Instanzbildung 490
– Instanzen 490
– Objekte übergeben 498
– parametrisierte Listen 490
– Standard Template Library 506
– Vererbung 507
– Vergleich zu Makros 435
terminate() 517
Ternärer Operator 550
Testen
– #define 429
TheFunction() 245
this-Zeiger 208
– konstante 214
– statische Elementfunktionen 396
THREE 435
throw 516
Tiefe Kopie 260, 549
Tilde 161
Token 428, 552
true (Wahrheitswert) 86
try 512
Typdefinition 66, 545
typedef 66, 415, 545
Typen 151, 545
– abgeleitete 45
– Exceptions 516
– Konstanten 71
– konvertieren 282
– neue erzeugen 152
– Parameter 490
– Rückgabewerte 52
– Variablen 58
– vordefinierte 267
– Vorzeichen 60
Typprüfung
– #define 428
– const 71
– Konstanten 71
Typumwandlung
– absteigende 351
– Compiler-Warnungen 67
– dynamische 351, 353
– Zeiger 341

U
Übergabe
– als Referenz mit Zeigern 225
– Argumente als Referenz 224
– konstante Zeiger 239
– Objekte als Referenz 235
– Referenzen auf Objekte 241
– Template-Objekte 498
– Zeiger auf Funktionen an andere Funktionen 412
Überladen
– Additionsoperator 272
– Destruktoren 259
– Einschränkungen 276
– Elementfunktionen 253, 258
– Funktionen 118, 258
– Gleich-Operator 279
– Inkrement-Operator 268
– Konstruktoren 258
– operator+ 274
– operator= 276
– Operatoren 267
– Postfix-Operator 270
– Rückgabetyp 118
– vs. Redefinieren 328
Überschreiben 326, 550
– vs. Überladen 328
UML (Unified Modeling Language) 460, 552
Umleitung 188, 549
Umleitungsoperator 48
Umwandlung
– int 282
– Operatoren 279
Unäre Operatoren 272, 550
USHORT 66
using 165
using namespace 133

V
Variablen 545
– Adressen 57, 187
– Auswahl eines Typs 67
– Begriff 57
– Bereichsüberschreitung 68
– const 529
– Datenelemente von Klassen 153
– definieren 62
– globale 110, 547
– Groß-/Kleinschreibung 62
– Größe 58
– Gültigkeitsbereich 110
– Indirektion 189
– initialisieren 64
– konstante this-Zeiger 214
– konstante Zeiger 211
– lokale 107, 195, 547
– mehrere gleichzeitig definieren 64
– mit new im Heap erzeugen 196
– Namen 57, 63
– neue Typen erzeugen 152
– Platzbedarf 186
– Speicher 58
– Speicher mit delete freigeben 197
– Speicherlücken 199
– Speicherung 186
– this-Zeiger 208
– Typen 58, 61
– Wert 58
– Werte zuweisen 64
– wilde Zeiger 210
– Zeiger 190
– Zeiger auf Datenelemente 206
– Zeiger deklarieren 187
Verbergen
– Methoden 328
Verbundanweisungen 546
Vererbung 45, 311, 544
– Ableitung 312
– Argumente an Konstruktoren der Basisklasse 321
– Destruktoren 318
– Funktionen redefinieren 326
– Hierarchien 471
– Konstruktoren 318
– private vs. protected 316
– Syntax 314
– Templates 507
Vergleichsoperatoren 86, 546
– Rangfolge 96
Verketten
– Strings 436
Verkettete Listen 375, 551
– Endeknoten 377
– interne Knoten 387
– Knoten 375
– Komponenten 377
– Kopfknoten 377
Verschachteln
– Klammern 85
– Schleifen 143
Verzweigungen 88
Virtuelle Destruktoren 344
Virtuelle Elementfunktionen 340
– Slicing 342
Virtuelle Kopierkonstruktoren 344
Virtuelle Methoden 335, 550
– Effektivität 348
void 52, 113
Vorrang 84, 546
Vorzeichen 60
Vorzeichenbehaftet 545
Vorzeichenlos 545
V-Tabelle 340, 550
V-Zeiger 340

W
Wahrheitswerte 86
– bool 86
Warnungen
– Compiler 67
Werte
– Variablen 58
– zuweisen 64
while(1)-Schleifen 133
– Ereignisschleifen simulieren 463
while-Schleifen 128
Whitespaces 76, 546
– Makro-Definition 433
Wiederverwendbarkeit 44, 45, 400

Z
Zahlensysteme 533
– 0-te Potenz 535
– Basis 534
– Binärsystem 536
– Hexadezimalsystem 539
Zeichenkettenbildung 436
Zeichensätze, ASCII 61
Zeiger 183, 190, 249, 549
– Adressen 187

– Adressen manipulieren 192
– Array-Namen 300
– Arrays 297
– auf Arrays 300
– auf Elementfunktionen 418
– auf Funktionen 415
– auf Funktionen an andere Funktionen 412
– auf konstante Objekte deklarieren 212
– Daten manipulieren 191
– Datenelemente im Heap 206
– deklarieren 187
– delete 197
– Einsatzfälle 194
– Funktionen 405
– Größe 187
– Heap 195
– Indirektion 188
– Indirektionsoperator 189
– initialisieren 187, 196
– konstante 211
– konstante this-Zeiger 214
– Kurzaufruf 408
– löschen 198
– Namen 189
– neu zuweisen 199
– new 196
– Null 197
– Objekte im Heap erzeugen 203
– Objekte im Heap löschen 204
– Polymorphie 335
– Speicher freigeben 198
– this 208
– typedef 415
– Typumwandlung 341
– Übergabe konstanter 239
– Vergleich mit Referenzen 243
– V-Zeiger 340
– wilde 187, 210, 261, 549
Zeigerarithmetik
– Arrays 300
Zeilennummern 34, 238
– __LINE__ 437
– Ausgabe 324
Zeilenschaltung 48, 160
Zeitbomben 211
Zugriff
– Arrays 288
– auf Elemente enthaltener Klassen 402
– Datenelemente im Heap 205
– kennzeichnen 527
– Klassenelemente 155
Zugriffsfunktionen
– konstante 169
Zugriffsmethoden 157, 548
Zugriffsoperator
– Zeiger auf Elementfunktionen 418
Zustände
– simulieren 462
Zuweisungen 75
– Postfix 82
– Präfix 82
– Variablen 64
Zuweisungsoperator 77, 546
Zwischendateien 428
Zwischenwerte 447

InformIT™
CHECK IT

... aktuelles Fachwissen rund
um die Uhr – zum Probelesen,
Downloaden oder auch auf Papier.

www.InformIT.de

InformIT.de, Partner von **Markt+Technik**, ist unsere Antwort auf alle Fragen der IT-Branche.

In Zusammenarbeit mit den Top-Autoren von Markt+Technik, absoluten Spezialisten ihres Fachgebiets, bieten wir Ihnen ständig hochinteressante, brandaktuelle Informationen und kompetente Lösungen zu nahezu allen IT-Themen.

wenn Sie mehr wissen wollen ... **www.InformIT.de**

Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt. Dieses eBook stellen wir lediglich als persönliche Einzelplatz-Lizenz zur Verfügung!

Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschliesslich

- der Reproduktion,
- der Weitergabe,
- des Weitervertriebs,
- der Platzierung im Internet, in Intranets, in Extranets,
- der Veränderung,
- des Weiterverkaufs
- und der Veröffentlichung

bedarf der schriftlichen Genehmigung des Verlags.

Insbesondere ist die Entfernung oder Änderung des vom Verlag vergebenen Passwortschutzes ausdrücklich untersagt!

Bei Fragen zu diesem Thema wenden Sie sich bitte an: info@pearson.de

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf unseren Websites ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und viele weitere eBooks können Sie rund um die Uhr und legal auf unserer Website

http://www.informit.de

herunterladen