

Walter Saumweber

C++ Programmierhandbuch



Walter Saumweber

**C++**

# Programmierhandbuch

**entwickler.press**

Walter Saumweber:  
C++ Programmierhandbuch  
ISBN: 978-3-939084-50-1

© 2007 entwickler.press  
Ein Imprint der Software & Support Verlag GmbH

<http://www.entwickler-press.de>  
<http://www.software-support.biz>

Ihr Kontakt zum Verlag und Lektorat: [lektorat@entwickler-press.de](mailto:lektorat@entwickler-press.de)

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

Satz: text&form GbR, Carsten Kienle  
Titelgrafik: Melanie Hahn, Maria Rudi  
Umschlaggestaltung: Caroline Butz  
Belichtung, Druck & Bindung: M.P. Media-Print Informationstechnologie GmbH,  
Paderborn

Alle Rechte, auch für Übersetzungen, sind vorbehalten. Reproduktion jeglicher Art (Fotokopie, Nachdruck, Mikrofilm, Erfassung auf elektronischen Datenträgern oder andere Verfahren) nur mit schriftlicher Genehmigung des Verlags. Jegliche Haftung für die Richtigkeit des gesamten Werks kann, trotz sorgfältiger Prüfung durch Autor und Verlag, nicht übernommen werden. Die im Buch genannten Produkte, Warenzeichen und Firmennamen sind in der Regel durch deren Inhaber geschützt.

# Inhaltsverzeichnis

<b>Liebe Leserin, lieber Leser!</b>	<b>13</b>
<b>Teil I</b>	
<b>Grundlagen</b>	<b>15</b>
<b>1 Was ist Programmieren?</b>	<b>17</b>
1.1 Was ist eigentlich ein Computerprogramm?	17
1.2 Entwicklungsschritte zum ausführbaren Computerprogramm	20
Aufsetzen des Quellcodes	20
Kompilierung	20
Präprozessor und Linker	22
1.3 Was versteht man unter einer IDE?	23
1.4 Was sind Konsolenprogramme?	24
<b>2 Welchen Compiler sollten Sie verwenden?</b>	<b>29</b>
2.1 Borland C++-Compiler	29
2.2 Wie finde ich einen passenden Compiler?	30
2.3 Installation des Borland-Compilers	31
<b>3 Ihr erstes C++-Programm</b>	<b>35</b>
3.1 Die Lösung vorweggenommen	35
C++ und Zwischenraumzeichen	37
3.2 Von Anfang an	39
Aufbau von C++-Programmen	39
Funktionen	40
#include <iostream>	43
using namespace std;	44
<b>4 Kompilieren und Ausführen von C++-Programmen</b>	<b>47</b>
4.1 Was für einen Editor sollten Sie verwenden?	47
4.2 Kompilierung	49
4.3 Ausführen der .exe-Datei	52
Ausführung über den Windows-Explorer	53

<b>5</b>	<b>Über Programm(ier)fehler</b>	<b>55</b>
5.1	Unterscheidung der verschiedenen Fehlerarten	55
	Syntaxfehler	55
	Programmfehler	57
5.2	Fehlermeldungen »aus Compilersicht«	58
5.3	Historisches	60
	C/C++	60
	ANSI/ISO-Standard	62
 <b>Teil II</b>		
	<b>Das C++-ABC</b>	<b>63</b>
<b>6</b>	<b>Kommentare</b>	<b>65</b>
6.1	Einzeilige Kommentare	65
6.2	Mehrzeilige Kommentare	67
6.3	Einsatz von Kommentaren bei der Fehlersuche	70
<b>7</b>	<b>Syntaxregeln</b>	<b>75</b>
7.1	Textbausteine	75
7.2	Anweisungsende	77
7.3	Blöcke	77
7.4	Leerräume	80
7.5	Programmierstil	83
<b>8</b>	<b>Ausgabe mit cout</b>	<b>85</b>
8.1	Datenströme	85
8.2	C++ unterscheidet zwischen verschiedenen Datentypen	86
8.3	Ausgabeeinheiten aneinander hängen	89
	In der Programmierung gibt es meist mehrere Lösungen	89
	Anweisungen auf mehrere Zeilen verteilen	90
	Verkettung von Strings	91
8.4	Manipulatoren	92
<b>9</b>	<b>Steuerzeichen</b>	<b>95</b>
9.1	Die Escape-Sequenzen \", \' und \\	95
9.2	ASCII-Code	98
9.3	Darstellung von Zeichen mittels Escape-Sequenzen	101
	Ausgabe von Umlauten sowie des Zeichens »ß«	103
9.4	Das Steuerzeichen \n	104
9.5	Weitere Steuerzeichen	106

<b>10</b>	<b>Variablen</b>	<b>111</b>
10.1	Variablen deklarieren	111
	Mehrere Variablen mit einer Anweisung deklarieren	113
10.2	Regeln zur Bezeichnerwahl	114
10.3	Zuweisung	116
	Additionsoperator (+)	121
	Initialisierung	124
	string-Variablen	125
10.4	Das Objekt cin	127
10.5	Dreieckstausch	130
<b>11</b>	<b>Ausdrücke und Operatoren</b>	<b>133</b>
11.1	Was ist ein Ausdruck?	133
	Mehrere Zuweisungen hintereinander schalten	134
	Komplexe Ausdrücke	135
11.2	Arithmetische Operatoren	137
	Der Modulo-Operator	138
11.3	Zusammengesetzte Zuweisungsoperatoren	140
11.4	Inkrement- und Dekrementoperatoren	142
11.5	Priorität von Operatoren	145
<b>12</b>	<b>Zahlen mit Nachkommastellen</b>	<b>149</b>
12.1	Die Datentypen double und float	149
	sizeof-Operator	150
12.2	Literele zur Darstellung von Zahlen mit Nachkommastellen	152
12.3	Ein Programm zur Berechnung der Mehrwertsteuer	155
12.4	Konstanten mit const deklarieren	159
<b>13</b>	<b>Ausgabe mit Manipulatoren formatieren</b>	<b>165</b>
13.1	Standardeinstellungen	165
	Genauigkeit	165
	Ausgabeformat	168
13.2	Die Manipulatoren setiosflags() und resetiosflags()	172
13.3	Die Manipulatoren fixed und scientific	177
13.4	setprecision()	179
	setprecision() in Verbindung mit Scientific-/Fixed-Format	182
	Anzeige von Nachkommastellen begrenzen	183
13.5	Feldbreite setzen	185
	Füllzeichen festlegen mit setfill()	186
	Ausgaben linksbündig/rechtsbündig ausrichten mit left/right	187

<b>14</b>	<b>Datentypen</b>	<b>191</b>
14.1	Welche Datentypen gibt es noch?	191
	Datentypqualifizierer	192
14.2	Literale	197
	Literale zur Darstellung von ganzzahligen Werten	198
	Gleitkommaliterale	201
<b>15</b>	<b>Typumwandlungen</b>	<b>203</b>
15.1	Implizite Typumwandlungen	203
	Konvertierung von char nach int	205
15.2	Wann gehen bei der Konvertierung Informationen verloren?	208
15.3	Welchen Datentyp hat ein bestimmter Ausdruck?	210
	Ausdrücke als Operanden des sizeof-Operators	213
	Datentypen werden nicht konvertiert	215
	Reihenfolge der Konvertierungen	217
	char und short	219
15.4	Explizite Typumwandlungen	220
<b>16</b>	<b>Verzweigungen</b>	<b>223</b>
16.1	Logische Ausdrücke	223
	Vergleichsoperatoren	223
	Logische Operatoren	225
	Priorität von logischen und Vergleichsoperatoren	229
16.2	Die if-Anweisung	231
	Verschachteln von Kontrollstrukturen	235
	Konvertierung in logischen Ausdrücken	237
	if else	239
	Stringvergleiche	241
	else if	241
16.3	Die switch-Anweisung	247
16.4	Bedingungsoperator	257
16.5	Zufallszahlen auslosen	259
<b>17</b>	<b>Wiederholungsanweisungen</b>	<b>267</b>
17.1	Die while-Schleife	267
	Endlosschleifen	270
	Fakultät berechnen	271
17.2	Die do while-Schleife	275
17.3	Die for-Schleife	277
	Sequenzoperator	284
17.4	Die Anweisungen break und continue	285



17.5	Gültigkeitsbereich von Variablen	288
	»Lebensdauer« von Variablen	290
	static-Variablen	294
	Namensgleichheit	296
<b>18</b>	<b>Arrays</b>	<b>299</b>
18.1	Deklaration von Arrays	299
18.2	Mit Arrays arbeiten	300
18.3	Arrays in for-Schleifen durchlaufen	302
18.4	Initialisierung von Arrays	305
	Die Größe eines Arrays bestimmen mit dem sizeof-Operator	307
18.5	Mehrdimensionale Arrays	308
<b>19</b>	<b>Strings</b>	<b>313</b>
19.1	Wie Zeichenketten dargestellt werden	313
19.2	Arrays und Adressen	315
	cin.get()	319
	cout.put()	323
	Die Funktionen getch(), getche(), kbhit()	324
19.3	Funktionen zur Stringverarbeitung	326
	strcmp()	326
	strcpy()	328
	Die Konvertierungsfunktionen atoi(), itoa(), atof()	328
19.4	string-Variablen	331
	Die Methode c_str()	333
<b>20</b>	<b>Funktionen</b>	<b>337</b>
20.1	Funktionen definieren und aufrufen	337
20.2	Funktionsprototypen	343
	Den Quellcode auf mehrere Dateien verteilen	346
	Prototypen von vordefinierten Funktionen	348
20.3	Funktionsparameter	351
	Wertübergabe	356
	Arrays an Funktionen übergeben	358
20.4	Rückgabewerte von Funktionen	360
	Rückgabewert von main()	363
	Vorgabeargumente	364
20.5	Überladen von Funktionen	365
20.6	Rekursionen	367
20.7	inline-Funktionen	370
20.8	Globale Variablen	372

<b>21</b>	<b>Eine Funktionsbibliothek</b>	<b>375</b>
21.1	Funktion ArrMinIndex()	375
21.2	Funktion sortiereArr()	379
21.3	Funktion doppelte()	382
<b>22</b>	<b>Ein Lottospiel</b>	<b>387</b>
22.1	Im Programm Lottozahlen auslosen	387
22.2	Tippen	391
22.3	Gewinnanzeige	393
 <b>Teil III</b>		
	<b>Einführung in die objektorientierte Programmierung</b>	<b>397</b>
<b>23</b>	<b>Strukturen</b>	<b>399</b>
23.1	Strukturen definieren	399
23.2	Auf die Komponenten von Strukturen zugreifen	404
23.3	Ausblick	410
<b>24</b>	<b>Klassen und Objekte</b>	<b>413</b>
24.1	Methoden	413
	Methoden außerhalb einer Klasse definieren	418
	Den Code von Klassendefinitionen auf mehrere Dateien verteilen	420
	Wie man vermeidet, dass eine Klassendefinition mehrfach eingebunden wird	421
24.2	Zugriffsspezifizierer	424
24.3	Konstruktoren und Destruktoren	431
	Überladen von Konstruktoren	434
	Ersatzkonstruktor	436
24.4	Was es mit Namensräumen auf sich hat	440
<b>25</b>	<b>Statische Elemente einer Klasse</b>	<b>443</b>
25.1	Statische Attribute	443
25.2	Statische Methoden	446
<b>26</b>	<b>Dateioperationen</b>	<b>455</b>
26.1	In Textdateien schreiben	455
26.2	Aus Textdateien lesen	461
	Programm zum Verwalten von (Entenhausener) Bankkunden	466

## Teil IV

<b>Fortgeschrittene Programmierkonzepte</b>	<b>475</b>
<b>27 Präprozessor-Direktiven</b>	<b>477</b>
27.1 #include	477
27.2 Symbolische Konstanten mit #define vereinbaren	478
27.3 Bedingte Kompilierung	479
<b>28 Zeiger</b>	<b>485</b>
28.1 Der Adressoperator (&)	485
28.2 Zeigervariablen deklarieren und verwenden	487
Wilde Zeiger	490
Dereferenzierung von Zeigern	491
Konstante Zeiger	493
Elementzugriff über Zeiger	495
28.3 Zeiger als Parameter von Funktionen	498
28.4 Zeiger als Klassenelemente	500
28.5 Der this-Zeiger von Methoden	502
28.6 Speicherplatz dynamisch anfordern	504
Arrays dynamisch allozieren	507
Destruktoren sorgen für die Aufräumarbeiten	509
<b>29 Referenzen</b>	<b>521</b>
29.1 Was sind Referenzen?	521
29.2 Referenzen als Parameter von Funktionen	524
29.3 Referenzen als Rückgabewerte von Funktionen	526
<b>30 Vererbung</b>	<b>529</b>
30.1 Klassen von Basisklassen ableiten	529
30.2 Zugriff auf die geerbten Elemente einer Basisklasse	531
private-Vererbung	531
protected-Vererbung	536
public-Vererbung	540
30.3 Überschreiben von Methoden	541
<b>31 Überladen von Operatoren</b>	<b>547</b>
31.1 Welche Operatoren lassen sich überladen?	548
31.2 Definition von Operatormethoden	548
Überladen von binären Operatoren	549
Überladen von unären Operatoren	560
Überladen eines Vergleichsoperators	566

<b>32</b>	<b>Ausnahmebehandlung</b>	<b>569</b>
32.1	try – catch	569
	Benutzerdefinierte Klassen für die Fehlerbehandlung	574
	Mehrere catch-Handler	576
32.2	Auffangen von Ausnahmen im Aufrufer	577
<b>Bonusteil</b>		
	<b>Managed Code; GUI-Programmierung</b>	<b>581</b>
<b>33</b>	<b>Microsoft und das .NET</b>	<b>583</b>
33.1	Microsoft Intermediate Language	583
33.2	Das .NET Framework	585
33.3	Common Language Runtime	586
	Garbage-Collection	587
	Programmieren für .NET	587
<b>34</b>	<b>Windows-Anwendungen</b>	<b>589</b>
34.1	Windows Forms	589
	Ein neues Projekt beginnen	590
	Das Eigenschaftsfenster im Visual Studio	593
	Weitere Steuerelemente	596
34.2	Steuerelemente mit Code verbinden – Ereignisbehandlungsroutinen	599
<b>ANHANG</b>		
<b>A</b>	<b>CD-ROM zum Buch</b>	<b>607</b>
<b>B</b>	<b>Schlüsselwörter in C++</b>	<b>609</b>
<b>C</b>	<b>Prioritätsreihenfolge der C++-Operatoren</b>	<b>611</b>
<b>D</b>	<b>ASCII-Tabelle</b>	<b>613</b>
<b>E</b>	<b>Glossar</b>	<b>615</b>
	<b>Stichwortverzeichnis</b>	<b>619</b>

# Liebe Leserin, lieber Leser!

Vielen Dank, dass Sie sich für dieses Buch entschieden haben. Es sollte Ihnen dazu verhelfen, auf möglichst angenehme Weise ein gutes Stück Weg in der Welt der Programmierung im Allgemeinen und in der von C++ im Besonderen hinter sich zu bringen. Allerdings ist wie bei den meisten Dingen im Leben auch hierzu etwas Geduld erforderlich. Gerade dem programmiererunfähigen Leser sei es deshalb nahe gelegt, diese aufzubringen, auch wenn ihm die ersten Kapitel möglicherweise etwas langweilig erscheinen werden (dem Fortgeschrittenen steht es natürlich frei, an einer späteren Stelle mit dem Lesen zu beginnen).

Ich habe mich grundsätzlich bemüht, die Lerninhalte mit möglichst vielen prägnanten, aber einfachen Beispielen zu verdeutlichen. Die Lektüre dieses Buches sollte daher auch ohne unmittelbaren Rechnerkontakt möglich sein (ich selbst gehöre zu den Leuten, die nicht gerne am Computer sitzen, wenn sie ein Buch lesen). Praktische Übung, zumindest im Nachhinein, ist natürlich immer anzuraten.

Alle abgedruckten Listings plus erweiterte Beispiele finden Sie, nach Kapiteln geordnet, als Quellcodedateien und – soweit es sich um größere oder inhaltlich wichtige Programme handelt – samt ausführbarer Datei auf der Buch-CD im Verzeichnis *Beispiele*. Speziell das Visual-Studio-Projekt aus Kapitel 34 befindet sich als *.zip*-Archiv im Unterverzeichnis *K34*.

Die Beispiele können Sie mit dem Borland C++-Compiler kompilieren. Er steht ebenfalls auf der Buch-CD im Verzeichnis *Borland-Compiler* zur Verfügung (Dateiname *freecommandLinetools.exe*). Die Installation und Bedienung ist in Kapitel 2 und 4 beschrieben. Es sei jedoch darauf hingewiesen, dass Sie genauso gut jeden anderen aktuellen Compiler verwenden können.

Auf die Abhandlung einiger Fortgeschrittenen-Themen wie beispielsweise Templates habe ich bewusst – wenn auch nicht ganz ohne schlechtes Gewissen – verzichtet. Als Gegenleistung sollte der Leser eine gut verständliche Darstellung der behandelten Themen in Händen halten (da diese Beurteilung letzten Endes immer auf persönlicher Einschätzung beruht, möchte ich hiermit der Hoffnung Ausdruck geben, dass mir dies gelungen ist).

Um aus dem Buch den größtmöglichen Nutzen zu ziehen, empfehle ich Ihnen, sich für die Lektüre und für das Nachvollziehen der Beispiele Geduld und Zeit zu nehmen.

Eventuell zusätzliche Infos zum Buch sowie gegebenenfalls eine Fehlerliste werde ich beizeiten auf meiner Webseite <http://www.mipueblo.de> veröffentlichen. Über die dort angegebene E-Mail-Adresse können Sie mich jederzeit kontaktieren. Für Anregungen und Hinweise bin ich Ihnen schon jetzt sehr dankbar.

Und nun viel Spaß beim Lesen und viel Erfolg mit der Programmierung.

Walter Saumweber



# Teil I

## Grundlagen

Dieser einführende Teil soll Ihnen den Einstieg in die Programmierung erleichtern. Bevor wir uns speziell mit der Programmiersprache C++ befassen, werden wir zunächst auf allgemeine Aspekte der Programmierung eingehen. Insbesondere werden Sie mit Fachbegriffen bekannt gemacht, die Ihnen im weiteren Verlauf des Buches immer wieder begegnen werden. Im Anschluss daran werden Sie ein erstes kleines C++-Programm erstellen und ausführen, wobei Sie einige Tipps zu Entwicklungswerkzeugen erhalten. Falls Sie schon einige Programmiererfahrung haben und sich auch die Frage der zu verwendenden Software für Sie nicht stellt, steht es Ihnen natürlich frei, diesen Teil zu überspringen und sogleich mit Teil 2 fortzufahren.





# 1

# Was ist Programmieren?

»... sich um das Entstehen einer ausführbaren Datei zu kümmern.« Diese einfache Antwort ist nahe liegend, da das Endprodukt jeder erfolgreichen Programmentwicklung in einer solchen (ausführbaren Datei) besteht.

Eine ebenso lapidare – und auch mehr oder weniger korrekte – Erwiderung würde lauten: »Programmieren bedeutet, Text in einen Editor einzugeben.«

## Hinweis

Die eigentliche Schwierigkeit liegt natürlich darin, dies syntaktisch und logisch korrekt, effizient und auch für andere Programmierer nachvollziehbar zu tun (Letzteres wird beim Modifizieren von Programmen bedeutsam). Was das alles genau bedeutet, werden Sie im Weiteren sehen.

Da eine Textdatei eben nicht ausführbar ist, scheint die zweite Aussage im Widerspruch zur ersten zu stehen. Lassen Sie uns die Klärung mit einer weiteren Frage einleiten.

## 1.1 Was ist eigentlich ein Computerprogramm?

Ein ausführbares Programm enthält Befehle, die vom Computer ausgeführt werden. Die Reihenfolge der Abarbeitung ist dabei durch die Kodierung der Programmdatei festgelegt.

## Hinweis

Die Begriffe »Programmdatei«, »ausführbare Datei«, »Programm« und »ausführbares Programm« werden gleichbedeutend verwendet. Es handelt sich um solche Dateien, die vom Computer zur Ausführung vorgesehen sind. Dies trifft z.B. für Text- oder Grafikdateien nicht zu.

Zu den Begriffen »Programm« sowie »ausführbares Programm« werden bezüglich dieser Definition später noch gewisse Einschränkungen zu nennen sein, die aber hier keine Rolle spielen.

Programmdateien können auf verschiedene Weise zur Ausführung gelangen. Möglich ist ein Aufruf aus bereits laufenden anderen Programmen heraus: Programm A startet Programm B (dieses wiederum Programm C usw.). Einige Programme werden vom Betriebssystem automatisch beim Start des Computers in Gang gesetzt.

Am anschaulichsten geschieht dies jedoch durch einen expliziten Startbefehl des Anwenders – durch einen Doppelklick auf die Programmdatei im Windows-Explorer, mithilfe einer Verknüpfung (Desktop, Startmenü) oder über einen Befehl in der Konsole (die bei DOS und Windows als »Eingabeaufforderung« bezeichnet wird).

Wenn Sie beispielsweise mit dem Tabellenkalkulationsprogramm Microsoft Excel arbeiten möchten, werden Sie dieses vermutlich über einen Eintrag im Startmenü oder mittels eines Symbols auf dem Desktop aufrufen. Da es sich bei beiden um Verknüpfungen zur entsprechenden Programmdatei handelt, wird diese damit gestartet. Der Name der Excel-Programmdatei lautet übrigens *excel.exe*. Sie befindet sich in der Regel im Verzeichnis *C:\Programme\Microsoft Office\Office* bzw. *C:\Programme\Microsoft Office\OFFICE11* oder *C:\Programme\Microsoft Office\Office12*.

### Achtung

Für alle Pfadangaben in diesem Buch gilt: Falls sich bei Ihnen die Dateien bzw. Verzeichnisse auf einem anderen Laufwerk befinden, ersetzen Sie den Laufwerksbuchstaben C durch den jeweiligen Buchstaben.

Eine Programmdatei wird in den meisten Betriebssystemen an der Dateierweiterung erkannt. Die am häufigsten anzutreffende Erweiterung von ausführbaren Dateien unter Windows und DOS ist *.exe*. Unter Unix und Unix-Derivaten wie Linux gibt es für ausführbare Dateien keine vorgeschriebene Erweiterung, die meisten Programme verzichten daher ganz auf eine Erweiterung.

Auch die Resultate Ihrer zukünftigen Bemühungen werden sich vor allem in *.exe*-Dateien zeigen (vorausgesetzt, Sie arbeiten mit einem Windows-Betriebssystem).

Wenn also Ihr Computer auf eine Datei mit der Erweiterung *.exe* trifft, wird er versuchen, den Inhalt dieser Datei als Befehlsfolge zu verstehen und auszuführen. Wenn dieser Versuch misslingt, wird er dies mit einer Fehlermeldung kundtun.

### Hinweis

Genau genommen ist es der Prozessor Ihres Computers, der die Datei zu interpretieren versucht, und der Mittler zwischen Programmdatei und Prozessor ist das Betriebssystem. Mit anderen Worten, das Betriebssystem reicht *.exe*-Dateien an den Prozessor des Computers zur Ausführung weiter.

Was hier am Beispiel von *.exe*-Dateien geschildert wird, gilt natürlich in Abhängigkeit vom Betriebssystem ebenso für bestimmte andere Dateierweiterungen – wie oben erwähnt.

Angenommen, Sie würden eine Textdatei beliebigen (!) Inhalts editieren und diese mit der Erweiterung *.exe* speichern. Auf einen entsprechenden Startbefehl hin würde Ihr Computer auch diese Datei als Programmdatei ansehen. Bleibt die Hoffnung, dass in diesem Fall alle Interpretationsversuche fehlschlagen, andernfalls würde das nicht vorhersehbare Aktionen zur Folge haben. Es verhält sich dann gewissermaßen so, als ob ein

Spanier einem Italiener in der irrigen Annahme, es handle sich um einen Landsmann, Anweisungen erteilt und dabei Worte wählt, die auch im Italienischen vorkommen, nur eben in ganz anderer Bedeutung.

### Hinweis

Aus dem genannten Grund handelt es sich um keinen wirklich ernst gemeinten Vorschlag. Möglicherweise interpretiert Ihr Computer eine Zeichenfolge als »Lösche ...«. Ich empfehle Ihnen also, von solchen Experimenten abzusehen und mir auch so zu glauben.

Eine Programmdatei ist somit nur dann ausführbar (und verdient auch nur in diesem Fall, so genannt zu werden), wenn die in ihr enthaltenen Befehle vom Computer verstanden werden können. Das setzt voraus, dass sie in der Sprache des Computers, der so genannten Maschinensprache, abgefasst sind.

Befehle in Maschinencode sind binärkodiert, das heißt, sie liegen als Folge von Nullen und Einsen vor:

```
01101010100101010101110001011100
10101010110101100101100111101011
00101000110101110101100100101010
```

Der offensichtliche Nachteil von Maschinencode liegt in seiner schlechten Lesbarkeit. Er lässt sich kaum sinnvoll strukturieren und ist somit nur schwer nachvollziehbar. Hinzu kommt, dass sich Maschinenbefehle nur schwer aus dem Gedächtnis reproduzieren lassen. Alles zusammengenommen gestaltet sich die Entwicklung eines Programms auf diese Weise sehr schwierig, ganz zu schweigen von einer eventuellen späteren Überarbeitung.

Wie viel erfreulicher wäre es, wenn ein Computer etwa folgendermaßen kodierte Anweisungen akzeptieren würde:

```
Nimm eine Eingabe des Anwenders entgegen.
Prüfe, ob die Eingabe als Zahlenwert interpretierbar ist.
Falls nicht, schreibe die Zeichenfolge "Falsche Eingabe" auf den Bildschirm.
...
```

Nun, ganz so einfach ist es leider nicht (andernfalls würden Sie dieses Buch nicht lesen). Allerdings lassen sich Wortschatz und Grammatik moderner Programmiersprachen wie C++ etwa auf einem Niveau zwischen Maschinensprache und dieser gedachten Variante ansiedeln, was die Sache schon sehr viel angenehmer macht.

### Hinweis

Es sei angemerkt, dass nahezu alle Programmiersprachen sich angloamerikanischer Begriffe bedienen.

In gewisser Weise erscheint eine Programmiersprache wie C++ sogar eleganter und unkomplizierter als jede Umgangssprache, da sie über geeignete Schlüsselwörter und Zeichen verfügt, mit denen sich extrem präzise das formulieren lässt, was man erreichen möchte – Konfusionen wie in der Alltagssprache sind ausgeschlossen.

## 1.2 Entwicklungsschritte zum ausführbaren Computerprogramm

Welche Verbindung besteht nun zwischen der unvermeidbaren Maschinsprache (Nur diese versteht ja der Computer!) und der Programmiersprache C++? Um dies aufzuzeigen, wollen wir in aller Kürze die Entwicklungsstationen zum fertigen Programm skizzieren, und zwar von Anfang an.

### 1.2.1 Aufsetzen des Quellcodes

Ursprung jeder fertigen Anwendung ist der so genannte Quellcode. Dieser besteht aus reinem Text, setzt sich also aus Buchstaben, Zahlen und anderen Zeichen zusammen, die Ihnen von der Tastatur Ihres PCs her bekannt sein dürften. Der Quellcode (Englisch »source code«) wird daher auch als »Quelltext« bezeichnet. Vor allem, wenn man nur einen Teil des Quellcodes meint, spricht man auch kurz von »Code«.

Die Hauptaufgabe für Sie als Programmierer besteht im Erstellen dieses Quellcodes. Dazu müssen Sie diesen, wie eingangs erwähnt, in einen Editor eingeben, wobei Sie den Vorgaben der verwendeten Programmiersprache unterworfen sind. Eben das Erlernen dieser Vorgaben – oder positiv ausgedrückt: Möglichkeiten – der Programmiersprache C++ bildet das Thema dieses Buches.

#### Hinweis

Die Bezeichnung »Programm« wird – je nach Sichtweise – nicht nur für die fertige Anwendung, sondern mitunter auch für den – noch nicht lauffähigen – Quellcode verwendet. Während der Anwender unter »Programm« in der Regel die ausführbare Anwendung versteht, meint der Programmierer mit dieser Bezeichnung häufig in erster Linie den von ihm verfassten Quellcode.

### 1.2.2 Kompilierung

Nichtsdestotrotz ist es letzten Endes erforderlich, dass die ausführbare Programmdatei in Maschinsprache vorliegt. Es muss also eine binärkodierte Entsprechung des C++-Quellcodes hergestellt werden. Das heißt nichts anderes, als dass der Quelltext in Maschinsprache zu übersetzen ist. Dies besorgt eine Software, der so genannte Compiler.

Der Compiler übersetzt (kompiliert) den Quellcode als Ganzes in Maschinsprache und speichert das Endprodukt in einer eigenen Datei (eben unter Windows mit der Dateierweiterung *.exe*). Dies geschieht in verhältnismäßig kurzer Zeit und ohne Ihr wei-

teres Zutun. Da es sich beim Compiler ebenfalls um ein Computerprogramm handelt, müssen Sie dieses lediglich starten.

Allerdings sind auch Compiler keine Multitalente. Sie unterscheiden sich in Abhängigkeit sowohl von der Programmiersprache als auch vom System, für das die zukünftige Anwendung bestimmt ist. So ist etwa ein C++-Compiler nicht in der Lage, einen in einer anderen Programmiersprache wie z.B. Pascal geschriebenen Quellcode zu übersetzen. Dazu bedarf es eines Compilers, der die Sprache Pascal versteht, wie z.B. Turbo Pascal. Da wir ja C++-Programme schreiben wollen, liegen wir jedoch mit einem C++-Compiler völlig richtig.

Aber auch ein unter Windows kompiliertes C++-Programm ist z.B. auf Unix/Linux-Systemen grundsätzlich nicht lauffähig. Wenn Sie also ein entsprechendes Linux-Programm erzeugen möchten, dann müssen Sie denselben Quelltext mit einem für diese Plattform vorgesehenen C++-Compiler erneut kompilieren.

### Hinweis

Dass dies überhaupt möglich ist, ist keineswegs selbstverständlich. In den Urzeiten der Programmierung war es nicht die Regel, dass ein in einer bestimmten Programmiersprache verfasster Quelltext auf mehreren Betriebssystemen kompilierbar war. Wenn ein Quelltext auf mehreren Systemen kompiliert werden kann, sagt man, er (bzw. die entsprechende Programmiersprache) ist portierbar. Die Portierbarkeit von Programmiersprachen hat sich erst in den Achtzigerjahren zum Standard entwickelt.

Natürlich verrichtet der Compiler seine Arbeit nur dann, wenn der Quelltext syntaktisch korrekt ist, ansonsten quittiert er den Kompilierversuch mit entsprechenden Fehlermeldungen. In diesem Fall müssen Sie den Quellcode nochmals überarbeiten und den Kompilervorgang erneut in Gang setzen.

Nach erfolgreichem Übersetzungsvorgang liegt das Programm nunmehr in seiner ausführbaren Form vor (Abbildung 1.1).

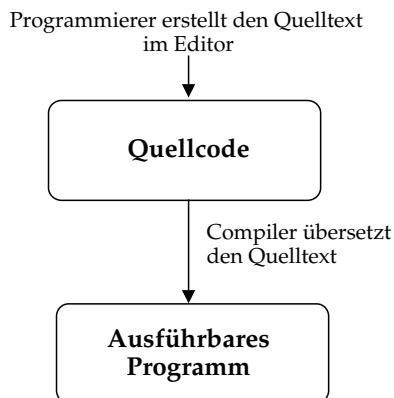


Abbildung 1.1: Der Weg zum lauffähigen Programm

### Hinweis

Selbstverständlich durchläuft zumindest jede professionelle Anwendung in der Regel eine mehr oder weniger umfangreiche Testphase, bevor die Entwicklung als abgeschlossen betrachtet werden kann. Diese haben wir bei unseren Betrachtungen großzügig außer Acht gelassen.

Beachten Sie, dass die Quellcodedatei beim Übersetzungsvorgang unverändert erhalten bleibt. Es liegen nunmehr als Ergebnis der Programmierarbeit zwei Dateien vor: die Quellcodedatei – oder kurz »Quelldatei« – und die ausführbare Programmdatei (.exe-Datei). Streng genommen sind es sogar drei Dateien, da, sozusagen als Zwischenprodukt, zusätzlich eine so genannte Objektdatei (Endung .obj) entsteht. Diese soll uns aber zum jetzigen Zeitpunkt nicht weiter interessieren.

### 1.2.3 Präprozessor und Linker

Genau genommen leistet der eigentliche C++-Compiler nur einen – wenn auch den größten – Teil des Übersetzungsvorgangs, an dem außerdem noch Präprozessor und Linker beteiligt sind (das Wort »Präprozessor« soll Sie nicht irritieren; wie beim Linker und dem eigentlichen Compiler handelt es sich auch beim Präprozessor um Software – diese hat mit dem Prozessor Ihres Computers nichts zu tun).

### Hinweis

Da der Compiler bei der Übersetzung des Quelltextes die Hauptaufgabe leistet, ist es allgemein üblich, die ganze Software für den Übersetzungsvorgang als »Compiler« zu bezeichnen. Analog dazu nennt man den vollständigen Übersetzungsvorgang Kompilierung.

Da ein Compiler (also die Gesamtheit der für den Übersetzungsvorgang benötigten Software) in modernen integrierten Entwicklungsumgebungen (»IDE«, dazu gleich mehr) immer enthalten ist, verwendet man das Wort »Compiler« mitunter auch synonym für die ganze integrierte Entwicklungsumgebung. Der Begriff »Compiler« kommt somit in dreifacher Bedeutung vor.

Der Kompiliervorgang stellt sich für den Programmierer in der Wahrnehmung als ein einziger Schritt dar. Wenn Sie z.B. mit Visual Studio oder der Visual C++ 2005 Express Edition arbeiten, erfolgt die Kompilierung, also das Erzeugen einer .exe-Datei, auf einen einzigen Mausklick hin. Bei dem C++-Compiler von Borland, den Sie auf der Buch-CD finden, geben Sie dazu einen Befehl auf der Konsole ein. Präprozessor und Linker treten so gesehen nach außen hin nicht sichtbar in Erscheinung<sup>1</sup>. Dennoch soll ihre Wirkungsweise hier kurz skizziert werden. Es sei darauf hingewiesen, dass es für Sie zum jetzigen Zeitpunkt weder erforderlich noch möglich ist (sofern Sie zu den Programmierestei-

---

1 In den meisten Compiler ist der Präprozessor direkt integriert, tritt also gar nicht mehr als eigenständiges Programm auf. Der Linker hat sich dagegen seine Eigenständigkeit bewahrt. Er wird bei der Programmerstellung üblicherweise automatisch vom Compiler aufgerufen.

gern gehören), das Folgende in allen Einzelheiten zu verstehen. Genauer werden Sie erfahren, wenn wir uns mit Funktionen beschäftigen.

Die Übersetzung von C++-Programmen (mit »Programm« ist hier natürlich der Quelltext gemeint) erfolgt in drei internen Schritten:

- Der Präprozessor entfernt eventuelle Kommentare aus dem Quelltext und führt bestimmte Textersetzungen durch.
- Der eigentliche Compiler übersetzt den so aufbereiteten Quelltext in Maschinsprache. Als Ergebnis entstehen eine oder mehrere so genannte Objektdateien mit der Erweiterung *.obj*.
- Im letzten Übersetzungsschritt fügt der Linker Code aus den Bibliotheken ein und bindet die *.obj*-Dateien, soweit es sich um mehrere handelt, zu einer ausführbaren Datei zusammen.

### Hinweis

Wie oben angedeutet, ist es auch möglich, eine ausführbare Datei aus mehreren Quellcode-Dateien zu kompilieren. In diesem Fall erzeugt der Compiler für jede Quelldatei eine *.obj*-Datei (wie das geht, erfahren Sie ebenfalls in Kapitel 20 »Funktionen«).

## 1.3 Was versteht man unter einer IDE?

Wie Sie gerade erfahren haben, ist es für die Programmerstellung notwendig, Text in einen Editor einzugeben und diesen dann kompilieren zu lassen. Dazu kann jeder beliebige Texteditor benutzt und für den Übersetzungsvorgang ein externer Compiler herangezogen werden.

In modernen Entwicklungsumgebungen ist die Software zum Übersetzen des Quellcodes genauso wie ein mit nützlichen Funktionen ausgestatteter Editor als fester Bestandteil in ein umfassendes Softwareprogramm eingebunden. Dort wird der Compiler meist über einen entsprechenden Befehl in der Menüleiste oder noch einfacher mit einem Klick auf ein Symbol gestartet. Zudem werden in der Regel eine Reihe weiterer Hilfsmittel für die Programmentwicklung bereitgestellt.

Weil dort eben alle Programmierwerkzeuge unter einer Bedienoberfläche zusammengefasst sind, nennt man ein solches System »integrierte Entwicklungsumgebung« oder kurz IDE (für Integrated Development Environment).

Eine beliebte IDE für C++-Programme ist z.B. das Visual Studio von Microsoft. Dieses fungiert gleichzeitig als Entwicklungsumgebung für weitere Programmiersprachen wie z.B. Visual Basic und C#. Soweit nur die Features zur Entwicklung von C++-Programmen gemeint sind, spricht man von Visual C++ bzw. vom Visual C++-Compiler.

### Hinweis

Bei der bereits erwähnten Visual C++ 2005 Express Edition handelt es sich gewissermaßen um eine Tochter des Visual Studio bzw. von Visual C++. Sie ist voll funktionsfähig und obendrein kostenfrei erhältlich.

Der Name »Visual« rührt daher, dass zusätzliche Funktionen zur Programmierung von grafischen Benutzeroberflächen bereitgestellt werden (visual – zu Deutsch »visuell«). Diese sind jedoch compilerspezifisch, gehören also nicht zum Standardumfang der Programmiersprache C++. Das heißt, ein Programm (gemeint ist der Quellcode), welches diese (compilerspezifischen) Funktionen verwendet, kann nicht auf einem anderen Compiler übersetzt werden – auch wenn es sich dabei um einen anderen C++-Compiler handelt.

Es sei hier ausdrücklich darauf hingewiesen, dass C++ ein Sprachstandard, also keine Entwicklungsumgebung, ist. Eben dieser Sprachstandard bildet das Thema dieses Buches. Wir werden in den Programmierbeispielen daher nur auf Features zurückgreifen, die von diesem Standard (dem so genannten ANSI/ISO-Standard) unterstützt werden.

### Hinweis

Mit einer Ausnahme: Im letzten Teil dieses Buches werde ich Sie anhand einer mehr oder weniger umfangreichen Beispielanwendung mit der Programmierung von grafischen Benutzeroberflächen bekannt machen. Dieser Teil ist gewissermaßen als Bonus zu verstehen.


Prinzipiell können Sie also mit jedem beliebigen Compiler arbeiten, solange dieser den ANSI/ISO-Sprachstandard unterstützt. Im Übrigen wurden die Beispiele im Buch mit verschiedenen C++-Compilern getestet, u.a. mit Bloodshed, Visual C++ und der Kommandozeilenversion des Borland C++-Compilers. Ein Kommandozeilen-Compiler wird durch Eingabe eines Startbefehls in der Konsole in Gang gesetzt, ist also nicht Teil einer IDE.

## 1.4 Was sind Konsolenprogramme?

Das wirft die Frage auf, was unter einer Konsole zu verstehen ist. Zu alten DOS-Zeiten lief die Kommunikation zwischen Anwender und Computer allein über die Konsole, die den ganzen Bildschirm einnahm. Oder anders ausgedrückt: Der Bildschirm war die Konsole.

Der Computer nahm Eingaben des Benutzers (Befehle an das Betriebssystem, aber auch Startbefehle für Computerprogramme) an einer bestimmten Stelle des Bildschirms entgegen. In der Regel forderte dort das Blinken eines Cursors zur Eingabe auf. Diese Stelle wurde eben »Eingabeaufforderung«, »Befehlszeile« oder kurz »Prompt« genannt.



Falls am Prompt der Startbefehl für ein Programm eingegeben wurde (durch Eingabe der dafür notwendigen Zeichenfolge über die Tastatur und anschließendes Betätigen der -Taste), übernahm dieses sozusagen die Konsole. Das heißt, es nutzte die Konsole – damals also den ganzen Bildschirm – für Mitteilungen an den Benutzer («Geben Sie bitte ... ein«, «Die von Ihnen zu zahlende Einkommensteuer beträgt ...») und um Daten vom Benutzer entgegenzunehmen.

Heutzutage verläuft die Kommunikation zwischen Anwender und Programm meist über grafische Benutzeroberflächen. Dies gilt auch für die Interaktion mit dem Betriebssystem. So bietet Windows verschiedene grafische Schnittstellen zum Starten von Computerprogrammen an: Eintrag im Startmenü, Symbol auf dem Desktop, Auflistung von ausführbaren Dateien im Windows-Explorer.

### GUI

GUI ist die Abkürzung für Graphical User Interface, zu Deutsch grafische Benutzeroberfläche. Darunter versteht sich die Gesamtheit aller Oberflächenelemente wie Fenster, Menüleisten, Schaltflächen etc., welche für den Dialog mit dem Benutzer zur Verfügung stehen.

Das ändert aber nichts an der Tatsache, dass die Konsole nach wie vor zur Verfügung steht. Nur nimmt sie nicht mehr wie früher den ganzen Bildschirm ein. Außerdem erscheint sie nicht sofort beim Start des Computers, sondern ist ebenfalls über grafische Schnittstellen erreichbar.

Unter Windows-Betriebssystemen erreichen Sie die Konsole über START/PROGRAMME/(ZUBEHÖR/)EINGABEAUFFORDERUNG bzw. START/AUSFÜHREN.... Bei der Variante START/AUSFÜHREN... müssen Sie in dem dann angezeigten Dialogfeld zunächst `cmd` eingeben und mit einem Klick auf die Schaltfläche OK bestätigen.

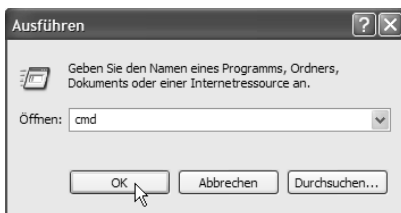


Abbildung 1.2: Dialog zum Aufrufen der Konsole

### Hinweis

Weitere Bezeichnungen für die Konsole sind neben »Eingabeaufforderung«, »Befehlszeile« und »Prompt« auch »DOS-Fenster« und »DOS-Box«. Unter Linux heißt die Konsole auch »Terminal«.

Eine Anwendung wie beispielsweise das Tabellenkalkulationsprogramm Excel ließe sich auch über die Konsole starten. Falls der Verzeichnispfad zur zugehörigen Programmdatei

tei `C:\Programme\Microsoft Office\OFFICE11` lautet, geben Sie an der Eingabeaufforderung nacheinander die Befehle

```
cd Programme
cd Microsoft Office
cd OFFICE11
```

ein, um in das Verzeichnis zu gelangen, in dem sich die ausführbare Datei `excel.exe` befindet. Diese starten Sie dann mit dem Befehl `excel` (Eingabe des Dateinamens ohne Erweiterung).

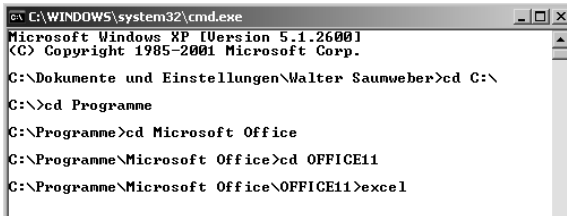


Abbildung 1.3: Starten von Excel über die Konsole

### Hinweis

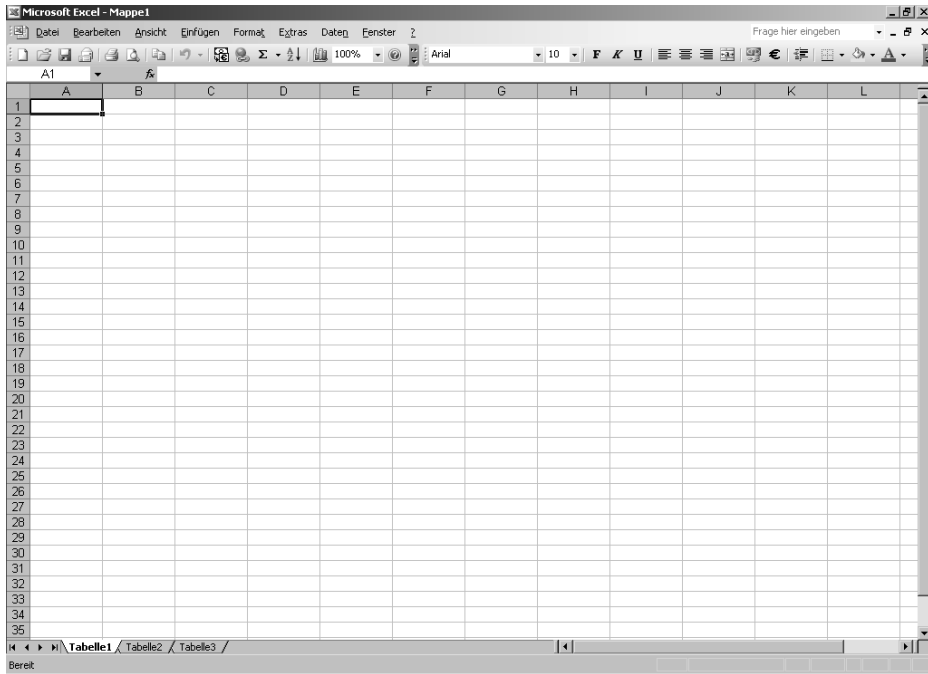
Es sei erneut darauf hingewiesen, dass der Startbefehl für ausführbare Programme vom verwendeten Betriebssystem abhängt. Unter Linux beispielsweise wird eine Datei mit dem Namen `xy.out` auch mit `xy.out` gestartet, also mit Angabe des vollständigen Dateinamens (einschließlich Dateierweiterung).

Auch wenn es (wie jedes andere Programm) von der Konsole aus gestartet werden kann, ist Excel dennoch ein Windows-Programm, da es ja selbst eine grafische Benutzeroberfläche zur Verfügung stellt.

Vielmehr zeichnet sich ein Konsolenprogramm dadurch aus, dass es eben keine GUI zur Verfügung stellt, sondern für die Interaktion mit dem Anwender ausschließlich die Konsole benutzt. Dies gilt auch für die Kommandozeilenversion des Borland C++-Compilers (das Wort »Kommandozeile« steht hier ebenfalls für die Konsole). Wie Ihnen ja bekannt ist, handelt es sich auch bei einem Compiler um nichts anderes als um ein ausführbares Computerprogramm.

Entscheidend ist also die Bereitstellung einer grafischen Benutzeroberfläche seitens des Programms. Von wo aus ein Programm gestartet wird – über eine grafische Benutzeroberfläche oder durch Eingabe eines Startbefehls in der Konsole –, spielt für die Einordnung als Windows- bzw. Konsolenprogramm keine Rolle.

Abgesehen vom letzten Teil des Buches, werden wir uns bei der Programmierung auf Konsolenprogramme beschränken. Dies mag Sie zunächst etwas enttäuschen, ist aber tatsächlich von Vorteil. Ziel des Buches ist es schließlich, den Standard der Programmiersprache C++ zu erlernen. Die Konzentration auf andere Themen wäre dabei nur hinderlich.



**Abbildung 1.4:** Die GUI von Excel

In diesem Zusammenhang sei darauf hingewiesen, dass die Kenntnisse, die Ihnen im Weiteren vermittelt werden (vor allem auf dem Gebiet der objektorientierten Programmierung), für ein tiefer gehendes Verständnis der Windows-Programmierung unabdingbar sind.

Weitere Gründe, um aus didaktischen Erwägungen von einer Beschäftigung mit der GUI-Programmierung vorerst abzusehen, sind:

- Windows-Programme besitzen den Nachteil der Plattformabhängigkeit, da sie an den auf einem System integrierten Window-Manager gebunden sind. Der Quelltext von Windows-Programmen ist also nicht portierbar (zur Portierbarkeit siehe oben). Mit anderen Worten: Jede IDE stellt spezielle Funktionen zur Programmierung von grafischen Benutzerschnittstellen zur Verfügung. Dagegen wird eine Funktion, die zum Standardsprachumfang von C++ gehört, von jedem Compiler unterstützt (ist das nicht der Fall, so handelt es sich wahrscheinlich um einen veralteten Compiler, und es empfiehlt sich der Umstieg auf eine neuere Compilerversion).
- Die Programmierung von GUIs ist in der Regel sehr aufwändig. Verschiedene IDEs, darunter auch das Visual Studio, können allerdings den Quellcode mehr oder weniger automatisch erzeugen – zum Teil erhält man dann für bestimmte Aufgaben mehrere Seiten entsprechend komplexen Codes.

Von den letztgenannten Möglichkeiten (die auch Visual C++ bietet) werden wir aber, wie gesagt, erst in den Bonuskapiteln Gebrauch machen. Selbstverständlich bleibt es Ihnen freigestellt, sich nach dem Lesen dieses Buches entsprechende Lektüre zu besorgen (z.B. spezielle Literatur über Visual Studio). Sie haben dann die besten Voraussetzungen dazu.

Des Weiteren sei darauf hingewiesen, dass grafische Benutzerschnittstellen zwar erhebliche Verbesserungen des Bedienkomforts mit sich bringen, die Leistungsfähigkeit von Programmen jedoch nicht erhöhen. Überdies sind auch heute noch eine Reihe von professionellen Programmen im Einsatz, bei denen grafische Schnittstellen keine Rolle spielen.

# 2

## Welchen Compiler sollten Sie verwenden?

Bevor Sie Ihr erstes Programm schreiben, werden wir zunächst auf die Frage der benötigten Software eingehen. Einerseits ist natürlich auch die Wahl des Compilers, wie vieles andere, eine Frage des persönlichen Geschmacks. Auf der anderen Seite spielt es eine Rolle, ob Sie bereits Programmiererfahrung – in C++ oder möglicherweise in einer anderen Programmiersprache – mitbringen. Gerade wenn Sie zu den Programmierneulingen gehören, sollten Sie besonders darauf achten, dass der Compiler, den Sie verwenden, den ANSI/ISO-C++-Sprachstandard in vollem Umfang unterstützt. Des Weiteren rate ich Ihnen in diesem Fall, fürs Erste einen einfachen Kommandozeilen-Compiler zu verwenden. Natürlich bietet eine aufwändige Entwicklungsumgebung wie z.B. das Visual Studio von Microsoft viele Vorteile. Diese werden Sie aber erst richtig nutzen können, wenn Sie schon einiges Verständnis und Erfahrung in der Programmierung besitzen. Dem Anfänger wird die Komplexität einer modernen IDE jedoch eher zum Nachteil gereichen.

### 2.1 Borland C++-Compiler

Lesern, die unter einem Windows-Betriebssystem arbeiten, sei die Kommandozeilen-Version des C++-Compilers von Borland empfohlen. Sie finden ihn in der zur Zeit der Drucklegung aktuellen Version 5.5.1 auf der Buch-CD im Verzeichnis *Borland-Compiler* (Dateiname: *freecommandLinetools.exe*). Ansonsten können Sie den C++-Compiler über die Website <http://www.codegear.com> – klicken Sie hierzu auf den Link **DOWNLOADS**, dann auf **C++BUILDER** und anschließend auf **COMPILER** – oder gleich unter <http://www.codegear.com/tabid/139/Default.aspx>, Link **COMPILER**, kostenlos herunterladen. Der Kommandozeilen-Compiler von Borland ist zuverlässig und leistungsfähig. Außerdem unterstützt er den ANSI/ISO-C++-Sprachstandard in vollem Umfang. Die Installation wird weiter unten beschrieben. Insbesondere werden wir die einzelnen Schritte aufzeigen, die für das Übersetzen unseres ersten Programms speziell mit diesem Compiler nötig sind.

Für Leser, die mit einem anderen Compiler arbeiten, ist es leider unerlässlich, sich mit dessen Bedienung bis zu einem gewissen Grad selbständig vertraut zu machen. Bemühen Sie gegebenenfalls die Dokumentation Ihres Compilers. Damit sich diese Lesergruppe nicht benachteiligt fühlt, sei noch einmal vermerkt, dass der Fokus dieses Buches auf dem Erlernen der Programmiersprache C++ liegt, nicht auf der Beschreibung einer bestimmten IDE. Zudem sind die Anleitungen, die Sie in Kapitel 4 im Zusammenhang mit unserem ersten Programm in Bezug auf den Borland-Compiler erhalten, mehr oder weniger auch auf andere Compiler anwendbar. Mit Beginn des zweiten Teils wird eine gewisse Vertrautheit mit der von Ihnen verwendeten Software vorausgesetzt, sodass in dieser Hinsicht alle Leser gleich behandelt werden.

Im Übrigen ist es nicht immer zwingend notwendig, dass Sie am Computer sitzen, wenn Sie dieses Buch lesen (manche Leser mögen dies generell als unangenehm empfinden). Die meisten Beispiele sollten Sie auch ohne direkten Rechnerkontakt gut verstehen können. Praktische Übung empfiehlt sich natürlich in jedem Fall.

## 2.2 Wie finde ich einen passenden Compiler?

Diese Frage stellt sich natürlich nur, wenn Sie nicht mit dem oben genannten Borland-Compiler arbeiten können oder aus einem bestimmten Grund nicht wollen. Leider ist der Borland C++-Compiler nicht für Linux-Betriebssysteme verfügbar. In aller Regel ist jedoch dort ein C++-Kommandozeilen-Compiler (GCC) bereits vorinstalliert, sodass Sie diesen verwenden können. Er gewährleistet ebenfalls eine ausreichende Unterstützung des ANSI/ISO-C++-Sprachstandards. Im Zuge der Ausführungen zu unserem ersten Programm werden wir auch zum GCC-Compiler einige Hinweise geben.

### **Tipp**

Noch ein Tipp, der möglicherweise zu einem späteren Zeitpunkt für Sie von Interesse sein mag: DJGPP, eine sehr aufwändig zu installierende, aber sehr leistungsfähige IDE, wird von der Firma delorie software auf der Website

*<http://www.delorie.com/djgpp>*

kostenlos zum Download angeboten. Die Informationen sind allerdings in Englisch gehalten, und es ist trotz umfangreicher Dokumentation nicht unbedingt leicht, sich zurechtzufinden.

Die Standardversion von DJGPP ist auch auf der deutschsprachigen Website

*<http://c.theflow.de/djgpp.htm>*

erhältlich. Dort finden Sie auch eine ausführliche Installationsbeschreibung und weitere interessante Links zu Tutorials, Compilern etc.

Leider ist uns zum Zeitpunkt der Drucklegung dieses Buches kein frei erhältlicher C++-Compiler für den Macintosh bekannt. Möglicherweise werden Sie auf der Suche nach einem passenden C++-Compiler für den Mac jetzt oder später auf den Entwickler-Seiten von Apple fündig (<http://developer.apple.com/de>). Ansonsten bleibt Lesern, die mit einem Macintosh arbeiten, nichts anderes übrig, als gegebenenfalls einen passenden Compiler käuflich zu erwerben (z.B. Symantec C++) und sich mit der Bedienung hinreichend vertraut zu machen. Aber wie schon gesagt, Thema dieses Buches ist die Programmiersprache C++ und nicht die Beschreibung eines Compilers.

### Tipp

Falls Darlegungen hinsichtlich der Editierung des Quelltextes, dessen Kompilierung und Ausführung für Sie nicht zutreffen (da compiler- oder betriebssystemabhängig), nehmen Sie dies zum Anlass, einen entsprechenden Suchbegriff in Ihrer Compiler-hilfe zu verwenden, um die gewünschte Information zu erhalten.

## 2.3 Installation des Borland-Compilers

Beginnen wir nun mit der Installation des Borland C++-Compilers:

1. Legen Sie die Buch-CD in Ihr CD-ROM-Laufwerk ein und starten Sie im Windows-Explorer die Datei *freecommandLinetools.exe*. Sie befindet sich im Ordner *Borland-Compiler*.

Die Willkommenseite zeigt Ihnen die aktuelle Compiler-Version. Außerdem die Adresse, wo Sie Ihren C++-Compiler bei Borland registrieren lassen können.



Abbildung 2.1: Borland C++ Compiler - Setup

2. Bestätigen Sie den Begrüßungsbildschirm mit **WEITER >**.
3. Falls keine Gründe dagegen sprechen, belassen Sie es hinsichtlich des Zielverzeichnisses bei der Voreinstellung *C:\Borland\BCC55*. Andernfalls wählen Sie über die Schaltfläche **BROWSE...** ein anderes Verzeichnis. Klicken Sie jetzt auf **FERTIG STELLEN**.



Abbildung 2.2: Zustimmung zum Lizenzvertrag

Danach entpackt das Setup alle notwendigen Dateien in das angegebene Verzeichnis. Damit haben Sie bereits alles getan, zumindest was die eigentliche Installation angeht. Allerdings sind noch wenige Schritte notwendig, damit Ihr Compiler funktionsfähig ist und Sie ihn auch komfortabel nutzen können.

4. Richten Sie für Ihre C++-Programme ein eigenes Unterverzeichnis ein.

Nennen Sie es einfach *MeineProgramme*. Wir gehen im Folgenden davon aus, dass es sich unterhalb von *C:\Borland* befindet und Sie im zweiten Installationsschritt die Vorgabe *C:\Borland\BCC55* übernommen haben. Dann ergibt sich auf Ihrer Festplatte folgende Verzeichnisstruktur.

- Die oberste Ebene stellt das Verzeichnis *C:\Borland* dar. Im Unterverzeichnis *BCC55* – der Pfad lautet *C:\Borland\BCC55* – befinden sich alle Dateien des Borland-Compilers.
- Die Datei *bcc32.exe*, die gestartet werden muss, um den Compiler in Gang zu setzen, befindet sich wiederum im Unterverzeichnis *Bin* (Pfad: *C:\Borland\BCC55\Bin*).
- Als Sammelort für unsere C++-Programme ist das von Hand angelegte Unterverzeichnis *MeineProgramme* vorgesehen (Pfad: *C:\Borland\MeineProgramme*).

### Hinweis

Natürlich müssen Sie die oben genannten Daten im Weiteren auf Ihre Umgebung übertragen. Anpassungen sind vor allem dann notwendig, wenn Sie mit einem anderen Compiler arbeiten oder ein anderes Zielverzeichnis bei der Installation bestimmt haben.

5. Fügen Sie den Pfad zum Bin-Verzeichnis (*C:\Borland\BCC55\Bin*) Ihrer Umgebungsvariablen *Path* hinzu.

Umgebungsvariablen können Sie in den Systemeigenschaften im Register **ERWEITERT** bearbeiten (siehe Kasten »So bearbeiten Sie Ihre Path-Umgebungsvariable«).



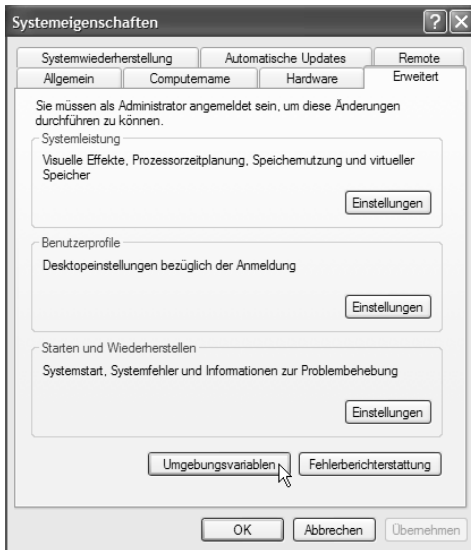


Abbildung 2.3: Umgebungsvariablen bearbeiten unter Windows XP

### So bearbeiten Sie Ihre Path-Umgebungsvariable

Klicken Sie auf Ihrem Desktop mit der rechten Maustaste auf das Arbeitsplatz-Symbol und dann im Kontextmenü ganz unten auf den Eintrag EIGENSCHAFTEN. Alternativ können Sie in der Systemsteuerung den Menüpunkt SYSTEM wählen. Im Register ERWEITERT klicken Sie auf die Schaltfläche UMGEBUNGSVARIABLEN. Wählen Sie dann im Abschnitt SYSTEMVARIABLEN die Variable *Path* aus und klicken Sie auf BEARBEITEN. Fügen Sie im unteren Textfeld neben WERT DER VARIABLEN:, getrennt durch ein Semikolon, den Pfad zum Bin-Verzeichnis hinzu und bestätigen Sie mit OK. Die Beschreibung entspricht der Vorgehensweise unter Windows XP. Sie sollte sich aber bei anderen Windows-Versionen nicht wesentlich unterscheiden.



Abbildung 2.4: Path-Variablen um den Pfad zum Bin-Verzeichnis ergänzt

Danach kann die Datei *bcc32.exe* ohne Pfadangabe aus jedem Verzeichnis heraus aufgerufen werden. Sie können den Compiler dann direkt aus dem Verzeichnis *MeineProgramme* starten, in dem sich später alle Ihre Quelldateien befinden werden. Dies ist recht komfortabel, da Sie auf diese Weise während der Entwicklung nicht laufend das Verzeichnis wechseln müssen.

6. Editieren Sie zwei Dateien *bcc32.cfg* und *ilink32.cfg* wie folgt und legen Sie diese in Ihr Bin-Verzeichnis.

Datei *bcc32.cfg*:

```
-I"c:\Borland\Bcc55\include"  
-L"c:\Borland\Bcc55\lib"
```

Datei *ilink32.cfg*:

```
-L"c:\Borland\Bcc55\lib"
```

Wir gehen wiederum davon aus, dass Ihr Pfad zum BCC55-Verzeichnis *C:\Borland\BCC55* lautet. Falls dies nicht zutrifft, ändern Sie die Pfadangaben entsprechend ab.

### **Tipp**

Beide Dateien finden Sie fertig auf der Buch-CD. Insofern Ihr Pfad zum BCC55-Verzeichnis *C:\Borland\BCC5* lautet, brauchen Sie *bcc32.cfg* und *ilink32.cfg* nur zu kopieren. Andernfalls könnten Sie zunächst die Pfadangaben korrigieren und danach die Dateien ins Bin-Verzeichnis speichern.

Was es mit den beiden Dateien auf sich hat, braucht Sie zum jetzigen Zeitpunkt nicht weiter zu interessieren. Hier nur so viel: Mit den Angaben in *bcc32.cfg* setzen Sie die Compiler-Optionen für die Include- und Lib-Dateien. Die Zeile in *ilink32.cfg* definiert für den Linker den Pfad zu den Bibliotheken. Was das genau bedeutet, erfahren Sie, wenn wir uns in Kapitel 20 mit Funktionen beschäftigen.

Damit haben Sie Ihren Borland C++-Compiler vollständig eingerichtet.

# 3

## Ihr erstes C++-Programm

Nun ist es endlich so weit. Sie werden Ihr erstes kleines Programm schreiben. Es soll sich darin erschöpfen, einen Text auf den Bildschirm auszugeben. Wir werden den Benutzer mit den Worten »Willkommen bei C++« begrüßen. Sie können aber auch jeden beliebigen anderen Text verwenden, der Ihnen gerade einfällt. Oder Sie schreiben das berühmte »Hallo Welt«-Programm. Berühmt deshalb, weil nahezu jeder Autor eines Programmierbuches es als Einführungsprogramm verwendet, seitdem es in den Siebzigerjahren in einem Lehrbuch von Kernighan und Ritchie über die Programmiersprache C stand.

Im folgenden Abschnitt 3.1 »Die Lösung vorweggenommen« werde ich Ihnen schon einmal den kompletten Quellcode vorstellen und wenn Sie wollen, können Sie diesen im Anschluss daran anhand der Ausführungen in Kapitel 4 »Kompilieren und Ausführen von C++-Programmen« kompilieren lassen und ausführen. In Abschnitt 3.2 »Von Anfang an« werden wir das vorweggenommene Ergebnis Schritt für Schritt entwickeln. Hier geht es vor allem darum, die ersten Grundlagen der Programmiersprache C++ kennen zu lernen.

### 3.1 Die Lösung vorweggenommen

Zur besseren Übersicht und für diejenigen, die nicht länger auf Ihr erstes Programm warten wollen, sei hier schon einmal der vollständige Quellcode dargestellt.

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

int main(void)
{
    cout << "Willkommen bei C++";
    cout << endl;
    return 0;
}
```

Beachten Sie, dass C++ »case-sensitiv« ist, das heißt, es unterscheidet zwischen Groß- und Kleinschreibung. Der Name `main` (dritte Zeile) darf also z.B. nicht als `Main` oder `MAIN` geschrieben werden. Das Gleiche gilt natürlich auch für `include`, `int`, `cout` usw.

**Achtung**

Falls Sie mit einem älteren Compilermodell arbeiten (was hoffentlich nicht der Fall ist), müssen Sie anstelle von

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

Folgendes schreiben:

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

Die Zeile `using namespace std;` entfällt in diesem Fall, `iostream` erhält die Endung `.h` (erste Zeile). Der gesamte Quelltext lautet dann

```
#include <iostream.h>

int main(void)
{
    cout << "Willkommen bei C++";
    cout << endl;
    return 0;
}
```

Es sei darauf hingewiesen, dass die meisten neueren Compiler beide Varianten zulassen. Die erste Variante (`iostream` ohne `.h` und `using namespace std;`) ist jedoch vorzuziehen, da sie dem aktuellen ANSI/ISO-Standard von C++ entspricht (mehr zum ANSI/ISO-Standard der Programmiersprache C++ erfahren Sie in Abschnitt 5.3.2).

Wie gesagt, bewirkt das Programm (nachdem es kompiliert und ausgeführt wurde) die Ausgabe von `Willkommen bei C++` auf den Bildschirm. Wenn Sie einen anderen Ausgabertext wünschen, geben Sie statt der Zeichenfolge `Willkommen bei C++` zwischen den beiden doppelten Anführungszeichen den gewünschten Text ein.

So würde das Programm

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

int main(void)
{
    cout << "Guten Tag, lieber Leser";
    cout << endl;
    return 0;
}
```

die Ausgabe

```
Guten Tag, lieber Leser
```

erzeugen.

**Achtung**

Vermeiden Sie einen Zeilenumbruch innerhalb des Ausgabetextes. Wenn dieser zu lang wird, dann benutzen Sie `cout` ein zweites Mal.

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

int main(void)
{
    cout << "Willkommen";
    cout << " bei C++";
    cout << endl;
    return 0;
}
```

Ausgabe ist ebenfalls

Willkommen bei C++

Denken Sie in diesem Fall daran, dass Leerzeichen, die innerhalb der Zeichenfolge stehen, ebenfalls auf den Bildschirm ausgegeben werden. Es hat also auch Auswirkungen auf die Bildschirmausgabe, wenn diese Leerzeichen fehlen.

So ist im obigen Quelltext ein Leerzeichen vor dem Wort `bei` in der zweiten Ausgabeanweisung notwendig, da sonst die Bildschirmausgabe

Willkommenbei C++

lauten würde.

**3.1.1 C++ und Zwischenraumzeichen**

C++ geht mit Leerzeichen zwar recht großzügig um (Genauerer dazu erfahren Sie in Abschnitt 7.4 »Leerräume«). Von daher spielt es für die Kompilierbarkeit oft keine Rolle, ob Sie einen – sagen wir zehnzeiligen – Quelltext auf mehrere Seiten verteilen oder auf ein, zwei Zeilen komprimieren.

Dennoch bereits hier der Hinweis: Gewöhnen Sie sich schon jetzt an, die geschweiften Klammern `{` und `}` jeweils in eine eigene Zeile zu schreiben und die Programmbefehle in den Zeilen, die zwischen einer öffnenden geschweiften Klammer und einer schließenden geschweiften Klammer stehen, um eine konstante Anzahl von Stellen einzurücken. Dies ist zwar nicht zwingend notwendig, gehört aber zum guten Programmierstil, da es die Lesbarkeit Ihrer Quelltexte erhöht. Sie werden schon sehr bald feststellen, dass sich ein Quellcode wie oben dargestellt sehr viel leichter nachvollziehen lässt – und sich damit später auch die Überarbeitung leichter gestaltet – als derselbe Quelltext etwa in der Form

```
#include <iostream>
using namespace std;int main(void){
    cout<<"Willkommen bei C++";cout<<endl;return 0;}
```

#### Hinweis

Bei sehr kleinen Quelltexten wie hier sind natürlich Ausnahmen denkbar. Für Einsteiger ist es aber zunächst wichtig, sich gar nicht erst einen schlechten Programmierstil anzugewöhnen.

Da C++, wie gesagt, in Bezug auf das Vorhanden- oder Nichtvorhandensein von Leerzeichen sehr tolerant ist, ist auch diese dreizeilige Fassung des Quelltextes mit demselben Ergebnis kompilierbar.

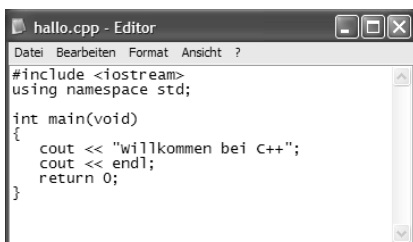
#### Hinweis

Allerdings muss die Präprozessor-Direktive `#include <iostream>` in einer separaten Zeile stehen (mit `#` eingeleitete Befehle richten sich an den Präprozessor – dazu später mehr), und es gilt nach wie vor, dass innerhalb der zwischen doppelten Anführungszeichen eingeschlossenen Zeichenfolge ("Willkommen bei C++") kein Zeilenumbruch erfolgen darf.

Aber schon bei diesem kleinen Programm wäre es für den weniger geübten C++-Programmierer nicht leicht, auf den ersten Blick zu erkennen, welche Teile des Quelltextes sinngemäß zusammengehören. Wenn Sie das jetzt noch nicht überzeugt (weil es sich zum einen um ein sehr kleines Programm handelt und weil auch die besser lesbare Ausführung für Sie noch nicht nachvollziehbar ist, da Sie die Bedeutung der einzelnen Programmbefehle noch nicht kennen), bitte ich Sie dennoch, meiner Empfehlung zu folgen und sich beide Quelltexte im Anschluss an die folgenden Ausführungen noch einmal anzuschauen.

Wir werden die Programmierbefehle innerhalb zweier geschweiften Klammern immer um genau drei Leerzeichen einrücken. Es bleibt Ihnen aber überlassen, für Ihre Programme eine andere Anzahl von Leerzeichen festzulegen, sie sollte aber im weiteren Quelltext konstant bleiben. Einige C++-Editoren, darunter der von Visual Studio, führen diese Einrückungen sogar automatisch durch.

Zum Abschluss noch einmal die lesbare Ausführung unseres Programms, eingegeben in Notepad, den Editor des Windows-Betriebssystems.



```
hallo.cpp - Editor
Datei Bearbeiten Format Ansicht ?
#include <iostream>
using namespace std;

int main(void)
{
    cout << "Willkommen bei C++";
    cout << endl;
    return 0;
}
```

Abbildung 3.1: Quelltext von hallo.cpp

Denjenigen, die es ganz eilig haben, sei es nun überlassen, einen Ausflug zum nächsten Kapitel (»Kompilieren und Ausführen von C++-Programmen«) zu machen und danach wieder hierher zurückzukehren. Wir werden im Folgenden so tun, als ob wir die Lösung unserer Aufgabe noch nicht kennen würden und uns diese Schritt für Schritt erarbeiten.

## 3.2 Von Anfang an

Wir müssen uns also überlegen, wie unser Quellcode auszusehen hat. C++ unterliegt wie jede andere Programmiersprache bestimmten Syntaxregeln und Vorgaben, die Sie einhalten müssen. Beachten Sie diese nicht, bricht der Compiler beim Übersetzen den Kompilervorgang ab und gibt eine oder mehrere Fehlermeldungen aus. Jedes einzelne Zeichen hat in C++ eine ganz bestimmte Bedeutung. C++-Compiler sind darin äußerst empfindlich, so großzügig C++ in Bezug auf Leer- und andere Zwischenraumzeichen auch sein mag.

### 3.2.1 Aufbau von C++-Programmen

Es stellt sich nun die Frage, wie ein C++-Programm aufgebaut ist (wir sprechen vom Quellcode). Die kleinste, sinngemäß zusammengehörige Einheit eines C++-Programms ist die so genannte Anweisung, auch Statement oder kurz Befehl genannt, mitunter auch Programm(ier)befehl. Ich hatte sie in Kapitel 1 bereits erwähnt (Abschnitt 1.1 »Was ist eigentlich ein Computerprogramm?«), ohne auf Näheres einzugehen.

Die Anweisung bildet praktisch die Ursache dafür, dass beim Programmlauf etwas geschieht, eine Aktion durchgeführt wird. Dabei hat jede einzelne Anweisung eine ganz bestimmte Wirkung:

- Es wird etwas auf den Bildschirm ausgegeben.
- Es wird eine Eingabe des Benutzers angenommen.
- Ein bestimmter Wert wird gespeichert.
- ...

Anweisungen sind damit sozusagen die Bausteine eines Programms. Je nachdem, was Ihr Programm leisten soll, wird sich daraus die Wahl der Anweisungen ergeben und auch deren Anordnung im Quelltext. Letzteres bestimmt, in welcher Reihenfolge die einzelnen Anweisungen auszuführen sind, welche Aktionen also beim Programmlauf zuerst stattfinden sollen.

#### **Hinweis**

Grundsätzlich werden die Anweisungen sequenziell ausgeführt, das heißt »von oben nach unten«, so wie sie im Quelltext stehen. Das bedeutet, die im Quellcode am weitesten oben stehende Anweisung wird zuerst ausgeführt, dann die unmittelbar folgende usw. Allerdings kann der Programmierer mittels Kontrollstrukturen in diesen Ablauf eingreifen. Wir werden darauf in Kapitel 16 »Verzweigungen« und Kapitel 17 »Wiederholungsanweisungen« ausführlich zu sprechen kommen.

Wir möchten den Benutzer unseres Programms über den Bildschirm begrüßen. Die dafür benötigte Anweisung lautet

```
cout << "...";
```

wobei zwischen den beiden Anführungszeichen der Ausgabertext einzusetzen ist.

```
cout << "Willkommen bei C++";
```

bewirkt also die Ausgabe von

```
Willkommen bei C++
```

#### Hinweis

`cout` ist genau genommen keine Anweisung, sondern ein Objekt, und `<<` ist der Ausgabeoperator. Beides zusammen bewirkt die Ausgabe von Daten auf das Standardausgabegerät (das »c« in `cout` steht für »character«, zu Deutsch »Zeichen«). So viel nur zu Ihrer Information.

#### Hinweis

Objekte im engeren Sinne sind Bestandteil der objektorientierten Programmierung, mit der wir uns im dritten Teil des Buches eingehend beschäftigen werden. Es sei hier darauf hingewiesen, dass wir bis dahin mit dem Begriff »Objekt« etwas großzügiger umgehen werden.

Wenn wir diesen Quellcode, bestehend aus der Anweisung

```
cout << "Willkommen bei C++";
```

nun kompilieren würden, erhielten wir eine Reihe von Fehlermeldungen, obwohl die Ausgabeanweisung selbst syntaktisch korrekt ist.

Da Anweisungen aber, wie gesagt, die kleinste Einheit eines C++-Programms darstellen, müssen sie in eine größere eingebunden werden. Sie dürfen nicht so wie jetzt für sich alleine im freien Raum stehen.

## 3.2.2 Funktionen

Ein Container für Anweisungen ist die Funktion. Es gilt:

- Jedes C++-Programm setzt sich aus einer oder mehreren Funktionen zusammen.
- Jede Anweisung muss innerhalb einer Funktion stehen.
- Eine Funktion kann eine beliebige Anzahl von Anweisungen enthalten.



**Hinweis**

Zu den Funktionen und Anweisungen kommen im Zuge der objektorientierten Programmierung noch die Klassen mit ihren Methoden hinzu. Auch von der Regel, dass Anweisungen innerhalb von Funktionen stehen müssen, gibt es Ausnahmen, die uns aber hier noch nicht interessieren sollen.

Der Aufbau einer Funktion gliedert sich grob in einen Funktionskopf und einen Funktionsrumpf. Bis auf weiteres werden wir uns mit nur einer Funktion befassen. Sie ist auch die wichtigste, nämlich die Funktion `main()`. Das Grundgerüst von `main()` wird folgendermaßen definiert:

```
int main(void)
{
}
```

Die Zeile `int main(void)` bildet den Funktionskopf von `main()`, wobei das `void` in den Klammern auch entfallen kann. Also

```
int main(void)
```

oder eben

```
int main()
```

**Hinweis**

Wenn Sie in Ihrem Quelltext einen Funktionsnamen schreiben, dürfen Sie die nachfolgenden Klammern nicht vergessen. Sie gehören zur Funktion. Aber auch wenn man in einer Dokumentation auf eine Funktion verweist, ist es üblich, hinter den Funktionsnamen zwei runde Klammern zu setzen, man schreibt also nicht `main`, sondern `main()`. Dann weiß der Leser sofort, dass es sich um eine Funktion handelt und nicht etwa um eine Variable oder eine Klasse (zu Variablen später mehr).

Der Beginn des Funktionsrumpfes wird durch eine öffnende geschweifte Klammer (`{`) und das Ende des Funktionsrumpfes durch eine schließende geschweifte Klammer (`}`) festgelegt. Alle Anweisungen müssen zwischen diesen beiden Klammern stehen, also

```
int main(void)
{
    // Anweisung(en)
}
```

Bei dem doppelten Schrägstrich (`//`) handelt es sich um ein Kommentarzeichen. Es bewirkt, dass alle folgenden Zeichen bis zum Zeilenende als Kommentar angesehen werden. Kommentare werden beim Übersetzungsvorgang ignoriert, haben also keine Aus-

wirkung auf den späteren Programmablauf – dazu mehr in Kapitel 6 »Kommentare«. Hier weist der Kommentar darauf hin, an welche Stelle Anweisungen zu positionieren sind. Wir müssen also die Kommentarzeile gleich durch die tatsächliche Anweisung ersetzen.

#### **Tipp**

Halten Sie sich an die oben genannten Programmierstilregeln. Schreiben Sie die geschweiften Klammern stets in eine extra Zeile und rücken Sie die Anweisung(en) innerhalb der beiden Klammern ein.

Die Funktion `main()` ist deshalb so wichtig, weil sie den Einstiegspunkt ins Programm darstellt. In der Regel besteht ein C++-Programm nicht nur aus einer (was auf unsere ersten Programme zutrifft), sondern aus mehreren Funktionen. Wenn Sie den Kompilervorgang starten, arbeitet der Compiler nicht etwa die Funktionen nacheinander ab, so wie sie im Quelltext stehen (wie das bei den Anweisungen der Fall ist), sondern er sucht gezielt nach der Funktion `main()`. Findet er sie nicht, bricht er die Übersetzung ab, und Sie erhalten eine entsprechende Fehlermeldung.

Aus dem Gesagten ergibt sich:

- In jedem C++-Programm muss genau eine Funktion `main()` definiert sein. Damit wird die Stelle festgelegt, an der die Programmausführung beginnt.
- Die erste Aktion, die beim späteren Programmablauf stattfindet, wird von der ersten Anweisung im Rumpf der Methode `main()` bestimmt.

Zur Erklärung: Die Ausführung eines C++-Programms beginnt immer mit der ersten Anweisung in der `main()`-Funktion. Wenn sich in einem Quellcode keine Funktion `main()` befindet oder ein Quellcode gleich mehrere `main()`-Funktionen enthält, ist nicht festgelegt, wo die Programmausführung beginnen soll. Innerhalb der `main()`-Funktion, wie auch allen anderen Funktionen, werden die Anweisungen sequenziell abgearbeitet (siehe oben).

In welcher Reihenfolge Funktionen ausgeführt werden, erfahren Sie in Kapitel 20 »Funktionen«. Bis dahin werden wir uns lediglich auf eine Funktion, `main()`, beschränken. Das bedeutet:

- Alle Anweisungen befinden sich im Rumpf der Funktion `main()`.
- Die im Quelltext am weitesten oben stehende Anweisung (in `main()`) wird als Erstes ausgeführt,
- die am weitesten unten stehende (ebenfalls in `main()`) zuletzt.

Zum letzten Punkt sei angemerkt, dass die »am weitesten unten stehende Anweisung« bis auf Weiteres stets `return 0;` lauten wird. Damit hat es Folgendes auf sich: Der ANSI/ISO-C++-Standard schreibt vor, dass die Funktion `main()` immer einen Wert zurückgeben muss. Dies erreichen Sie mit der `return`-Anweisung. Der Wert 0 zeigt gewöhnlich an, dass ein Programmablauf erfolgreich beendet wurde. Mehr über Funktionen und deren Rückgabewerte erfahren Sie in Kapitel 20. Bis dahin gewöhnen Sie sich einfach an, als letzte Anweisung in der `main()`-Funktion jeweils `return 0;` zu verwenden.

**Hinweis**

Ausnahmen können sich nur dann ergeben, wenn Sie in Ihren Programmen Verzweigungen einrichten. Dann müssen Sie gegebenenfalls die `return`-Anweisung für jeden einzelnen Programmzweig vorsehen, der das Programm beendet (zu Programmverzweigungen siehe Kapitel 16 »Verzweigungen«).

Wenn wir nun die Ausgabeanweisung `cout << "Willkommen bei C++";` – zusammen mit der `return`-Anweisung – in die Funktion `main()` integrieren, ergibt sich für unseren Quelltext:

```
int main(void)
{
    cout << "Willkommen bei C++";
    return 0;
}
```

**Hinweis**

Über die Zeile `int main (void)` – bzw. `int main()` – brauchen Sie sich noch keine Gedanken zu machen. Ihnen soll nur klar sein, dass es sich dabei um den Funktionskopf von `main()` handelt.

Die Bedeutung von `int` vor und `void` einschließlich der runden Klammern nach dem Funktionsnamen `main` werden Sie später noch kennen lernen (siehe die Abschnitte 20.3 »Funktionsparameter« und 20.4 »Rückgabewerte von Funktionen«). Solange wir nur in `main()` programmieren, lautet der Funktionskopf stets gleich.

Ebenso wie den Funktionskopf von `main()` sollten Sie auch die Zeilen

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

zunächst kritiklos zu übernehmen (mehr darüber erfahren Sie ebenfalls in Kapitel 20 »Funktionen«). Beide Programmzeilen werden, abgesehen von weiteren Präprozessor-Direktiven, in allen Quelltexten stets gleich lauten.

**3.2.3 #include <iostream>**

Fürs Erste dazu nur so viel: Anweisungen, die mit einer Raute beginnen (`#`), richten sich an den Präprozessor, `iostream` (bzw. `iostream.h` bei älteren Compilern) ist eine so genannte Headerdatei. Mit der Anweisung `#include <iostream>` weisen Sie den Präprozessor an, den Inhalt der Headerdatei an dieser Stelle in den Quellcode einzufügen. Dies geschieht, bevor ihn der eigentliche Compiler zur Übersetzung in Maschinencode erhält (siehe Abschnitt 1.2.3 »Präprozessor und Linker«).

Um die Frage nach dem Sinn zu klären, bedarf es einer weiteren Erläuterung: Ihr Compilersystem verfügt über eine Vielzahl von vordefinierten Klassen, Funktionen, Objekten usw. Dies ist für Sie als Programmierer eine große Hilfe, da Sie diese Elemente, ohne sich große Gedanken machen zu müssen, einfach verwenden können, wenn Sie sie benötigen. So brauchen Sie nicht selbst eine Funktion zu definieren, die Daten auf den Bildschirm ausgibt (wie Sie Funktionen selbst definieren, erfahren Sie in Kapitel 20 »Funktionen«). Sie verwenden dazu einfach das vordefinierte Objekt `cout`.

Allerdings müssen Sie dem Compiler vorher mitteilen, dass Sie bestimmte Funktionen verwenden wollen. Genau dies geschieht mit der Einbindung von Headerdateien via `include` (zu Deutsch »einbeziehen, einschließen«), in diesem Fall der Headerdatei `iostream`. Das »i« in `iostream` steht für »input« (Eingabe), das »o« für »output« (Ausgabe). Das bedeutet, wenn Sie die Headerdatei `iostream` mit der `#include`-Anweisung einbinden, stehen Ihnen im Weiteren eine Reihe von vordefinierten Elementen für Ein- und Ausgabeoperationen zur Verfügung. Wir werden im weiteren Verlauf dieses Buches davon noch ausgiebig Gebrauch machen.

### 3.2.4 using namespace std;

Die vordefinierten Funktionen, Klassen etc., die nach Einbindung der Headerdatei `iostream` zur Verfügung stehen, sind Teil des Namensraums `std` (`std` steht für »Standard«). Um sie bequem nutzen zu können, müssen Sie den Namensraum `std` mit `using namespace std;` aktivieren.

#### Hinweis

Dies gilt jedoch nur für Compiler, die sich nach dem ANSI/ISO-C++-Standard richten. Bei älteren Compilern, die den ANSI/ISO-Standard nicht unterstützen, dürfen Sie `using namespace std;` nicht verwenden. Es sei darauf hingewiesen, dass den Ausführungen in diesem Buch der ANSI/ISO-Standard der Programmiersprache C++ zugrunde liegt. Aber nicht nur aus diesem Grunde möchten wir Ihnen empfehlen, gegebenenfalls auf ein neueres Compilermodell umzusteigen.

Würden Sie die Zeile `using namespace std;` weglassen, müssten Sie Ihrem Compiler (vorausgesetzt, dieser unterstützt den ANSI/ISO-C++-Standard) explizit mitteilen, dass Sie das Objekt `cout` in dem Namensraum `std` – und nicht etwa ein gleichnamiges anderes – meinen:

```
#include <iostream>
int main(void)
{
    std::cout << "Willkommen bei C++";
    return 0;
}
```

Um zu vermeiden, jedem Aufruf einer in `iostream` deklarierten Funktion den Text `std::` voranstellen zu müssen, teilen Sie Ihrem Compiler gleich zu Anfang mit, dass sich Namen, die im weiteren Code verwendet werden, auf Objekte des Namensraums `std` beziehen (`using namespace std;`).

Nach Einbeziehung der `#include`- und der `using namespace`-Zeilen ergibt sich nun für unseren Quellcode:

```
#include <iostream>
using namespace std;
int main(void)
{
    cout << "Willkommen bei C++";
    return 0;
}
```

Zu guter Letzt fügen wir nach der Ausgabe von

```
Willkommen bei C++
```

noch einen Zeilenvorschub ein. Dies geschieht mit der Anweisung

```
cout << endl;
```

Als vollständigen Quelltext für unser erstes Programm erhalten wir somit

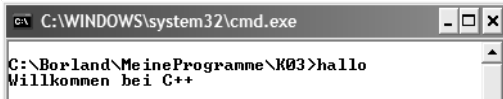
```
#include <iostream>
using namespace std;

int main(void)
{
    cout << "Willkommen bei C++";
    cout << endl;
    return 0;
}
```

### Hinweis

Wir haben im obigen Listing zwischen `using namespace std;` und `int main(void)` eine Leerzeile eingefügt, um die Präprozessor-Direktive und die Einbindung des Namensraums `std` vom übrigen Code abzuheben. Dies hat, wie Sie bereits wissen, keine Auswirkungen auf das Verhalten des Programms und ist letzten Endes eine reine Geschmacksfrage.

Das erwartete Ergebnis können Sie der Abbildung 3.2 entnehmen.



**Abbildung 3.2:** Ausgabe mit Zeilenvorschub

Was unser Programm leistet, ist nicht viel, aber fürs Erste wollen wir damit zufrieden sein:

- Ausgabe der Zeichenfolge Willkommen bei C++.
- Setzen des Cursors in die nächste Zeile.

Es sei erneut darauf hingewiesen, dass sich das Ergebnis im Grunde genommen auf die beiden Anweisungen

```
cout << "Willkommen bei C++";
```

und

```
cout << endl;
```

zurückführen lässt.

Woran erkennt ein C++-Compiler nun das Ende einer Anweisung? Die Frage drängt sich auf, da für ihn Zwischenraumzeichen weitgehend bedeutungslos sind. Möglicherweise haben Sie sich darüber bereits Gedanken gemacht und Sie haben diesbezüglich bei obigen Anweisungen eine Gemeinsamkeit entdeckt – das Semikolon am Ende.

Merken Sie sich schon jetzt:

Jede Anweisung in C++ muss mit einem Semikolon enden.

# 4

## Kompilieren und Ausführen von C++-Programmen

Sie werden nun die notwendigen Schritte zur Kompilierung und Ausführung Ihres ersten Programms (und damit auch aller weiteren Programme) kennen lernen. Zur besseren Übersicht hier noch einmal der Quellcode:

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

int main(void)
{
    cout << "Willkommen bei C++";
    cout << endl;
    return 0;
}
```

Wir werden diesen Quellcode mit Borland C++ 5.5 übersetzen lassen. Im Anschluss daran werden wir das Endprodukt, die ausführbare Datei, sowohl auf der Konsole als auch aus dem Windows-Explorer heraus zur Ausführung bringen.

### 4.1 Was für einen Editor sollten Sie verwenden?

Als Erstes müssen Sie den Quellcode in einen Editor eingeben. Fortgeschrittene Leser, die mit einer IDE wie Visual Studio arbeiten, werden natürlich auf den integrierten Editor dieser Entwicklungsumgebung zurückgreifen. Als Programmierneuling wird sich Ihnen jetzt aber vermutlich die Frage stellen, welcher Editor für die Eingabe des Quellcodes zu verwenden ist.

Grundsätzlich steht es Ihnen frei, welchen Editor Sie verwenden. Sie müssen nur darauf achten, den Quellcode als reinen Text, im so genannten ASCII-Format, zu speichern. Andernfalls werden vom Textverarbeitungsprogramm Steuerzeichen eingefügt, die vom Compiler bei der anschließenden Übersetzung nicht interpretiert werden können.

Sofern Sie mit einem Windows-Betriebssystem arbeiten, können Sie für die Eingabe Ihrer Quelltexte auch Notepad, den Windows-Editor benutzen. Fürs Erste ist dieser völlig ausreichend. Falls Notepad jetzt oder später Ihren möglicherweise steigenden Ansprüchen nicht mehr genügt, werden Sie im Internet eine Reihe von C++-Editoren mit Features wie farblicher Syntaxhervorhebung, automatischer Fehlererkennung, etc. finden.

### Tipp

Den Editor des Windows-Betriebssystems erreichen Sie mit der Befehlsfolge `START/PROGRAMME/ZUBEHÖR/EDITOR`.

Für Linux-Anwender bietet sich der Editor *vi* oder *emacs* an.

Unter Umständen stellt sich die Frage, unter welchem Namen die Quellcodedatei zu speichern ist, gleich beim Aufruf des Editors (so z.B. wenn Sie den *vi* verwenden). Dazu ist zu sagen: Den Namen (exklusive Dateierweiterung) können Sie – unter Beachtung der Vorgaben Ihres Betriebssystems – grundsätzlich frei wählen. Entscheidend ist allein die Dateierweiterung. Diese richtet sich nach dem Compiler, der den Quellcode später übersetzen soll.

### Hinweis

Zum besseren Verständnis noch einmal der Hinweis: Wir haben an anderer Stelle erklärt, dass mit dem Begriff Compiler unter Umständen die gesamte IDE (einschließlich der integrierten Software zum Übersetzen des Quellcodes) gemeint sein kann. Wenn der Compiler extern aufgerufen wird, wie das bei einem Kommandozeilen-Compiler der Fall ist, bezeichnet das Wort »Compiler« ausschließlich die Software zum Übersetzen des Quellcodes.

Schauen Sie daher in Ihrer Compiler-Dokumentation nach. In der Regel ist die Endung für C++-Quellcodedateien *.cpp* (für »C plus plus«). Dies trifft auf den Borland C++-Compiler 5.5 und ebenso auf den GCC-Compiler unter Linux zu. Falls Sie mit einer integrierten Entwicklungsumgebung arbeiten, brauchen Sie sich darum nicht zu kümmern, da diese automatisch dafür sorgt, dass die Quellcodedateien mit der richtigen Endung gespeichert werden.

Wir gehen in der folgenden Beschreibung davon aus, dass Sie unter Windows arbeiten und den Borland C++-Compiler sowie die in Kapitel 2, Abschnitt 2.3 »Installation des Borland-Compilers« dargestellte Verzeichnisstruktur verwenden. In diesem Zusammenhang sei nochmals die Empfehlung ausgesprochen, Ihre Quellcodedateien in einem separaten Verzeichnis zu speichern und später aus diesem Verzeichnis heraus den Compiler aufzurufen.

Tippen Sie also den Quellcode des »Willkommen bei C++«-Programms in Ihren Editor ein. Danach speichern Sie die Datei mit der Erweiterung *.cpp* (*hallo.cpp*).

### CD-ROM

Den Quellcode der Beispiele im Buch finden Sie auf der Buch-CD im Ordner *Beispiele*, geordnet nach Kapiteln. Die Datei *hallo.cpp* befindet sich dementsprechend im Unterverzeichnis *K04* (*Beispiele/K04*). Es steht Ihnen also frei, den Code aus den vorhandenen Quelldateien zu kopieren.



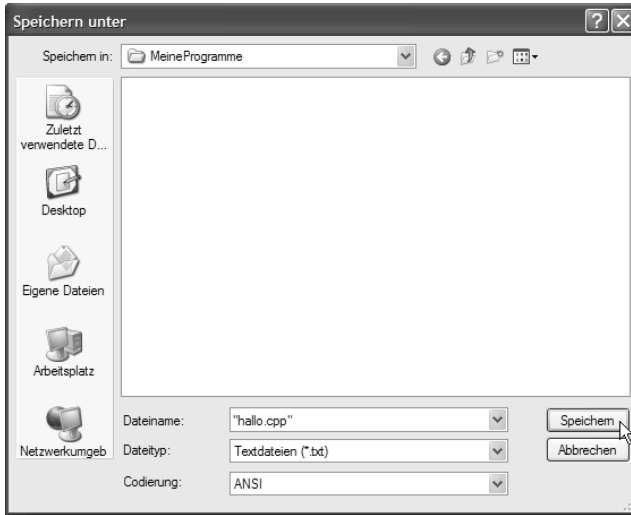


Abbildung 4.1: Speichern-Dialog des Windows-Editors

### Tipp

Wenn Sie sichergehen wollen, dass der Windows-Editor nicht von sich aus noch zusätzlich die Endung *.txt* anfügt, dann setzen Sie den Dateinamen zusammen mit der Erweiterung *.cpp* wie in Abbildung 4.1 zu sehen in doppelte Anführungszeichen.

Nach dem Speichern erscheint in der Titelleiste des Editors der Name der Quelldatei (Abbildung 4.2).

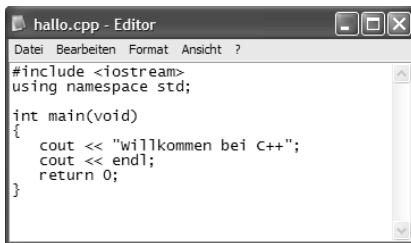


Abbildung 4.2: Nach dem Speichern

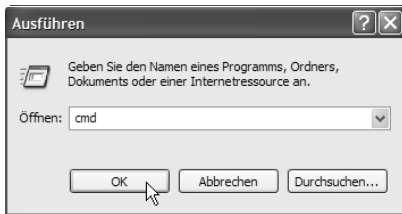
## 4.2 Kompilierung

Nun müssen wir die Datei *hallo.cpp* vom Borland-Compiler in Maschinensprache übersetzen lassen. Das Endprodukt nach erfolgreicher Kompilierung besteht in einer ausführbaren *.exe*-Datei.

Ein Kommandozeilen-Compiler wird als eigenständiges Computerprogramm von der Konsole aus gestartet. Dabei wird ihm die zu kompilierende Quellcodedatei mittels Parameterangabe bekannt gemacht, also ihm gewissermaßen beim Aufruf übergeben.

Für die weiteren Aktionen spielt es keine Rolle, ob die Quelldatei geschlossen oder noch geöffnet ist. Letzteres ist sogar von Vorteil, falls der Kompilierungsversuch scheitert und Sie sich in Ihrem Quellcode auf Fehlersuche begeben müssen.

Um den Kommandozeilen-Compiler unter Angabe eines Dateinamens aufrufen zu können, müssen wir uns auf die Konsole begeben. Sie ist für Windows-Betriebssysteme über das Startmenü erreichbar, entweder durch Auswahl von START/PROGRAMME(/ZUBEHÖR) und Klick auf EINGABEAUFFORDERUNG oder mit START/AUSFÜHREN (unter Windows 2000, Windows NT, Windows XP) bzw. START/PROGRAMME/ZUBEHÖR/AUSFÜHREN (Windows Vista) und Eingabe von `cmd` in der daraufhin erscheinenden Dialogbox (siehe Abbildung 4.3).



**Abbildung 4.3:** Aufruf der Konsole

Nach einer Bestätigung des Dialoges mit OK befinden wir uns in der Befehlszeile (Konsole), die je nach Windows-Version auf das Windows-Systemverzeichnis oder Ihr Benutzerverzeichnis eingestellt ist. Um im Verzeichnisbaum zum obersten Laufwerkverzeichnis `C:\` zu wechseln, schicken Sie den Befehl

```
cd c:\
```

ab.

Nun wechseln wir in das Verzeichnis *MeineProgramme*, in dem sich die Quellcodedatei *hallo.cpp* befindet. Um in ein Unterverzeichnis zu wechseln, geben Sie an der Eingabeaufforderung `cd` und – durch ein Leerzeichen getrennt – den Verzeichnisnamen ein. Um also zunächst ins Verzeichnis *Borland* und von da aus in das Verzeichnis *MeineProgramme* zu wechseln, müssen Sie nacheinander

```
cd Borland
```

und

```
cd MeineProgramme
```

eintippen (und den DOS-Befehl jeweils mit  abschicken).

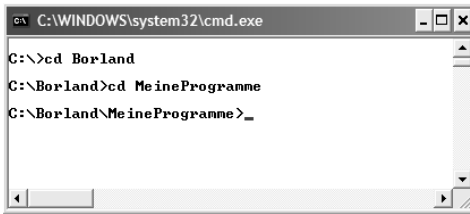


Abbildung 4.4: Wechsel ins Unterverzeichnis MeineProgramme

Wenn der Pfad zum Compiler (hier `C:\Borland\BCC5\Bin`) in der Path-Umgebungsvariablen enthalten ist (siehe Kapitel 2, Abschnitt 2.3 »Installation des Borland-Compilers«), dann lässt sich der Compiler ohne Pfadangabe aufrufen. Die entsprechende Datei heißt `bcc32.exe`.

In der Befehlszeile werden `.exe`-Dateien unter Angabe des Dateinamens (ohne Erweiterung) gestartet, also `bcc32.exe` mit dem Befehl `bcc32`. Allerdings ist noch der Name der zu kompilierenden Quelldatei (mit Erweiterung!) als Parameter zu übergeben. In unserem Fall lautet somit der notwendige DOS-Befehl zum korrekten Start des Compilers `bcc32 hallo.cpp`, um ihm die Datei `hallo.cpp` zum Übersetzen zu übergeben (Abbildung 4.5).

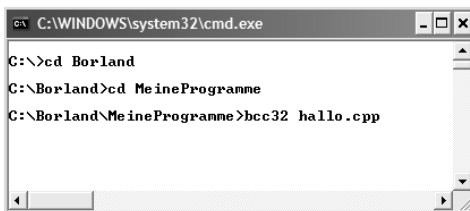



Abbildung 4.5: Aufruf des Borland C++-Compilers

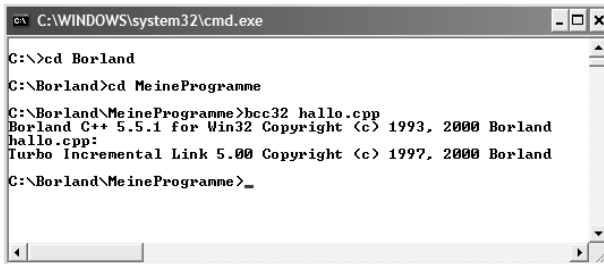
### Hinweis

Falls Sie unter Linux mit dem GCC- oder G++-Compiler arbeiten, lautet der Aufruf entsprechend `gcc hallo.cpp` bzw. `g++ hallo.cpp`.

### Tipp

Da wir uns im Verzeichnis *MeineProgramme* befinden – in dem die Quellcodedatei enthalten ist – reicht die alleinige Angabe des Dateinamens `hallo.cpp` aus. Falls die zu kompilierende Quelldatei jedoch nicht im aktuellen Verzeichnis liegt, ist die Angabe des vollständigen Pfades erforderlich. Auch unter diesem Gesichtspunkt ist es sinnvoll, ein eigenes Verzeichnis für alle Quellcodedateien einzurichten und den Compiler später aus diesem Verzeichnis heraus aufzurufen. Sie benötigen dann beim Compileraufruf weder Pfadangaben noch müssen Sie zwischen verschiedenen Verzeichnissen hin- und herwechseln.

Nach Eingabe des Startbefehls und dem Abschicken per  wird die Kompilierung in Gang gesetzt. Verläuft sie erfolgreich, wird im Anschluss daran eine ausführbare *.exe*-Datei im aktuellen Verzeichnis abgelegt. Ihr Name richtet sich nach der Quelldatei. In unserem Fall entsteht also die Datei *hallo.exe* (zusätzlich zur nach wie vor vorhandenen Datei *hallo.cpp*).



```

C:\WINDOWS\system32\cmd.exe
C:\>cd Borland
C:\Borland>cd MeineProgramme
C:\Borland\MeineProgramme>bcc32 hallo.cpp
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
hallo.cpp:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland
C:\Borland\MeineProgramme>_

```

Abbildung 4.6: Nach dem Kompilieren

Sollten Sie jedoch Fehlermeldungen erhalten, so müssen Sie Ihren Quelltext nochmals überarbeiten (heißt: die Fehler beseitigen – möglicherweise haben Sie ein Semikolon vergessen, »main« irrtümlicherweise großgeschrieben oder Ähnliches) und die Quelldatei erneut speichern.

## 4.3 Ausführen der .exe-Datei

Um das somit fertige Programm zur Ausführung zu bringen, starten Sie die *.exe*-Datei. Da diese in unserem Fall den Namen *hallo.exe* trägt, geschieht dies mit dem Befehl *hallo* (Eingabe des Dateinamens ohne Erweiterung).

### Tipp

Mit dem DOS-Befehl *CLS* bzw. *cls* (bei DOS-Befehlen brauchen Sie auf Groß- und Kleinschreibung keine Rücksicht zu nehmen) können Sie den Bildschirminhalt der DOS-Box löschen. *CLS* steht für Clear Screen.



```

C:\WINDOWS\system32\cmd.exe
C:\Borland\MeineProgramme>hallo

```

Abbildung 4.7: Starten der .exe-Datei

Nach Drücken der -Taste sollte sich folgendes Bild ergeben:



```

C:\WINDOWS\system32\cmd.exe
C:\Borland\MeineProgramme>hallo
Willkommen bei C++

```

Abbildung 4.8: Ausgabe des Programms hallo.exe

Unser Programm macht nun genau das, wozu wir es bestimmt haben: Es gibt die Zeichenfolge `Willkommen bei C++` auf den Bildschirm aus – und dann endet es auch schon wieder.

### Tipp

Hier noch ein Tipp für Leser, die in der Konsole arbeiten: Mit den Tasten `↑` und `↓` können Sie auf alle zuletzt verwendeten DOS-Befehle zugreifen, ohne diese erneut eintippen zu müssen.

## 4.3.1 Ausführung über den Windows-Explorer

Natürlich ist es auch möglich, *.exe*-Dateien über den Windows-Explorer – durch Doppelklick auf den Dateieintrag – zur Ausführung zu bringen.

Allerdings macht dies für das vorliegende Programm wenig Sinn, da Windows das Konsolenfenster nach Programmausführung sofort wieder schließt. Und da sich unser erstes Programm in der bloßen Ausgabe einer Zeichenfolge erschöpft, würde das Konsolenfenster in diesem Fall tatsächlich in Sekundenbruchteilen wieder verschwinden. Das heißt, Sie bekämen so gut wie nichts zu sehen.

Abhilfe schaffen Sie mit der Erweiterung des Quellcodes um einen so genannten Haltebefehl. Dann wartet das Programm erst auf einen Tastendruck, bevor es endet.

Für Interessierte hier ein entsprechend erweiterter Quellcode:

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

int main(void)
{
    cout << "Willkommen bei C++";
    cout << endl;
    getchar();
    return 0;
}
```

### CD-ROM

Die entsprechende Quellcodedatei finden Sie als *hallowe.cpp* auf der Buch-CD im Verzeichnis *Beispiele/K04* – das »we« in »hallowe« steht natürlich für Windows-Explorer.

Die Funktion `getchar()` wartet an sich auf die Eingabe eines Zeichens. Man kann Sie aber auch dazu nutzen, das Programm anzuhalten, bis der Benutzer die Eingabetaste drückt.

Wenn Sie den Quelltext in der obigen Form kompilieren, bekommen Sie bei der Ausführung auch dann etwas zu sehen, wenn Sie das Programm über den Windows-Explorer starten.



# 5

# Über Programm(ier)fehler

Es wird Ihnen vermutlich in den seltensten Fällen gelingen, den Quellcode von Anfang an fehlerfrei zu gestalten. Dies umso mehr, je umfangreicher Ihre Programme werden.

Tatsächlich nimmt die Fehlersuche in der Regel einen Großteil der Programmierstätigkeit in Anspruch. Auch wenn Sie sehr sorgfältig vorgehen, lässt sich die Fehlerhäufigkeit zwar minimieren, jedoch kaum gänzlich vermeiden. Es ist deshalb sinnvoll, sich schon sehr früh mit diesem Thema zu beschäftigen. Unter diesem Motto hier ein kleiner Exkurs zum Thema Fehler.

Am Ende des Kapitels weichen wir etwas vom eigentlichen Thema ab und geben – zum Abschluss des ersten Teils – einen kurzen historischen Abriss der Programmiersprache C++ (Abschnitt 5.3 »Historisches«).

## 5.1 Unterscheidung der verschiedenen Fehlerarten

Hinsichtlich ihrer Auswirkung unterscheidet man drei Gruppen von Fehlern:

- Syntaxfehler
- Laufzeitfehler
- logische Fehler

### 5.1.1 Syntaxfehler

Syntaxfehler sind solche, die vom Compiler erkannt und gemeldet werden. Quelltexte, die Syntaxfehler enthalten, können nicht erfolgreich kompiliert werden. Da somit kein ausführbares Programm aus syntaxfehlerbehaftetem Code entstehen kann, ist für diese Fehlergruppe allein die Bezeichnung Programmierfehler angebracht.

Bezüglich der Häufigkeit ihres Auftretens nimmt diese Fehlergruppe unbestreitbar den ersten Rang ein. Dennoch sind Syntaxfehler am einfachsten zu handhaben, weil man im Allgemeinen frühzeitig von ihnen Kenntnis erlangt. Um dies zu forcieren, empfiehlt es sich, den Quelltext zwischenzeitlich immer wieder zu kompilieren (um so von den Fehlermeldungen des Compilers zu profitieren).

Das setzt allerdings voraus, dass Sie im Verlauf der Programmierarbeit den Code in kompilierbaren Schritten weiterentwickeln. Sicherlich werden Sie sich nun fragen, was damit gemeint ist. Ziehen wir zur Verdeutlichung unser kleines Willkommen-Programm heran.

Als Sie Ihren Quellcode eingegeben haben, taten Sie dies vermutlich Zeile für Zeile, was – vom Programmieren einmal abgesehen – im Allgemeinen die übliche Vorgehensweise ist, um Text in einen Editor einzutippen. Nehmen wir an, Sie hätten gerade die erste Ausgabeanweisung eingegeben.

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

int main(void)
{
    cout << "Willkommen bei C++";
```

Wenn Sie obiges Codestück kompilieren würden, resultiert aus der fehlenden schließenden Klammer (}) in jedem Fall eine Fehlermeldung. Meist sogar mehrere, da die Interpretationsweise eines C++-Compilers von der menschlichen häufig abweicht (dazu gleich mehr). Von daher hätten die Meldungen Ihres Compilers also nur wenig Aussagekraft.

Richtig ist, die geöffnete Klammer gleich wieder zu schließen. Danach sollten Sie sogleich die return-Anweisung in die main()-Methode integrieren. Folgender Code ist für sich kompilierbar.

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

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

### Hinweis

Tatsächlich ist obiger Code auch ohne die return-Anweisung kompilierbar (ohne die schließende Klammer natürlich auf keinen Fall). Die Rückgabe eines Wertes erfolgt in diesem Fall automatisch. Strenggenommen ist Ihr Programm jedoch nur dann korrekt und entspricht dem ANSI/ISO-C++-Standard, wenn Sie explizit mit der return-Anweisung aus main() einen Wert zurückgeben.

Mit dem int vor dem Wort main in der Kopfzeile der Funktion kündigen Sie bereits an, dass main() einen Wert zurückliefert. Ansonsten würden Sie hier void verwenden. Bis zum Jahre 1999 hat man das auch so gehalten. Seitdem schreibt der ANSI/ISO-C++-Standard jedoch vor, dass main() einen Wert an das System zurückliefern muss. Wie gesagt, Sie brauchen sich darüber zum jetzigen Zeitpunkt noch keine Gedanken zu machen. Verwenden Sie bis auf Weiteres einfach int main(void) – oder int main() – im Kopf der main()-Funktion und schreiben Sie die return-Anweisung mit dem Wert 0 als letzte Anweisung in den Code. Mit Funktionen und deren Rückgabewerten werden Sie sich in Kapitel 20 noch intensiv auseinandersetzen.



Wenn Sie nun Fehlermeldungen erhalten, können Sie von einem Versehen Ihrerseits ausgehen (und den Fehler zunächst korrigieren, bevor Sie die Ausgabeanweisung zwischen die geschweiften Klammern setzen).

### Hinweis

Der Quelltext kann jetzt problemlos zu einem Programm kompiliert werden, da er die Mindestanforderung erfüllt: Er enthält eine Funktion `main()`. Dabei spielt es übrigens keine Rolle, wie viele Anweisungen im Rumpf einer Funktion codiert sind. Grundsätzlich ist eine Funktion auch dann kompilierbar, wenn Sie keine Anweisungen enthält. Allerdings würde die aus der Übersetzung hervorgehende *.exe*-Datei, wenn ausgeführt, mangels Anweisung keinerlei Aktionen durchführen. Dies trifft auch hier zu, da die einzige Anweisung, `return 0;`, praktisch nichts bewirkt. Die Weitergabe des Wertes 0 erfolgt – für den Anwender nicht sichtbar – systemintern.

Obiger Code bildet sozusagen das Grundgerüst für Ihre weiteren Programme, solange diese nur aus einer Funktion bestehen.

Nun könnten Sie die Anweisungen nacheinander (sofern es sich um mehrere handelt) in den Quellcode schreiben und zwischendurch immer wieder den Compiler aufrufen, um festzustellen, ob sich Fehler eingeschlichen haben.

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

int main(void)
{
    cout << "Willkommen bei C++";
    // ...

    return 0;
}
```

### Tipp

Wenn Sie an Ihrem Quellcode arbeiten, ist es ratsam, diesen zwischenzeitlich öfter zu kompilieren, um rechtzeitig auf Syntaxfehler aufmerksam gemacht zu werden. Dies verschafft Ihnen immer wieder das sichere Gefühl »bis hier ist alles o.k.«.

## 5.1.2 Programmfehler

Solange sich im Quelltext Syntaxfehler befinden, weigert sich der Compiler, daraus Maschinenbefehle zu erzeugen. Das hat den Vorteil, dass ein fehlerhaftes ausführbares Programm erst gar nicht entsteht. Wohlgermerkt gilt dies nur für Syntaxfehler. Leider erkennt der Compiler keine logischen Fehler und auch keine Laufzeitfehler. Beides wäre zu schön, aber andernfalls wäre der Compiler vermutlich in der Lage, gleich die ganze

Programmierarbeit zu übernehmen – und würde damit alle Programmierer überflüssig machen.

Laufzeitfehler sind solche, die die Syntaxprüfung des Compilers überstanden haben und daher erst bei Ausführung des Programms auftreten, meist in Abhängigkeit von anderen Faktoren, die z.B. durch das Benutzerverhalten oder das aktuelle Speicherbild des Computers bestimmt sein können. Ein Laufzeitfehler führt in der Regel zu einem so genannten Programmabsturz.

Ein typisches Beispiel für einen Laufzeitfehler ist eine versuchte Division durch die Zahl Null. Der Benutzer gibt eine Zahl ein, die vom Programm als Divisor verwendet wird. Ist der eingegebene Wert ungleich Null, treten keine Unstimmigkeiten auf. Wurde aber eine Null eingegeben, führt dies zum Programmabsturz. Folglich hätte der Programmierer es irgendwie verhindern müssen, dass an besagter Stelle überhaupt eine Null eingegeben bzw. vom Programm als Divisor eingesetzt werden kann, z.B. durch eine entsprechende Mitteilung und eine erneute Aufforderung zur Eingabe.

```
Eingabe: 0
Ungültiger Wert.
Bitte wiederholen Sie Ihre Eingabe: _
```

Zumindest müsste die Division vermieden werden, falls die Eingabe Null ist.

Am gefürchtetsten sind logische Fehler, weil sie in der Regel am schwersten zu finden sind. Das Programm scheint zwar fehlerfrei zu arbeiten, es macht aber eben nicht das, was es eigentlich soll. Ein Beispiel hierfür sind etwa fehlerhafte Berechnungsergebnisse.

Logische Fehler und Laufzeitfehler lassen sich als Programmfehler einstufen, da sie sich, wie gesagt, erst bei Programmausführung, also beim fertigen Programm, bemerkbar machen.

## 5.2 Fehlermeldungen »aus Compilersicht«

Wie oben angedeutet, reagiert der Compiler auf Syntaxfehler nicht immer in der Weise, wie wir das von unserem Denken her erwarten würden. Zeilenangaben, Fehleranzahl und angegebene Fehlerbeschreibung müssen daher nicht immer exakt zutreffen.

Betrachten wir folgendes Codestück:

```
...
cout << "Willkommen bei C++"
cout << endl;
...
```

Wie Sie bereits wissen, muss jede Anweisung in C++ mit einem Semikolon abgeschlossen werden. Daher ist für uns der Fall klar: Es fehlt ein Semikolon am Ende der ersten Codezeile (nach `cout << "Willkommen bei C++"`). Dort hätte es der Programmierer auch sicherlich hingesetzt, wenn er es nicht vergessen hätte.

Der Borland C++-Compiler vermutet den Fehler jedoch in der unteren Zeile. Denn dort (unmittelbar vor `cout << endl;`) bestünde sozusagen die letzte Möglichkeit, es zu platzieren.

```
...
cout << "Willkommen bei C++"
;cout << endl;
...
```

### Hinweis

Es sei daran erinnert, dass für einen C++-Compiler Zwischenraumzeichen (Zeilenumbrüche, Tabulatorschritte, Leerzeichen) weitgehend bedeutungslos sind (mehr dazu in Kapitel 7, Abschnitt 7.4 »Leerräume«). Von daher sind das obige Codestück und der entsprechende Auszug aus dem ursprünglichen (lauffähigen) Programm gleichwertig.

Hierzu noch eine Bemerkung. Tatsächlich erkennen moderne C++-Compiler (auch der von Borland) hier, dass es sich um ein fehlendes Semikolon handelt. Grundsätzlich ist für einen C++-Compiler eine Anweisung aber erst dann zu Ende, wenn er auf ein Semikolon trifft. Nach dieser Betrachtungsweise handelt es sich bei besagtem Codestück um genau *eine* Anweisung, deren Ende das Semikolon in der zweiten Zeile darstellt. So betrachtet sind

```
cout << "Willkommen bei C++"
cout << endl;
```

und

```
cout << "Willkommen bei C++" cout << endl;
```

gleichwertig (was hinsichtlich der genannten Compiler nicht zutrifft, da sie, wie gesagt, »schlauer« sind).

Fehlermeldungen könnten sich nun daraus ergeben, dass in dieser einen Anweisung (bestehend aus beiden Zeilen) Zeichen und Namen stehen, die im Kontext so nicht stehen dürfen. Beispielsweise ist die Verwendung des Namens `cout` pro Anweisung nur ein einziges Mal erlaubt (Kapitel 8, Abschnitt 8.3 »Ausgabeeinheiten aneinander hängen«).

Ein anderer nicht so »intelligenter« C++-Compiler könnte nun, beginnend mit der zweiten Zeile, konsequenterweise jeden einzelnen Textbaustein, der nicht in den gedachten Zusammenhang passt (eine Anweisung, bestehend aus zwei Zeilen), als Fehler melden. Ein fehlendes Semikolon wäre nach dieser Interpretationsweise jedoch nicht zu bemängeln.

Es sei hier angemerkt, dass das Verhalten eines Compilers in dieser Hinsicht von Fall zu Fall verschieden und auch für erfahrene Programmierer nicht immer vorhersehbar ist. Wie gesagt, hat jeder Compiler bei der Fehlerbewertung seine eigenen Maßstäbe. Und

letzten Endes handelt es sich bei dem Compiler auch nur um ein Softwareprogramm, das sich eben so verhält, wie es programmiert wurde.

Das Beispiel soll Sie auf die vereinzelt auftretende Notwendigkeit vorbereiten, eventuelle Fehlerangaben Ihres Compilers relativiert zu betrachten. Als Richtlinie für die Fehlersuche sollten Sie die Meldungen Ihres Compilers aber auf jeden Fall ansehen. Außerdem gilt ohne Einschränkung: Wenn Ihr Compiler sich weigert, den Quellcode zu übersetzen, sind in diesem mit Sicherheit Syntaxfehler vorhanden.

## 5.3 Historisches

Die Programmiersprache C++ wurde zu Beginn der Achtzigerjahre von Bjarne Stroustrup entwickelt. Zurückführen lässt sie sich auf die in den Siebzigerjahren entstandene Sprache C, die gegen Ende der Siebzigerjahre eine sehr große Akzeptanz erreicht hatte. Als Gründe für die damalige Popularität von C (die bis Mitte der Neunzigerjahre anhielt und auch heute noch zu einem großen Teil besteht, allenfalls abgelöst bzw. übertroffen von C++) sind wohl zu nennen:

- C erlaubt es, systemnah zu programmieren. Das bedeutet, man kann mit C sehr nahe an der Hardware (also am System) programmieren und damit Arbeiten durchführen, für die eigentlich die umständlichere und prozessorabhängige Maschinensprache vonnöten wäre. Gleichzeitig hat C aber auch Elemente von »höheren« Programmiersprachen (wie Funktionen). C vereint damit beide Welten.
- Ein in der Programmiersprache C verfasster Quellcode ist portierbar (zur Portierbarkeit von Programmen siehe Kapitel 1, Abschnitt 1.2.2 »Kompilierung«).

Beides zeichnet die Programmiersprache C gegenüber anderen Sprachen aus. Es spielten aber auch zwei Aspekte eine Rolle, die mit der Programmiersprache C an sich unmittelbar nichts zu tun hatten:

- das Aufkommen der Personalcomputer.
- das Buch »The C Programming Language« von Brian W. Kernighan und Dennis M. Ritchie, das 1978 erschien und seither als die »Bibel« von C gilt. Viele behaupten, dass gerade dieses Buch C erst richtig populär gemacht hat.

### 5.3.1 C/C++

Genau genommen ist C++ eine Erweiterung der Programmiersprache C um objektorientierte Merkmale (der ursprüngliche Name war »C mit Klassen«). Tatsächlich ist der Sprachumfang von C eine Untermenge von C++. Das heißt, ein C++-Compiler versteht jeden C-Befehl. (Die Umkehrung trifft natürlich nicht zu – C++ ist eine echte Obermenge von C.) Oder mit anderen Worten: Die Möglichkeiten von C sind voll von C++ übernommen worden. So gesehen ist jede C-Funktion gewissermaßen auch eine C++-Funktion.

Es ergibt durchaus Sinn, auf bereits in C vorhandene Möglichkeiten zurückzugreifen, es sei denn C++ (gemeint ist hier die Erweiterung gegenüber C, also derjenige Teil, der

nicht von C stammt) hat etwas Besseres zu bieten. So hätten wir unser erstes Programm unter Verwendung der C-Funktion `printf()` auch wie folgt implementieren können.

```
#include <stdio.h>
int main(void)
{
    printf("Willkommen bei C++");
    printf("\n");
    return 0;
}
```

Ausgabe: Willkommen bei C++

Um `printf()` verwenden zu können, muss die Headerdatei *stdio.h* eingebunden werden (`#include <stdio.h>`).

### Hinweis

Die Endung *.h* deutet darauf hin, dass eine Headerdatei aus der Programmiersprache C übernommen wurde bzw. aus älteren C++-Zeiten stammt (Kapitel 20, Abschnitt 20.2.2 »Prototypen von vordefinierten Funktionen«).

Die Funktion `printf()` bewirkt ebenso wie `cout` die Ausgabe eines Textes auf den Bildschirm. Die Anweisung `printf("\n");`, die hier anstelle von `cout << endl;` verwendet wird, erzeugt ebenfalls einen Zeilenvorschub. Wenn Sie obigen Quellcode kompilieren und ausführen, zeigt sich das gleiche Ergebnis wie bei unserem in C++-Manier verfassten Quelltext:

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

int main(void)
{
    cout << "Willkommen bei C++";
    cout << endl;
    return 0;
}
```

Ausgabe: Willkommen bei C++

Es sei darauf hingewiesen, dass wir im Zweifel natürlich auf die Mittel von C++ zurückgreifen, also Programme im C++-Stil schreiben werden. Wir verwenden für Ausgaben also nicht die C-Funktion `printf()`, sondern das Klassenobjekt `cout`, das in der Programmiersprache C noch nicht zur Verfügung stand.

Letzten Endes ist es ja so, dass es gute Gründe gab, zu bestimmten C-Funktionen C++-Entsprechungen anzubieten. Tatsächlich besitzt `cout` gegenüber `printf()` den Vorteil der Typsicherheit (was das heißt, werden Sie noch erfahren).

Es stellt sich die Frage, warum es `printf()` trotzdem noch gibt. Ein Grundgedanke bei der Konzeption von C++ war, dass jedes in C geschriebene Programm ebenfalls unter einer C++-Umgebung kompilierbar sein sollte. Man wollte ja bereits bestehenden Code nicht verwerfen, sondern mithilfe der zusätzlichen Möglichkeiten von C++ optimieren und erweitern.

### 5.3.2 ANSI/ISO-Standard

Wie nicht anders zu erwarten, unterliegt eine Programmiersprache, wenn sie eine gewisse Popularität erreicht hat, zahlreichen Veränderungen. Das rührt daher, dass verschiedene Hersteller ihre Compilermodelle mit zusätzlichen Funktionen ausstatten oder – was schlimmer ist – gar echte Veränderungen einführen. Somit besteht für die Sprache selbst die Tendenz, auseinander zu driften, was im äußersten Fall zu einem vollständigen Verlust der Portierbarkeit führen könnte.

Das war schon bei der Programmiersprache C in den Siebzigerjahren der Fall und genauso verhielt es sich auch mit C++ in den Neunzigerjahren. Während man sich bei C jedoch schnell auf einen gemeinsamen Standard einigen konnte, den so genannten ANSI C-Standard (benannt nach dem American National Standards Institute, dem Initiator der verabschiedenden Kommission), dauerte dies bei der Programmiersprache C++ fast ein ganzes Jahrzehnt. Die endgültige Verabschiedung eines gemeinsamen Standards erfolgte erst 1998. Da diese Entwicklung seit 1990 sowohl vom American National Standards Institute als auch von der International Organization for Standardization (ISO) gemeinsam vorangetrieben wurde, spricht man vom ANSI/ISO-Standard der Programmiersprache C++ (einige nennen ihn nur ANSI-Standard, wir werden im Folgenden des Öfteren einfach vom C++-Standard sprechen, gemeint ist aber dasselbe).

Letzten Endes ist der ANSI/ISO-Standard als ein gemeinsamer Nenner zu sehen, sozusagen als Richtschnur. Dass verschiedene Compilermodelle mehr oder weniger von diesem Standard abweichen, lässt sich leider auch in Zukunft nicht gänzlich vermeiden. Sie sollten aber möglichst einen Compiler verwenden, der diesen Standard weitgehend unterstützt, zumindest was den gemeinsamen Part betrifft. Zusätzliche Funktionen, wie sie z.B. ein Visual C++-Compiler zur Programmierung von grafischen Benutzeroberflächen anbietet, schaden nicht, solange sie nur eine Erweiterung und keine Veränderung darstellen.

Wir werden uns in den Beispielen dieses Buches grundsätzlich an den ANSI/ISO-Standard halten. Damit ist gewährleistet, dass alle dargestellten Listings auf Ihrem C++-Compiler erfolgreich übersetzt werden können, soweit dieser den ANSI/ISO-C++-Standard unterstützt. Zudem ist, wenn man von der Programmiersprache C++ redet, eben genau dieser ANSI/ISO-Standard von C++ gemeint und nicht eine spezielle Implementierung eines bestimmten Compilerherstellers.

# Teil II

## Das C++-ABC

In diesem Teil lernen Sie die elementaren Sprachelemente von C++ kennen, beispielsweise Variablen, Konstanten, Arrays, Schleifen und Verzweigungsanweisungen, um einige zu nennen. Es geht um das A und O der Programmierung – die hier beschriebenen Elemente sind nicht nur in C++ zu finden, sondern (wenngleich in veränderter Form) auch in anderen Programmiersprachen.

In Kapitel 20 »Funktionen« erfahren Sie, wie man Programme modularisiert, das heißt, den Quellcode auf mehrere Dateien verteilt.

Den Abschluss dieses Teils bildet die Programmierung eines Lottospiels unter Verwendung von zuvor selbst definierten Funktionen (Kapitel 21 »Eine Funktionsbibliothek« und Kapitel 22 »Ein Lottospiel«). Das bis dahin Gelernte soll auf diese Weise – gewissermaßen praxisnah – veranschaulicht werden.





# 6 Kommentare

Programmentwickler sehen sich häufig der Situation gegenüber, eigene oder gar fremde Quelltexte überarbeiten zu müssen. Dann ist es nicht immer leicht, sich im Quellcode zurechtzufinden, umso mehr, wenn es sich um sehr umfangreiche Programme handelt. Dies gilt insbesondere auch für selbst entwickelte Programme, wenn zwischen Erstellung und Überarbeitung ein größerer Zeitraum verstrichen ist.

Ein Mittel, Quellcode leichter nachvollziehbar zu machen, liegt darin, diesen an geeigneten Stellen zu kommentieren, das heißt ihn mit erläuternden Anmerkungen zu versehen. C++ stellt zu diesem Zweck verschiedene Kommentarzeichen zur Verfügung. Code Teile, die mittels dieser Symbole als Kommentar kenntlich gemacht sind, werden bei der Übersetzung des Quellcodes ignoriert.

Kommentare haben also keinen Einfluss auf das Laufzeitverhalten von C++-Programmen und bringen auch sonst keine Nachteile mit sich. Es handelt sich lediglich um Hinweise, die der Programmentwickler in den Quelltext einfügen kann. Adressaten dieser Informationen sind daher nicht Anwender, sondern ebenfalls Programmierer.

Man unterscheidet zwei Gruppen von Kommentarsymbolen:

- einzeilige Kommentare
- mehrzeilige Kommentare

## 6.1 Einzeilige Kommentare

Einzeilige Kommentare werden mit einem doppelten Schrägstrich (//) eingeleitet. Der Text, der dem // nachfolgt, wird bis zum Zeilenende als Kommentar angesehen.

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

int main(void)
{
    cout << "Willkommen bei C++"; // gibt die Zeichenfolge
    // "Willkommen bei C++" auf den Bildschirm aus.
    cout << endl;
    return 0;
}
```


Im obigen Beispiel haben wir den Quellcode unseres ersten Programms mit zwei einzeiligen Kommentaren versehen. Der Informationsgehalt beider Kommentare bezieht sich

sinngemäß auf die Anweisung `cout << "Willkommen bei C++";`, deren Wirkungsweise beschrieben wird. Beachten Sie, dass Sie den doppelten Schrägstrich in jeder Kommentarzeile erneut verwenden müssen.

Wenn Sie den Quelltext nun erneut speichern und kompilieren, dann würde sich am Ergebnis nichts ändern, da beide Kommentarzeilen nicht mitübersetzt werden. Was den Maschinencode der ausführbaren Datei *hallo.exe* angeht, hat sich also gegenüber der Ausführung in Kapitel 4 nichts geändert.

### Hinweis

Tatsächlich ist es der Präprozessor, der die Kommentare entfernt, bevor der Quellcode an den eigentlichen Compiler zur Übersetzung weitergereicht wird (zum Präprozessor siehe Kapitel 1, Abschnitt 1.2.3 »Präprozessor und Linker«).

An dieser Stelle möchte ich Sie auf eine Besonderheit aufmerksam machen. Wie Sie bereits wissen, ist C++ nicht zeilenorientiert. Obwohl es im Allgemeinen für die Lesbarkeit des Quelltextes sehr nützlich ist, jede Anweisung in eine eigene Zeile zu setzen, ist dies nicht zwingend notwendig. Das Ende einer Anweisung wird nicht am Zeilenumbruch erkannt, der durch  erzeugt wird, sondern am abschließenden Semikolon (wir werden darauf im nächsten Kapitel, dort im Abschnitt 7.2 »Anweisungsende«, noch einmal zu sprechen kommen).

Bei der Verwendung von einzeiligen Kommentaren tritt insofern ein Sonderfall auf, als in diesem Kontext das Zeilenende syntaktische Bedeutung erlangt: Es wird als das Ende des Kommentars interpretiert. Der Text, der in der Folgezeile steht, wird vom Compiler wieder als C++-Anweisung angesehen – es sei denn, es handelt sich wiederum um eine Kommentarzeile.

Machen Sie sich in diesem Zusammenhang auch klar, dass ein Kommentar im programmiertechnischen Sinne keine Anweisung ist. Zur Erinnerung: Eine Anweisung ist die Ursache dafür, dass – beim Programmablauf – etwas geschieht, eine Aktion ausgeführt wird. Kommentare dürfen zwar überall im Quelltext stehen – vorausgesetzt, sie sind als solche kenntlich gemacht –, haben jedoch auf das Laufzeitverhalten eines Programms keinerlei Auswirkung.

Der Kommentar in obigem Listing erfüllt dort allein den Zweck der Demonstration. Natürlich werden Sie später nicht daran denken, so unproblematische Codeteile mit Kommentaren zu versehen. Durchaus üblich sind aber eine einleitende Kommentarzeile ganz oben im Quellcode sowie die Kenntlichmachung des Endes von Funktionen, Kontrollstrukturen oder Klassendefinitionen (Letzteres ergibt natürlich nur dann Sinn, wenn der Quelltext entsprechend umfangreich ist).

```
// Dies ist mein erstes Programm
#include <iostream>
using namespace std;

// Dies ist mein erstes Programm
```

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

int main(void)
{
    cout << "Willkommen bei C++";
    cout << endl;
    return 0;
} // end main
```

## 6.2 Mehrzeilige Kommentare

Es ist auch möglich, zusammenhängende Codeabschnitte in Kommentare zu setzen. Die Kürzel `/*` und `*/` stehen hierbei für den Anfang und das Ende des Kommentarblocks. Der gesamte Text, der sich zwischen beiden Kürzeln befindet, wird dann als Kommentar angesehen.

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

/* blablabla ...
Dies ist ein Kommentar, der sich
über mehrere Zeilen erstreckt ...
blablabla ... Hier beginnt die Definition
der Funktion main()... blablabla*/
int main(void)
{
    cout << "Willkommen bei C++";
    cout << endl;
    return 0;
}
```

Selbstverständlich dürfen die Kürzel `/*` und `*/` auch verwendet werden, wenn sich die Kommentierung nur über eine Zeile erstrecken soll.

```
/* Dies ist mein erstes Programm */
#include <iostream>
using namespace std;

int main(void)
{
    cout << "Willkommen bei C++";
    cout << endl;
    return 0;
}
```

Da bei Verwendung von `/* ... */` das Kommentarende nicht am Zeilenende, sondern an dem Symbol `*/` erkannt wird, ist es sogar zulässig, Kommentare innerhalb einer Programmierzeile einzufügen:

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

void main(void)
{
    cout << /*Dies ist ein Kommentar*/ "Willkommen bei C++";
    cout << endl;
}
```

So ungewöhnlich diese Konstellation auch sein mag – obiges Listing würde für das ausführbare Programm kein anderes Resultat ergeben als alle zuvor genannten Varianten. Beim Programmlauf erscheint ebenfalls die Ausgabe `Willkommen bei C++` auf dem Bildschirm. Anschließend wird der Cursor in die nächste Zeile gesetzt (`cout<<endl;`).

Unerwünschte Ergebnisse würden Sie aber z.B. mit

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

int main(void)
{
    cout << "Willkommen//*Dies ist ein Kommentar*/ bei C++";
    cout << endl;
    return 0;
}
```

erzielen, da hier der »Kommentar« keiner mehr ist. Er wird als Teil der auszugebenden Zeichenfolge interpretiert, weil er zwischen deren Begrenzungszeichen steht ("`...`"). Die Folge ist, dass das Programm die Bildschirmausgabe

```
Willkommen//*Dies ist ein Kommentar*/ bei C++
```

erzeugt.

## Hinweis

Zeichenfolgen, die im Quellcode zwischen "`...`" stehen, werden »Zeichenketten« oder »Strings« genannt.

Zwischen den Kommentarsymbolen `/*` und `*/` darf jedes beliebige Zeichen stehen. Das gilt auch für die Zeichen `*` und `/` – mit der Einschränkung, dass diese Zeichen nicht in der Reihenfolge, die dem Kommentarende entspricht (`*/`), verwendet werden dürfen.

## Hinweis

Beachten Sie, dass C++ die Kommentarzeichen `/*` und `*/` wie auch Steuerzeichen und verschiedene Operatoren als *ein* Symbol behandelt – obwohl diese tatsächlich aus zwei Zeichen bestehen (zu Steuerzeichen und Operatoren siehe Kapitel 9 »Steuerzeichen« bzw. Kapitel 11 »Ausdrücke und Operatoren«).

Aus dem oben genannten Grund ist es auch nicht erlaubt, Kommentarblöcke zu verschachteln:

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

int main(void)
{
    /* blablabla
       cout << "Willkommen bei C++";
    /* blablabla
    blablabla */
       cout << endl;
       return 0;
    blablabla */
}
```

Nehmen wir obiges – fehlerhaftes – Listing zum Anlass, Überlegungen »aus der Sicht eines gedachten Übersetzers« anzustellen: Das erste Kommentarsymbol (`/*` in Zeile 6) interpretiert der Compiler als Kommentaranfang – was vom Programmierer sicherlich auch so gewollt war.

Die Zeichenfolge `/* blablabla` in Zeile 8 wird ebenfalls als zum Kommentarblock gehörig angesehen. (Wie oben dargelegt, dürfen die Zeichen `/` und `*` in dieser Reihenfolge im Kommentar verwendet werden.)

Die nächste Folge der Zeichen `*` und `/` (Zeile 9) darf in dieser Kombination (`*/`) jedoch nicht als Kommentartext zwischen zwei korrespondierenden Kommentarsymbolen stehen. Sie wird immer als das Ende eines mit `/*` eingeleiteten Kommentars betrachtet. Somit präsentiert sich dem Compiler folgender Kommentarblock (Zeile 6 bis Zeile 9):

```
/* blablabla
   cout << "Willkommen bei C++";
/* blablabla
blablabla */
```

Folglich wird Ihr Compiler versuchen, alle nachfolgenden Textbausteine als reguläre C++-Anweisungen zu interpretieren. Denkt man sich nun den Teil weg, der im Sinne dieser Auslegung als Kommentar zu verstehen ist, dann präsentiert sich der in Maschinensprache zu übersetzende Quellcode wie folgt:

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

int main(void)
{
    cout << endl;
    return 0;
    blablabla */
}
```

In diesem ist somit ein für C++ unbekannter Name enthalten (blablabla) sowie ein Kommentarendesymbol (\*//), das ohne Vorhandensein eines einleitenden Pendants nicht interpretierbar ist. Folglich wird Ihr Compiler auf einen Kompilierversuch mit mehreren Fehlermeldungen reagieren (mindestens zwei).

Dabei spielt es selbstverständlich keine Rolle, wie der Programmierer sich das beim Erstellen des Quellcodes vorgestellt hat. Es sei daher an dieser Stelle noch einmal ausdrücklich darauf hingewiesen, wie wichtig es ist, Quelltexte so aufzubauen, dass sie eindeutig interpretiert werden können.

### **Tipp**

Versetzen Sie sich bei der Kontrolle Ihres Quellcodes in die Lage eines gedachten Übersetzers, um zu überprüfen, ob die einzelnen Anweisungen auch eindeutig interpretierbar sind.

Da Ihnen Kommentare nun bekannt sind, werden wir sie ab jetzt des Öfteren verwenden, um Programmierzeilen erläuternde Anmerkungen hinzuzufügen. Dies geschieht dann im Zusammenhang mit neuen Lerninhalten vor allem aus didaktischen Gründen, also unter Berücksichtigung der besonderen Voraussetzungen des Lesers als Programmier- bzw. C++-Neuling.

Im Übrigen ist eine eher sparsame Verwendung von Kommentaren angezeigt. Kommentare sollen zur besseren Lesbarkeit des Quellcodes beitragen. Setzen Sie sie in Ihren Programmen also nur dann ein, wenn es wirklich angebracht ist. Andernfalls würden Sie den gegenteiligen Effekt erzielen. Sie werden im weiteren Verlauf noch ausreichend Gelegenheit haben, den sinnvollen Einsatz von Kommentaren zu üben.

## **6.3 Einsatz von Kommentaren bei der Fehlersuche**

Ein weiterer Nutzen von Kommentaren bezieht sich auf die Fehlersuche. Um bei längeren Quelltexten Fehler einzugrenzen, ist es häufig sinnvoll, manche Codeteile vorübergehend als Kommentare zu kennzeichnen – man spricht dabei auch von »auskommentieren« –, damit sie bei der Übersetzung vorübergehend ignoriert werden.

Hierzu folgendes Beispiel:

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

int main(void)
{
    Cout << "Willkommen bei C++";
    Cout << endl
    return 0;
}
```

Beim Versuch, obiges Listing zu kompilieren, meldet der Borland-Compiler zwei, andere Compilermodelle sogar bis zu fünf Fehlern. Womöglich können Sie zum jetzigen Zeitpunkt mit den Fehlermeldungen noch nicht viel anfangen. Sie könnten aber zuverlässig feststellen, ob die Fehlerursache in einer der Ausgabeanweisungen liegt, indem Sie diese vorübergehend als Kommentare kennzeichnen. Das kann entweder mit den Symbolen `/*` und `*/`

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

int main(void)
{
    /*Cout << "Willkommen bei C++";
    Cout << endl*/
    return 0;
}
```

oder mit zwei einzeiligen Kommentaren geschehen (`//`):

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

int main(void)
{
    //Cout << "Willkommen bei C++";
    //Cout << endl
    return 0;
}
```

Da obiger Quelltext nunmehr kompilierbar ist, lässt das den Schluss zu, dass die (der) Fehler in den Ausgabeanweisungen zu suchen sind (ist). Wie gesagt, ist es nicht auszuschließen, dass es sich um nur eine einzige Fehlerursache handelt. Wie Sie ja bereits wissen, ist es durchaus denkbar, dass Ihr Compiler bei Weglassen auch nur eines Zeichens mit mehreren Fehlermeldungen reagiert.

Um die Fehlerursache(n) einzugrenzen, erscheint es sinnvoll, zunächst bei einer Anweisung das Kommentarsymbol zu entfernen:

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

int main(void)
{
    Cout << "Willkommen bei C++";
    //Cout << endl
    return 0;
}
```

Nun wird die Ausgabeanweisung

```
Cout << "Willkommen bei C++";
```

mitkompiliert.

Da Sie nun Fehlermeldung(en) erhalten, empfiehlt es sich, diese Anweisung etwas genauer unter die Lupe zu nehmen. Sicher werden Sie bald erkennen, dass der Name `cout` fälschlicherweise großgeschrieben ist. Es muss heißen

```
cout << "Willkommen bei C++";
```

und nicht

```
Cout << "Willkommen bei C++";
```

Nach der Änderung verläuft die Kompilierung erfolgreich. Damit ist bis auf die Anweisung

```
Cout << endl
```

alles in Ordnung.

Nachdem Sie gerade darauf aufmerksam geworden sind, erkennen Sie natürlich sofort zumindest eine Fehlerursache in dem großgeschriebenen `Cout`. Also schreiben Sie es klein und entfernen das Kommentarsymbol:



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

int main(void)
{
    cout << "Willkommen bei C++";
    cout << endl
    return 0;
}
```

Wie ein erneuter Kompilierversuch zeigt, ist dieser Quelltext immer noch nicht fehlerfrei. Allerdings wird es Ihnen jetzt nicht mehr schwer fallen, die Fehlerursache in dem fehlenden Semikolon am Ende der Zeile

```
cout << endl
```

zu erkennen. Wenn Sie dieses nun eingefügt haben, verläuft die Kompilierung erfolgreich.

### Hinweis

Mitunter und gerade für den Programmierneuling kann der Ausschluss von nur einer oder zwei Programmierzeilen bei der Fehlersuche zwar schon von Nutzen sein. In der Regel bedient man sich dieser Vorgehensweise – Eingrenzung von Fehlern durch Auskommentierung von Codestücken – aber vor allem bei umfangreicheren Quelltexten. Denken Sie daran, dass diese Methode nicht nur in Bezug auf Syntaxfehler, sondern auch bei der Beseitigung von Laufzeitfehlern oder logischen Fehlern hilfreich sein kann.

Hier sei noch erwähnt, dass moderne Entwicklungsumgebungen wie die Visual-C++-IDE für die Suche nach logischen und Laufzeitfehlern einen integrierten Debugger bereitstellen. Mit diesem lässt sich das fertige Programm (die *.exe*-Datei) schrittweise ausführen. Dieses Werkzeug kann jedoch zum jetzigen Zeitpunkt für den Programmierneuling noch nicht von Nutzen sein. Bemühen Sie zur Bedienung des Debuggers die Dokumentation Ihres Compilers, wenn es so weit ist. Für die Lektüre dieses Buches kommen Sie aber auch gut ohne Debugger aus.

Achten Sie bei der Ausblendung von Codestücken darauf, dass der übrige Teil erfolgreich kompiliert werden kann.

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

int main(void)
/*{
    cout << "Willkommen bei C++";
    cout << endl;*/
    return 0;
}
```

Beispielsweise ist obiger Quelltext in keinem Fall kompilierbar, da die öffnende Klammer des Funktionsrumpfes von `main()` versehentlich mit auskommentiert wurde. Für jede schließende Klammer muss im zu kompilierenden Code aber auch eine öffnende vorhanden sein und umgekehrt. Bedenken Sie zudem, dass Sie das Grundgerüst von `main()` auf keinen Fall ausblenden dürfen, da dann im Quellcode de facto keine Methode `main()` vorhanden ist.

# 7

# Syntaxregeln

Wie in jeder anderen Programmiersprache sind auch in C++ beim Editieren des Quellcodes bestimmte sprachspezifische Regeln zu beachten. Einige davon wurden bereits angesprochen. Das Wesentliche soll hier noch einmal in kompakter Form und abschließend dargestellt werden.

## 7.1 Textbausteine

Wie Sie bereits wissen, ist die kleinste zusammengehörige Einheit in C++ die Anweisung. Allerdings setzt sich eine Anweisung in der Regel aus einer Reihe von Textbausteinen zusammen – solange es sich nicht um eine leere Anweisung handelt (was das ist, erfahren Sie gleich weiter unten).

Textbausteine können Schlüsselwörter, Bezeichner, konstante Werte, Operatoren oder Zeichen mit besonderer Bedeutung sein.

Schlüsselwörter sind Namen, die der Compiler kennt und die eine ganz bestimmte Bedeutung haben. Beispiele für Schlüsselwörter sind etwa `int`, `void` oder `using`. Beide haben Sie in Ihrem ersten Programm bereits verwendet. Was es mit Operatoren, Bezeichnern und Konstanten auf sich hat, werden Sie in Kürze erfahren. Beispiele für Zeichen mit besonderer Bedeutung sind etwa in Verbindung mit Zeichenketten die doppelten Anführungszeichen ("`...`"), die den Beginn und das Ende einer Zeichenkette festlegen. Auch das Semikolon als Endezeichen einer Anweisung fällt darunter.

### Hinweis

Textbausteine werden auch »Token« genannt.

Die einzelnen Textbausteine müssen innerhalb einer Anweisung eindeutig unterscheidbar sein, um vom Compiler in ihrer Bedeutung erkannt zu werden. Wo das nicht automatisch der Fall ist, müssen sie durch mindestens ein Leerzeichen voneinander getrennt werden. Nehmen wir als Beispiel die Anweisung

```
int name = 15;
```

Die genaue Bedeutung dieser Anweisung und der einzelnen Textbausteine werden Sie in Kapitel 10 »Variablen« erfahren. Hier nur so viel: `int` ist ein Schlüsselwort zur Angabe eines Datentyps, `name` ist ein Bezeichner für ein Datenobjekt – nämlich eine Variable –, das Gleichheitszeichen steht für einen Zuweisungsoperator und bei der Zahl 15 handelt es sich um einen konstanten Wert.

Wenn Sie nun schreiben

```
intname = 15; // FEHLER
```

kann das Schlüsselwort `int` natürlich nicht mehr als solches erkannt werden. Auch ist jetzt in der Anweisung kein Bezeichner `name` enthalten – allenfalls ein Bezeichner `intname`. Die Fehlermeldung, die diese Programmierzeile hervorruft, ergibt sich aus dem Fehlen eines einleitenden Schlüsselwortes zur Angabe eines Datentyps bzw. daraus, dass der Compiler mit einem Bezeichner `intname` in diesem Kontext vermutlich nichts anfangen kann (was das genau bedeutet, erfahren Sie ebenfalls in Kapitel 10).

In der obigen Anweisung ist es also zwingend erforderlich, zwischen dem Schlüsselwort `int` und dem Bezeichner `name` zumindest ein Leerzeichen zu setzen. Andererseits ist die Anweisung

```
int name=15;
```

– ohne Leerzeichen zwischen dem Bezeichner `name`, dem Zuweisungsoperator (`=`) und der Zahl `15` – syntaktisch in Ordnung. Der Grund dafür ist, dass der Compiler hier die einzelnen Token auch ohne Trennzeichen auseinander halten kann. Ein Bezeichner `name=` ist nicht zulässig, da Bezeichner, abgesehen von dem Unterstrich (`_`), keine Sonderzeichen enthalten dürfen (siehe dazu Abschnitt 10.2 »Regeln zur Bezeichnerwahl«). Folglich wird der Ausdruck `name=` vom Compiler auch nicht als Bezeichner interpretiert, sondern als Bezeichner `name` und Zuweisungsoperator `=`.

### Tipp

Um sich selbst vor Nachlässigkeiten zu schützen und auch die Lesbarkeit des Quellcodes zu verbessern, ist es im Allgemeinen jedoch sinnvoll, die einzelnen Textbausteine in jedem Fall durch ein Leerzeichen voneinander zu trennen.

Lassen Sie uns das eben Gesagte am Beispiel des Funktionskopfes von `main()` verdeutlichen:

```
int main(void)
```

Wenn Sie `int` und `main` zusammenschreiben

```
intmain(void)
```

sind das Schlüsselwort `int` und der Bezeichner `main` nicht mehr als solche interpretierbar. Demgegenüber ist es zulässig, zwischen dem Bezeichner `main` und der folgenden Klammer `(` auf Leerstellen zu verzichten. (Tatsächlich handelt es sich bei der Funktionsklammer um einen Operator, also um ein eigenes Token.) Im Allgemeinen werden die Leerstellen zwischen Funktionsname und `(` weggelassen – entgegen der oben genannten Empfehlung. Damit will man die Zusammengehörigkeit von Funktionsname und Parameterliste (dabei handelt es sich um den geklammerten Ausdruck) im Code verdeutlichen.

## 7.2 Anweisungsende

C++ ist nicht zeilenorientiert. Das heißt, das Ende einer Anweisung wird nicht am Zeilenende erkannt. Ausschlaggebend hierfür ist allein das abschließende Semikolon.

Dabei spielt es keine Rolle, welche Textbausteine vor dem Semikolon stehen. Das geht so weit, dass ein Semikolon ohne vorangehenden Code ebenfalls als C++-Anweisung angesehen wird. Diese hat sogar einen speziellen Namen. Man spricht in diesem Fall von einer leeren Anweisung. In dem Codefragment

```
cout << "Hallo";; // zwei Anweisungen
;;; // drei Anweisungen
```

sind demzufolge fünf Anweisungen enthalten. Natürlich haben die letzten vier Anweisungen keinerlei Auswirkung auf das Geschehen während des Programmlaufs.

Beachten Sie:

- Immer wenn im C++-Quellcode ein Semikolon auftritt, bildet dieses zusammen mit den vorangehenden Textbausteinen – beginnend mit dem letzten Semikolon – eine Anweisung.
- Codeteile, die nicht mit einem Semikolon abgeschlossen werden, sind in der Regel unvollständig und damit fehlerhaft.

## 7.3 Blöcke

Weitere Bausteine des Quellcodes neben den Anweisungen sind Blöcke. Mit ihnen lassen sich mehrere Anweisungen zusammenfassen. Meist geschieht dies im Zusammenhang mit Klassen, Methoden, Funktionen, Verzweigungs- und Schleifenkonstrukten. Man spricht deshalb vom Funktionsblock, Schleifenblock usw.

Grundsätzlich wird jeder Bereich zwischen einer öffnenden ({) und einer schließenden (}) geschweiften Klammer in C++ als Block angesehen.

Sie haben bereits einen Block verwendet. So handelt es sich bei einem Funktionsrumpf wie beispielsweise den von `main()` ebenfalls um einen Block. Die Bezeichnungen Funktionsrumpf und Funktionsblock sind daher gleichbedeutend.

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

Wie Ihnen bekannt ist, dürfen im Rumpf (Block) von `main()` eine oder mehrere Anweisungen stehen. Mittels dieser Konstruktion bilden diese Anweisungen gewissermaßen eine Einheit. Mit anderen Worten: Alle Anweisungen, die sich in diesem Block befinden, werden als zur Funktion `main()` gehörig interpretiert.

Ein C++-Programm kann beliebig viele Blöcke enthalten, die auch ineinander liegen dürfen. Als Beispiel für eine verschachtelte Blockstruktur sei hier die Definition einer Klasse `X` genannt.

```
class X
{
    ...
    void Y(void)
    {
        // Anweisung(en)
    }
    ...
};
```

Eine Klassendefinition wird mit dem Schlüsselwort `class` eingeleitet, dem sich ein frei wählbarer Name für die Klasse anschließt (hier `X`). Die äußeren `{...}` bilden den Klassenblock. Zu diesem Grundgerüst gehört noch das Semikolon nach der schließenden geschweiften Klammer. Dieses ist am Ende einer Klassendefinition erforderlich, was uns hier aber nicht interessieren soll. (Klassen sind Elemente der objektorientierten Programmierung, mit der wir uns in Teil III dieses Buches beschäftigen werden.)

Wie Sie aus dem Konstrukt vermutlich ersehen können, befindet sich innerhalb des Klassenblocks die Definition einer Funktion `Y()`. Sie gleicht im Wesentlichen der Ihnen bekannten Funktionsdefinition von `main()` – bis auf den Namen (`Y` statt `main`) und die Tatsache, dass sie keinen Rückgabewert hat. (Daher beginnt der Funktionskopf mit dem Schlüsselwort `void` und im Funktionsrumpf fehlt die `return`-Anweisung.)

### Funktionen und Methoden

Eigentlich ist die korrekte Bezeichnung für `Y()` hier »Methode«. In Aufbau und Wirkungsweise stimmen Methoden und Funktionen überein. Im Zusammenhang mit Klassen spricht man jedoch von Methoden.

Rein objektorientierte Programmiersprachen wie Java oder das neuere C# kennen keine Funktionen. Ihr Quellcode ist ausschließlich in Klassen (mit ihren Methoden) aufgeteilt.

Nicht objektorientierte Programmiersprachen wie etwa Pascal, Basic, Cobol und auch C kennen keine Klassen und damit auch keine Methoden. Sie besitzen als Bausteine ausschließlich Funktionen (oder damit verwandte Konstruktionen).<sup>1</sup>

Anders dagegen C++. In C++ gibt es sowohl Funktionen als auch Methoden. Steht eine Funktion für sich alleine, ohne in eine Klasse eingebunden zu sein, nennt man sie eben Funktion. Wird eine Funktion aber innerhalb einer Klasse definiert, dann heißt sie Methode. C++ besitzt also sowohl objektorientierte Features (Klassen, Methoden) als auch nicht objektorientierte Features (Funktionen).

<sup>1</sup> Der wachsenden Bedeutung objektorientierter Programmierung ist es zu verdanken, dass es einige der hier aufgeführten Sprachen mittlerweile um objektorientierte Konzepte erweitert wurden.

Wir nehmen an, innerhalb des Methodenblocks von  $\gamma()$  befinden sich mehrere Anweisungen ( $//$  Anweisung(en)).

Alle diese Anweisungen (die im Block der Methode  $\gamma()$  stehen), sind somit Teil dieser Methode. Sie gehören zu ihr. Ebenso wie die Methode  $\gamma()$  und eventuell weitere Methoden, die in demselben Block definiert sind, als zur Klasse  $\chi$  gehörig interpretiert werden.

### Achtung

Nicht erlaubt ist das Verschachteln von Funktionen bzw. Methoden untereinander. Unzulässig ist demzufolge:

```
// FEHLER
void eineFunktion(void)
{
    ...
    void nochEineFunktion(void)
    {
        ...
    }
    ...
}
```

Richtig muss es heißen:

```
// OK
void eineFunktion(void)
{
    ...
}
void nochEineFunktion(void)
{
    ...
}
```

Wir werden von der Möglichkeit, Anweisungen in Blöcken zusammenzufassen, im Zusammenhang mit Kontrollstrukturen noch intensiv Gebrauch machen.

Im obigen Beispiel ist das Setzen von Blöcken zwingend vorgeschrieben, da eine Klasse sowie eine Funktion bzw. eine Methode von einem Block begrenzt werden muss. Es ist aber auch erlaubt, Blöcke willkürlich zu setzen, ohne dass dafür eine syntaktische Notwendigkeit besteht.

```
int main(void)
{ // Block A
    ...
    { // Block B1
        ...
        { // Block C
            ...
        }
    }
}
```

```

        ...
    } // end C
    ...
} // end B1
...
{ // Block B2
    ...
} // end B2
...
return 0;
} // end A

```

Hier wurden – eingeleitet mit einem Kommentarzeichen – den einzelnen Blöcken Namen gegeben bzw. das Blockende kenntlich gemacht. Die Buchstaben (A, B, C) geben dabei die Verschachtelungsstruktur wieder. Blöcke mit gleichen Buchstaben befinden sich auf derselben Verschachtelungsebene. Die Auslassungspunkte (...) sollen anzeigen, dass an den jeweiligen Stellen Anweisungen stehen dürfen.

Eine im innersten Block (C) platzierte Anweisung ist ebenso den sie umgebenden Blöcken (A und B) zuzurechnen. Sie ist also wie grundsätzlich alle Anweisungen, die innerhalb von A stehen, Teil der Funktion `main()`.

Das willkürliche Setzen von Blöcken kann unter Umständen ein Mittel zur besseren Strukturierung des Quelltextes sein. In den meisten Fällen werden Sie Blöcke aber nur dann setzen, wenn es die Syntax von C++ vorschreibt.

## 7.4 Leerräume

Wie Sie weiter oben erfahren haben (Abschnitt 7.1 »Textbausteine«), ist es oft zwingend notwendig, einzelne Token durch Leerzeichen voneinander zu trennen.

Umgekehrt verlieren jedoch Namen (Bezeichner und Schlüsselwörter) oder Operatoren, sofern diese aus mehreren Einzelzeichen zusammengesetzt sind, durch das Hinzufügen von Zwischenraumzeichen ihre Bedeutung. Wenn Sie also schreiben

```
i n t main()
```

dann ist das Schlüsselwort `int` nicht mehr als solches erkennbar. Da der Ausdruck `i n t` auch nicht anderweitig interpretiert werden kann, resultiert aus obiger Programmierzeile eine Fehlermeldung.

Damit wird deutlich, dass Leerräume im Quelltext in genau zwei Varianten syntaktische Bedeutung erlangen:

- Trennung zweier Textbausteine durch Einfügen von Zwischenraumzeichen, sofern dies zu deren Unterscheidung notwendig ist (wie Sie im Abschnitt über Textbausteine erfahren haben, ist das nicht immer der Fall).
- Unerlaubtes Einfügen von Zwischenraumzeichen innerhalb eines Textbausteins



In allen anderen Zusammenhängen haben Leerräume im Quellcode keinerlei Bedeutung. Sie werden bei der Kompilierung einfach ignoriert. Das heißt, Sie dürfen so viele Zwischenraumzeichen (Leerzeichen, Tabulatorschritte und Zeilenumbrüche) einfügen, wie Sie wollen. Ebenso können Sie auch ganz darauf verzichten. Auf die Bedeutung des Quellcodes hat dies keinen Einfluss.

Es spielt also im Hinblick auf die Kompilierbarkeit und das Endergebnis – die ausführbare Datei – keine Rolle, ob Sie unser Einführungsprogramm in lesbarer Form – was aus Ihnen schon bekannten Gründen zu empfehlen ist – oder in Kurzform

```
#include <iostream>
using namespace std;int main(void){cout
<<"Willkommen bei C++";cout<<endl;return 0;}
```

oder etwa wie folgt eintippen:

```
#include <iostream>

using
        namespace
                std;

int
main
        (void)
{
    cout
        <<

        "Willkommen bei C++"
        ;
    cout
<<
endl    ;
        return 0;


}
```

## CD-ROM

Falls Sie sich von der genannten Tatsache selbst überzeugen wollen, finden Sie beide Quelltexte samt .exe-Datei als *k07kl* und *k07gr* auf der Buch-CD im Verzeichnis *Beispiele/K07*.

Dies ließe sich noch beliebig weitertreiben. Solange die Bedeutung der einzelnen Textbausteine dabei erhalten bleibt – was in beiden Listings der Fall ist –, ist der Programmiercode syntaktisch in Ordnung.

**Achtung**

Beachten Sie jedoch, dass Präprozessor-Direktiven – also Anweisungen, die mit # beginnen – immer in einer eigenen Zeile stehen müssen. Zudem ist das Einfügen von Newline-Zeichen innerhalb von Zeichenketten nicht erlaubt. Sie dürfen also Text, der zwischen "..." steht, nicht mittels  umbrechen.

Wenn Sie Leerräume einfügen müssen, weil es zur Unterscheidung der Token untereinander notwendig ist, gilt: Die Anzahl und die Art der Zwischenraumzeichen spielt keine Rolle. In der Regel ist im Hinblick auf die Lesbarkeit des Codes genau ein Leerzeichen zur Trennung optimal. Alle weiteren Zwischenraumzeichen werden vom Compiler ignoriert. Statt

```
int main(void)
```

ist es also ebenfalls erlaubt zu schreiben

```
int    main    (    void    )
```

Das Ganze lässt sich sogar auf mehrere Zeile verteilen, wie im obigen Listing zu sehen.

Nicht gestattet ist jedoch

```
int m a i n(void)
```

da der Name main, wie das Schlüsselwort int, ein nicht zerlegbares Token darstellt.

Da Sie nun wissen, wie Ihr C++-Compiler mit Leerräumen umgeht, möchten wir Sie noch einmal auf die herausragende Bedeutung des Semikolons und der geschweiften Klammern {} aufmerksam machen. Da C++ eben nicht zeilenorientiert ist, wird das Anweisungsende allein am Semikolon und das Blockende allein an der schließenden Klammer erkannt.

Von daher sind die Anweisungen

```
cout << "Willkommen bei C++";
```

und

```
cout << "Willkommen bei C++"
;

```

für den Compiler völlig gleichwertig.

Ebenso die Blockstrukturen

```
int main(void)
{
    cout << "Willkommen bei C++";
    return 0;
}
```

und

```
int main(void)
{cout << "Willkommen bei C++";
return 0;

}
```

Der Block ist erst mit dem Zeichen } zu Ende, und selbst wenn die schließende Klammer im Abstand von mehreren Bildschirmseiten folgt, wird sie vom Compiler als Blockende erkannt.

### **Tipp**

Gewöhnen Sie sich an, eine öffnende Klammer sogleich wieder zu schließen, bevor Sie an Ihrem Quelltext weiterarbeiten. Dies ist eine ebenso einfache wie nützliche Methode, um Fehler im Vorfeld zu verhindern.

Zum Abschluss dieser Ausführungen hier noch der ausdrückliche Hinweis: Ein wesentliches Kriterium für die Güte eines Quelltextes ist seine gute Nachvollziehbarkeit. Lassen Sie sich daher gerade von der oben dargestellten dreizeiligen Variante des Quellcodes nicht verführen und beherzigen Sie die im Folgenden genannten Programmierstilregeln.

## **7.5 Programmierstil**

Anwendungsprogramme können in den seltensten Fällen ohne Wartung über einen längeren Zeitraum hinweg eingesetzt werden. Einerseits ist es aufgrund äußerer Gegebenheiten oft notwendig, die Funktionalität einer Anwendung zu erweitern, andererseits kann das Auftreten logischer Fehler zu einer Überarbeitung des Quellcodes zwingen.

Daher ist es sinnvoll, den Quelltext von Anfang an wartungsfreundlich zu gestalten. Das ausschlaggebende Kriterium hierfür ist eine gute Nachvollziehbarkeit des Quellcodes. Im Idealfall muss dieser daher so eingerichtet sein, dass er auch von anderen Programmierern gelesen und verstanden werden kann.

Teilweise werden Sie es zukünftig mit sehr viel komplexeren Strukturen zu tun haben, als im letzten Beispiel des Abschnitts 7.3 »Blöcke« angedeutet. Es ist deshalb ratsam, sich schon zu Beginn einen guten Programmierstil anzugewöhnen. Unter diesem Gesichtspunkt seien hier folgende – selbst auferlegte – Regeln genannt:

- Schreiben Sie die Zeichen für den Blockanfang ( { ) und das Blockende ( } ) jeweils in eine eigene Zeile (von Ausnahmen bei kleineren Funktionen, Methoden usw. einmal abgesehen).
- Rücken Sie alle Anweisungen – und auch innere Blockstrukturen – innerhalb eines Blocks um eine konstante Anzahl von Leerzeichen ein. In unseren Listings sind dafür genau drei Stellen vorgesehen, Sie können aber genauso gut zwei oder vier Leerzeichen verwenden.

```
{  
    // Anweisung  
    { // innerer Block  
        // Anweisung(en), die sich  
        // im inneren Block befindet (befinden)  
    } // Ende innerer Block  
    // Anweisung  
}
```

- Beginnen Sie jede Anweisung möglichst in einer eigenen Zeile. Dagegen ergibt es Sinn, eine sehr lange Anweisung auf zwei oder mehr Zeilen zu verteilen.

Wenn Sie sich diesen Regeln nicht unterwerfen, hat das zwar keine Fehlermeldungen Ihres Compilers zur Folge. Wie gesagt, kann es unter gewissen Umständen auch einmal begründet sein, von diesen Empfehlungen abzuweichen. Ihre Einhaltung wird Ihnen jedoch im Allgemeinen das Programmieren erleichtern.

# 8 Ausgabe mit cout

In diesem Kapitel werden wir uns das Objekt `cout` etwas näher ansehen. Sie erfahren dabei einiges Wissenswertes über Datenströme, Manipulatoren und die Möglichkeit, mehrere Ausgabeeinheiten innerhalb einer Anweisung aneinander zu hängen. In diesem Zusammenhang wird auch die Bedeutung der Datentypen in C++ zum ersten Mal angesprochen.

## 8.1 Datenströme

Wie Sie bereits wissen, erfolgt die Datenausgabe auf den Bildschirm (genauer: auf das Standardausgabegerät) mit dem Objekt `cout`. In diesem Zusammenhang spricht man von einem Datenstrom und es ist nicht verkehrt, sich einen Datenfluss zum Objekt `cout` hin vorzustellen. `cout` steht dabei für das Standardausgabegerät, was in der Regel der Bildschirm ist. Dies wird auch durch die Richtung des Ausgabeoperators `<<` deutlich, dessen Spitze auf `cout` zeigt.

```
cout << "blablabla";
```

### Hinweis

Das Gegenstück zu `cout` ist `cin`. Es steht für den Datenfluss in der umgekehrten Richtung, also für Eingaben des Benutzers. Bei `cin` handelt es sich ebenfalls um ein Objekt einer speziellen vordefinierten Klasse, die die Funktionalität für die Dateneingabe bereitstellt. `cin` steht dabei für die Standardeingabe, also in der Regel für die Tastatur. Demzufolge weist die Spitze des Eingabeoperators `>>` in die entgegengesetzte Richtung, von `cin` weg.

```
cin >> ...
```

Das Objekt `cin` lernen Sie in Kapitel 10, Abschnitt 10.4 »Das Objekt `cin`« kennen.

### Hinweis

Das »c« in `cout` sowie in `cin` steht jeweils für »character«, zu Deutsch »Zeichen«. Sprechen Sie also »character out« bzw. »character in«. Das englische Wort »out« (Deutsch »aus«) in `cout` bezeichnet die Datenausgabe, das Wort »in« (Deutsch »ein«) in `cin` die Dateneingabe.

## 8.2 C++ unterscheidet zwischen verschiedenen Datentypen

Im weiteren Verlauf des Buches wird Ihnen der Gebrauch von Variablen und Konstanten schnell zur Selbstverständlichkeit werden. Beide charakterisiert ihre Zugehörigkeit zu einem bestimmten Datentyp. So gibt es in C++ spezielle Datentypen für ganze Zahlen, Zahlen mit Nachkommastellen, Wahrheitswerte, einfache Zeichen und Zeichenfolgen.

Variablen lernen Sie in Kapitel 10 »Variablen« kennen. Konstante Werte eines bestimmten Datentyps haben Sie bereits benutzt. Es handelt sich um Zeichenketten. Eine Zeichenkette ist, wie Sie ja bereits wissen, die Gesamtheit aller Zeichen, die von doppelten Anführungszeichen ("...") begrenzt werden.

Zeichenketten werden, vielleicht etwas präziser, auch Zeichenkettenkonstanten genannt. Denn es handelt sich schließlich um Konstanten. Wegen des unangenehm langen Wortes »Zeichenkettenkonstanten« spricht man lieber von »Stringkonstanten«, nach dem englischen Wort »string«, was so viel heißt wie »Zeichenfolge« bzw. in der Computersprache: Folge von Zeichen, die gemeinsam verarbeitet werden. Um sich die Sache noch einfacher zu machen, werden Zeichenketten mitunter einfach nur »Strings« genannt.

```
"Willkommen bei C++"
```

oder

```
"Hallo"
```

sind also Zeichenketten, Zeichenkettenkonstanten, Stringkonstanten oder Strings. Suchen Sie sich die Bezeichnung aus, die Ihnen am liebsten ist.

Nun besitzt, wie oben erwähnt, jeder konstante Wert einen bestimmten Datentyp und eine Zeichenkette ist ein konstanter Wert. Ausgerechnet mit Zeichenketten verhält es sich aber gar nicht so einfach. In C++ existiert nur ein elementarer Datentyp für einzelne Zeichen, genannt `char`. Und eine Zeichenkette ist eigentlich ein Array von Werten des Typs `char` (zu Arrays siehe Kapitel 18 »Arrays«).

Um uns die Sache etwas einfacher und anschaulicher zu machen, wollen wir fürs Erste die Feststellung treffen, der Datentyp von Zeichenketten sei `string` (kleingeschrieben).

### Hinweis

Wir sagten oben, in C++ gibt es Datentypen für ganze Zahlen, Zahlen mit Nachkommastellen, einfache Zeichen und Zeichenfolgen. Das trifft zwar zu, elementare Datentypen gibt es jedoch nur für Zahlen, Wahrheitswerte und einfache Zeichen. Elementare Datentypen, auch »Standarddatentypen« genannt, sind solche, die unmittelbar zur Programmiersprache gehören. Es handelt sich sozusagen um eingebaute Datentypen.

**Hinweis (Forts.)**

Bei dem Datentyp `string` (für Zeichenketten) handelt es sich eigentlich um eine vordefinierte Klasse. Wir werden aber fürs Erste so tun, als sei dieser Datentyp ebenfalls elementar, zumal das zum jetzigen Zeitpunkt für Sie keinen Unterschied macht. Genaueres werden Sie erfahren, wenn wir uns im Zuge der objektorientierten Programmierung mit Klassen beschäftigen.


**Hinweis**

Man sagt, ein konstanter Wert besitzt einen bestimmten Datentyp bzw. er ist von diesem Datentyp. Folglich gilt: Strings (Zeichenketten) besitzen den Datentyp `string` bzw. Zeichenketten sind vom Datentyp `string`. Beachten Sie, dass der Name des Datentyps (`string`) kleingeschrieben wird. `String`, großgeschrieben, ist dagegen eine synonyme Bezeichnung für »Zeichenkette« bzw. für Variablen des Datentyps `string` (zu letzteren siehe Kapitel 10, Abschnitt 10.3.3 »string-Variablen«).

Im Unterschied zum Datentyp `string` sei hier der Datentyp `char` genannt. Er ist für die Verarbeitung von einzelnen Zeichen zuständig.

Wie konstante Werte vom Datentyp `string` im Quelltext dargestellt werden, wissen Sie bereits: mittels Begrenzung durch die Zeichen `"` und `"`. Wie ist das nun mit konstanten Werten vom Datentyp `char`? Die Antwort ist denkbar einfach: Als Begrenzungszeichen dienen einfache Apostrophe (`'`).

**Hinweis**

Das Zeichen `'` wird durch die Kombination von  und der Taste, auf der sich die Raute (`#`) – und natürlich das `'` – befindet, dargestellt.

```
'A'
'Z'
'z'
```

und auch

```
'&'
```

oder

```
'!'
```

sind also konstante Werte vom Datentyp `char`.

Kommen wir wieder auf `cout` zu sprechen. Eine Besonderheit von `cout` ist, dass Sie es, ohne dabei viel beachten zu müssen, zur Ausgabe von Werten aller elementaren Daten-

typen (einschließlich string) heranziehen können. Folgende Anweisungen sind demnach beide korrekt:

```
cout << "ABC";  
cout << 'D';
```

In der ersten Anweisung wird ein String (Datentyp: string) ausgegeben, in der zweiten ein einfaches Zeichen (Datentyp: char). Mit anderen Worten: In der ersten Anweisung wird ein konstanter Wert vom Typ string, in der zweiten ein konstanter Wert vom Typ char ausgegeben.

### Hinweis

Anstelle vom »Datentyp« spricht man auch einfach nur vom »Typ« (einer Konstanten, einer Variablen).

Verwendet man beide Ausgabeanweisungen nacheinander im Quellcode, so erfolgt die zweite Ausgabe (cout << 'D';) an der Stelle, an der sich der Cursor befindet, also unmittelbar nach dem C. Das Programm

```
#include <iostream>  
using namespace std;  
  
int main(void)  
{  
    cout << "ABC";  
    cout << 'D';  
    return 0;  
}
```

erzeugt somit – nach Kompilierung und Ausführung – die Ausgabe

```
ABCD
```

### Hinweis

Das gleiche Ergebnis ist natürlich ebenso mit einer einzigen Anweisung zu erzielen:

```
#include <iostream>  
using namespace std;  
  
int main(void)  
{  
    cout << "ABCD";  
    return 0;  
}
```

Ausgabe: ABCD



## 8.3 Ausgabeeinheiten aneinander hängen

Durch mehrfache Verwendung des Ausgabeoperators ist es möglich, mehrere Ausgabeeinheiten in einer einzigen Anweisung zu verarbeiten. Dabei darf sich auch der Datentyp von Ausgabeeinheit zu Ausgabeeinheit ändern.

```
cout << "Ha" << 'l' << "lo";
```

Obige Anweisung bewirkt die Ausgabe

```
Hallo
```

Statt

```
cout << "ABC";  
cout << 'D';
```

im obigen Beispiel hätten wir also auch schreiben können:

```
cout << "ABC" << 'D';
```

### Hinweis

Und selbstverständlich darf auch Folgendes geschrieben werden:

```
cout << "ABCD";
```

Beachten Sie, dass der Name `cout` innerhalb einer Anweisung nur ein einziges Mal, zu Beginn der Anweisung, verwendet werden darf. Alle weiteren Ausgabeeinheiten werden mit dem bloßen `<<` eingeleitet.

### 8.3.1 In der Programmierung gibt es meist mehrere Lösungen

Denken Sie einmal daran, wie viele Möglichkeiten Ihnen nun zur Verfügung stehen, um eine Zeichenfolge, sagen wir `Hallo`, auf den Bildschirm zu bringen. Folgendes Listing erscheint wohl diesbezüglich am naheliegendsten:

```
#include <iostream>  
using namespace std;  
  
int main(void)  
{  
    cout << "Hallo";  
    return 0;  
}
```

Ausgabe: `Hallo`

Die Anweisung

```
cout << "Hallo";
```

lässt sich aber auch durch

```
cout << "Ha" << 'l';  
cout << "lo";
```

oder aneinander gereiht in einer Anweisung

```
cout << "Ha" << 'l' << "lo";
```

oder gar

```
cout << 'H';  
cout << 'a';  
cout << 'l';  
cout << 'l';  
cout << 'o';
```

oder wiederum verkettet

```
cout << 'H' << 'a' << 'l' << 'l' << 'o';
```

ersetzen. Dies ließe sich, in Bezug auf die Zeichenkette "Hallo" vielleicht nicht unendlich, aber doch sehr lange, weitertreiben, um alle zur Verfügung stehenden Möglichkeiten auszuschöpfen. Das Ergebnis bliebe stets das gleiche.

### Hinweis

Im Hinblick auf eine gute Lesbarkeit des Quellcodes scheiden jedoch einige Varianten aus.

Dies soll uns schon jetzt Anlass zu der zwar trivialen, aber mitunter sehr bedeutsamen Feststellung geben, dass es in der Programmierung in den seltensten Fällen nur einen einzigen richtigen Weg gibt. Vielmehr ist für die Realisierung eines Programms im Allgemeinen eine Vielzahl von möglichen Implementationen denkbar.


## 8.3.2 Anweisungen auf mehrere Zeilen verteilen

Abgesehen vom Wechseln des Datentyps bietet die mehrfache Verwendung des Ausgabeoperators hinsichtlich der Übersichtlichkeit des Quellcodes eine bequeme Möglichkeit, sehr lange Anweisungen auf mehrere Zeilen zu verteilen.

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

int main(void)
{
    cout << "Dies ist eine sehr lange "
          << "Anweisung. Deshalb ist sie"
          << " auf mehrere Programmier"
          << "zeilen verteilt";
    return 0;
}
```

### Achtung

Es sei daran erinnert, dass im Quelltext das Drücken der Taste  innerhalb von Zeichenketten nicht erlaubt ist. Das damit verbundene Zeichen ist vom Compiler nicht interpretierbar.

### Hinweis

Möglicherweise wundern Sie sich darüber, dass in der Anweisung

```
cout << "Dies ist eine sehr lange "
      << "Anweisung. Deshalb ist sie"
      << " auf mehrere Programmier"
      << "zeilen verteilt";
```

die weiteren Ausgabeeinheiten nicht am Zeilenanfang beginnen, sondern gegenüber dem Namen `cout` eingerückt sind. Damit soll hervorgehoben werden, dass diese Programmzeilen zu ein und derselben Ausgabeanweisung gehören. Im Sinne einer guten Lesbarkeit verfahren die meisten Programmierer in der gleichen Weise. Letzten Endes handelt es sich dabei jedoch um eine Geschmacksfrage.

## 8.3.3 Verkettung von Strings

Soweit es sich ausschließlich um Zeichenketten handelt, kann auf die Verwendung weiterer Ausgabeoperatoren sogar verzichtet werden. Die Zeichenketten können – ohne `<<` – einfach aneinander gehängt werden. Der Compiler behandelt sie dann automatisch wie eine einzige Zeichenkette.

```
cout << "Dies ist eine sehr lange "
      "Anweisung. Deshalb ist sie"
      " auf mehrere Programmier"
      "zeilen verteilt";
```

**Hinweis**

Das Verfahren, mehrere Zeichenketten aneinander zu hängen, wird »Verkettung« genannt. Man spricht von Stringverkettung bzw. Verkettung von Stringkonstanten.

**Hinweis**

Die Verkettung von Strings ist grundsätzlich an jeder Stelle im Code möglich. Da Zwischenraumzeichen (Leerzeichen, Tabulatorschritte und Zeilenumbrüche), soweit sie die Bedeutung der einzelnen Textbausteine nicht beeinflussen, im Quelltext keine Rolle spielen, steht es Ihnen grundsätzlich frei, zwei Strings unmittelbar aneinander zu ketten oder dazwischen ein oder mehrere Zwischenraumzeichen (gar einen Zeilenumbruch und mehrere Leerzeichen, wie in der eben genannten Anweisung) zu setzen, also:

```
cout << "Will""kommen";

oder

cout << "Will" "kommen";

oder eben

cout << "Will"
      "kommen";
```

Es sei ausdrücklich darauf hingewiesen, dass die Möglichkeit der Verkettung ausschließlich für Zeichenketten gilt, nicht für Werte anderen Datentyps. Folgende Anweisung ist demnach fehlerhaft:

```
cout << 'A' 'B'; // FEHLER
```

## 8.4 Manipulatoren

Wenn Sie das Programm

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

int main(void)
{
    cout << "Dies ist eine sehr lange "
          << "Anweisung. Deshalb ist sie"
          << " auf mehrere Programmier"
          << "zeilen verteilt";
    getchar();
    return 0;
}
```

bzw.

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

int main(void)
{
    cout << "Dies ist eine sehr lange "
           "Anweisung. Deshalb ist sie"
           " auf mehrere Programmier"
           "zeilen verteilt";
    getchar();
    return 0;
}
```

kompilieren und ausführen, erhalten Sie möglicherweise eine ziemlich unschöne Ausgabe.

## CD-ROM

Es handelt sich um die Programme *k08e* bzw. *k08f* auf der Buch-CD im Ordner *Beispiele/K08*.



Abbildung 8.1: Ausgabe ohne Zeilenumbrüche

Mit der Ausgabe sollte eigentlich dasselbe geschehen, was im Quellcode mit der entsprechenden Ausgabeanweisung erreicht wurde: Die Ausgabe sollte auf mehrere Zeilen verteilt werden. (Der Zeilenumbruch in Abbildung 8.1 ist natürlich nicht durch die Programmierung, sondern allein durch die Größe des Konsolenfensters bedingt.)

Dazu ist es notwendig, an den geeigneten Stellen ein so genanntes Newline-Zeichen einzufügen. Dazu stehen Ihnen zwei Möglichkeiten zur Verfügung: die Verwendung des Steuerzeichens `\n` oder des Manipulators `endl`. Steuerzeichen bilden das Thema des nächsten Kapitels. Hier werden wir die Zeilenumbrüche mit `endl` umsetzen.

## Hinweis

Das »l« in endl steht für line (zu Deutsch »Zeile«). Sprechen Sie also »end line«.

Die Ausgabe von endl bewirkt an jeder Stelle, an der dieser Manipulator verwendet wird, einen Zeilenvorschub. Manipulatoren werden so genannt, weil sie auf die Ausgabe mit cout einwirken, sie sozusagen verändern, »manipulieren«.

Um die gewünschte Ausgabe – mit Zeilenumbrüchen – zu erhalten, könnte Ihr Quelltext in etwa so aussehen:

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

int main(void)
{
    cout << "Dies ist eine sehr lange Anweisung."
          << endl << "Deshalb ist sie auf mehrere "
          << endl << "Programmierzeilen verteilt"
          << endl << endl;
    getchar();
    return 0;
}
```

Beachten Sie, dass jede Verwendung eines Manipulators eine eigene Ausgabeinheit erfordert.

Die Ausgabe zeigt sich nun wie gewünscht (Abbildung 8.2).



**Abbildung 8.2:** Ausgabe mit Zeilenumbrüchen

Neben endl gibt es weitere wichtige Manipulatoren, unter anderem zur Steuerung von Zahlenausgaben. Diese werden Sie im weiteren Verlauf noch kennen lernen (Kapitel 13 »Ausgabe mit Manipulatoren formatieren«).

# 9

# Steuerzeichen

Nicht alle Zeichen sind druckbar oder mit der Tastatur zu erreichen. Zudem lassen sich manche Zeichen nicht in der gewohnten Form direkt in den Quellcode schreiben, da ihnen im Kontext eine besondere Bedeutung zugeordnet ist. Das ist etwa bei dem Zeichen " innerhalb von Zeichenketten der Fall, da dieses dort als Begrenzungszeichen fungiert.

Aus diesen Gründen stellt C++ für die Ausgabe eine Reihe von Steuerzeichen, auch »Escape-Sequenzen« genannt, zur Verfügung. Damit ist der Programmierer in der Lage, z.B. einen Signalton, Tabulatorschritte oder Zeilenumbrüche zu erzeugen oder etwa den Cursor um eine oder mehrere Stellen zurückzusetzen.

## 9.1 Die Escape-Sequenzen \", \' und \\

In Abwandlung unseres ersten Programms, das die Ausgabe

```
Willkommen bei C++
```

bewirkt, möchten wir nun die Ausgabe

```
"Willkommen bei C++"
```

erzielen. Die Zeichen " sollen also ebenfalls auf dem Bildschirm erscheinen.

Nichts – abgesehen von Fehlermeldungen des Compilers – erreichen wir, wenn wir die Zeichen wie die anderen direkt in die Zeichenkette schreiben.

```
// Fehlerhafter Code
#include <iostream>
using namespace std;

int main(void)
{
    cout << "Willkommen bei C++"; // FEHLER
    cout << endl;
    return 0;
}
```

Um sich mit der Interpretationsweise eines C++-Compilers vertraut zu machen, rentiert es sich jedoch, die – fehlerhafte – Anweisung

```
cout << "Willkommen bei C++";
```

einmal unter die Lupe zu nehmen. Nach `cout <<` erwartet der Compiler die Angabe eines Wertes, z.B. einer Zeichenkette. Und das erste `"` wertet er auch als Beginn einer solchen. Nun wird er alle folgenden Zeichen als Teil des Strings ansehen – bis zum Auftreten eines weiteren doppelten Anführungszeichens. Das dem ersten `"` folgende doppelte Anführungszeichen wird also automatisch als Zeichenkettenende interpretiert. Somit ergibt sich hier als Zeichenkette `"`, und eine Anweisung wie

```
cout << "";
```

ist auch syntaktisch korrekt, obwohl sie beim Programmlauf herzlich wenig bewirkt.

### Hinweis

Die Zeichenkette `""` hat sogar einen speziellen Namen. Sie heißt *leere Zeichenkette*, da in ihr keine Zeichen enthalten sind. Die leere Zeichenkette ist in gewisser Weise das Pendant zur Zahl 0.

Was den Compiler irritiert, sind die Textbausteine, die dem String `""` folgen:

```
Willkommen bei C++";
```

denn mit den Namen `Willkommen`, `bei` und `C++` kann er nicht viel anfangen. Es gibt im Quellcode keine Variablen, Funktionen oder Klassen, die so heißen, und als Teil einer Zeichenkette sind sie ebenfalls nicht zu begreifen, da ein entsprechendes Anfangszeichen (`"`) im Code nicht vorhanden ist. Also wird der Compiler möglicherweise jeden Namen, den er nicht versteht, einzeln monieren. Und die darauf folgende Zeichenkette `""` (nach `C++`) passt ebenfalls nicht in den Kontext. Tatsächlich beantworten manche C++-Compiler den Kompilerversuch mit insgesamt sieben Fehlermeldungen (Borland beschränkt sich auf nur eine – mit korrekter Zeilenangabe).

Wie machen wir dem Compiler nun verständlich, dass er bestimmte `"` nicht als Begrenzungszeichen, sondern als »normale« Zeichen behandeln soll? Die Antwort ist denkbar einfach. Wir stellen besagten Zeichen einen Backslash `\` (also einen umgekehrten Schrägstrich) voran.

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

int main(void)
{
    cout << "\\Willkommen bei C++\\";
    cout << endl;
    return 0;
}
```



Nun ist alles in Ordnung und das Programm zeigt, wenn es ausgeführt wird, die Ausgabe "Willkommen bei C++".

### CD-ROM

Die Programme auf der Buch-CD sind im Weiteren mit einem Haltebefehl (Anweisung `getchar();`) versehen, damit Sie sie auch über den Windows-Explorer ausführen können.

Escape-Sequenzen werden stets mit einem Backslash eingeleitet. Dieser nimmt hier dem darauf folgenden doppelten Anführungszeichen praktisch die Sonderbedeutung. Beachten Sie, dass es sich bei \" für den Compiler um nur ein Zeichen handelt, auch wenn dieses tatsächlich aus zwei Zeichen zusammengesetzt ist.

### Hinweis

Denken Sie an den Ausgabeoperator `<<`, der ebenfalls aus zwei Zeichen (`<`) besteht.

Kritisch ist allerdings nur die Verwendung von " innerhalb von Zeichenketten, nicht in Werten vom Typ `char`. Wenn Sie das " als einfaches Zeichen im Code verwenden, steht es Ihnen frei, entweder

```
cout << ' ';
```

oder

```
cout << '\\';
```

zu schreiben. Demnach hätten wir das gewünschte Ergebnis auch wie folgt erreicht:

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

int main(void)
{
    cout << ' ' << "Willkommen bei C++" << ' ';
    cout << endl;
    return 0;
}
```

Ausgabe: "Willkommen bei C++"

Solange ein auszugebendes Zeichen nicht mit den Begrenzungszeichen korreliert, kommt ihm auch keine Sonderbedeutung zu. Daher dürfen Sie auch das einfache Anführungszeichen (') ohne weiteres in einer Zeichenkette verwenden:

```
cout << "Sie sagte, '...' . ";
```

Als einfaches Zeichen notiert, müssen Sie es dagegen maskieren:

```
cout << '\\';
```

Einem Zeichen kommt allerdings stets eine Sonderbedeutung zu, nämlich dem Backslash selbst. Um einen Backslash auszugeben, muss diesem folglich in jedem Fall ein weiterer Backslash vorangestellt werden:

Die Anweisungen

```
cout << "\\C++\\";
```

oder

```
cout << '\\\' << "C++" << '\\';
```

erzeugen beide die Ausgabe

```
\\C++\\
```

## 9.2 ASCII-Code

Rechnerintern wird jedes druckbare und nicht druckbare Zeichen in binärer Form als ganzzahliger Wert dargestellt. Welcher Zahlenwert einem bestimmten Zeichen zugeordnet ist, hängt davon ab, welchen Zeichensatz verwendet wird. Einer der wichtigsten Zeichensätze ist der so genannte ASCII-Zeichensatz (American Standard Code for Information Interchange, zu Deutsch »Amerikanischer Standardcode für Informationsaustausch«). In diesem Zeichensatz wird beispielsweise das Zeichen »A« in binärer Schreibweise als 0100 0001 dargestellt, was dem Dezimalwert 65 und dem Hexadezimalwert 41 entspricht (zu Hexadezimalwerten gleich mehr).

Der vollständige ASCII-Code besteht aus 256 Zeichen, beginnend mit der Ordnungszahl 0. Jedem Zeichen ist somit eine Ordnungszahl im Bereich von 0 bis 255 zugeordnet. Wirklich standardisiert sind eigentlich nur die ersten 128 Zeichen des ASCII-Codes (Tabelle 9.1).

In diesem Bereich des ASCII-Codes sind nahezu alle Zeichen enthalten, die auf direktem Wege über die Tastatur erreichbar sind.

Das Zeichen mit der Ordnungsnummer 127 steht für das mit der Taste **Entf** verbundene Zeichen (DEL steht für »delete«, also »löschen«). Beim Zeichen mit der Ordnungszahl 32 handelt es sich um das Leerzeichen (SP steht für »space«).

Die ersten 32 Zeichen (Bereich 0 bis 31) sind – nicht druckbare – Steuerzeichen. Sie dienen vornehmlich zur Steuerung und Formatierung von Ein- und Ausgabe.

Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	\$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Tabelle 9.1: ASCII-Zeichensatz (Ordnungszahlen 0 bis 127)

### Tipp

Um auch andere Zeichen darzustellen, z.B. die des erweiterten ASCII-Codes, drücken Sie die **[Alt]**-Taste und geben gleichzeitig die entsprechende Ordnungszahl auf dem Nummernblock ein. Allerdings muss dieser aktiviert sein (drücken Sie gegebenenfalls die **[NumLock]**-Taste).

Auf diese Weise können Sie grundsätzlich jedes druckbare Zeichen des ASCII-Codes darstellen. Beispielsweise erhalten Sie das »Z« (Ordnungsnummer 90) mit der Tastenkombination **[Alt]+[9]+[0]**, wobei **[9]** und **[0]** auf dem – aktivierten – Nummernblock gedrückt werden müssen. Analog dazu wird das Zeichen »@« (Ordnungsnummer 64) mit **[Alt]+[6]+[4]** eingefügt.

### Hinweis

Machen Sie sich klar, dass, wenn hier von Steuerzeichen die Rede ist, die binäre Repräsentation dieser Zeichen im Computer gemeint ist. Auf der anderen Seite stellt C++ eine Reihe von Symbolen (C++-Steuerzeichen) wie etwa `\n`, `\t` oder `\a` zur Verfügung, um diese Steuerzeichen im Quellcode darstellen zu können (dazu gleich mehr).

Es sei darauf hingewiesen, dass einige dieser 32 Steuerzeichen heutzutage keine praktische Bedeutung mehr haben. Auf die wichtigsten werden wir gleich eingehen. Interessierte können Tabelle 9.2 eine kurze (formale) Beschreibung der ersten 32 Zeichen entnehmen.

Dec.	Hex.	Zeichen	Englisch	Deutsch
0	00	NUL	null	Nullzeichen
1	01	SOH	start of heading	Kopfzeilenbeginn
2	02	STX	start of text	Textanfangzeichen
3	03	ETX	end of text	Textendezeichen
4	04	EOT	end of transmission	Übertragungsende
5	05	ENQ	enquiry	Aufforderung zur Datenübertragung
6	06	ACK	acknowledge	positive Rückmeldung
7	07	BEL	bell	Signalton
8	08	BS	backspace	Rückwärtsschritt
9	09	HT	horizontal tab	horizontaler Tabulator
10	0A	LF	line feed	Zeilenvorschub
11	0B	VT	vertical tab	vertikaler Tabulator
12	0C	FF	form feed	Seitenvorschub
13	0D	CR	carriage return	Wagenrücklauf

**Tabelle 9.2:** Beschreibung der Steuerzeichen

Dec.	Hex.	Zeichen	Englisch	Deutsch
14	0E	SO	shift out	Dauerumschaltung
15	0F	SI	shift in	Rückschaltung
16	10	DLE	data link escape	Datenübertragungsumschaltung
17	11	DC1	device control 1	Gerätesteuerzeichen 1
18	12	DC2	device control 2	Gerätesteuerzeichen 2
19	13	DC3	device control 3	Gerätesteuerzeichen 3
20	14	DC4	device control 4	Gerätesteuerzeichen 4
21	15	NAK	negative acknowledge	negative Rückmeldung
22	16	SYN	synchronous idle	Synchronisierung
23	17	ETB	end of transmission block	Ende des Übertragungsblocks
24	18	CAN	cancel	ungültig
25	19	EM	end of medium	Ende der Aufzeichnung
26	1A	SUB	substitute	Substitution
27	1B	ESC	escape	Umschaltung
28	1C	FS	file separator	Hauptgruppentrennzeichen
29	1D	GS	group separator	Gruppentrennzeichen
30	1E	RS	record separator	Untergruppentrennzeichen
31	1F	US	unit separator	Teilgruppentrennzeichen

**Tabelle 9.2:** Beschreibung der Steuerzeichen (Forts.)

Mit dem so genannten Nullbyte (das Zeichen mit der Ordnungsnummer 0) werden Sie im Zusammenhang mit Zeichenketten noch in Berührung kommen (Kapitel 19, Abschnitt 19.1 »Wie Zeichenketten dargestellt werden«). Dargestellt wird es im Quellcode mit dem C++-Steuerzeichen `\0`.

Bezüglich des Bereichs mit den Ordnungszahlen von 128 bis 255, auch erweiterter ASCII-Code genannt, existieren verschiedene Variationen (die Ordnungsnummern der Umlaute sowie des Zeichens »ß« sind jedoch einheitlich).

## 9.3 Darstellung von Zeichen mittels Escape-Sequenzen

Grundsätzlich kann jedes Zeichen aus dem verwendeten Zeichensatz im C++-Quellcode auch durch eine Escape-Sequenz dargestellt werden. Dazu leiten Sie die Escape-Sequenz mit `\x` ein, gefolgt von der entsprechenden Hexadezimalzahl (Spalte »Hex.« in der obigen ASCII-Tabelle).

Im folgenden Beispiel werden das erste doppelte Anführungszeichen und das `C` in der Zeichenkette durch eine Escape-Sequenz repräsentiert:

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

int main(void)
{
    cout << "\x22Willkommen bei \x43++\n";
    cout << endl;
    return 0;
}
```

Ausgabe:

```
"Willkommen bei C++"
```

## Hexadezimalsystem

Unser vertrautes Dezimalsystem, dem die Basis 10 zugrunde liegt, kommt mit den Ziffern 0 bis 9 aus. Da das Hexadezimalsystem mit der Basis 16 arbeitet, sind neben den ersten zehn Ziffern weitere Zeichen zur Darstellung notwendig. Hierfür verwendet man die Buchstaben A bis F. Die aufsteigende Reihenfolge der »Zahlen« im Hexadezimalsystem ist also: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Daher ist z.B. F bzw. 0F die hexadezimale Darstellung des Dezimalwertes 15. In gleicher Weise entspricht die hexadezimale »Zahl« A bzw. 0A der Zahl 10 des Dezimalsystems.

Der Grund, warum das auf den ersten Blick unpraktische Hexadezimalzahlssystem bei Computern eine so große Rolle spielt, liegt darin, dass die Basis 16 eine Zweierpotenz ist ( $2^4$ ) und damit hervorragend mit der binären Arbeitsweise des Computers harmonisiert (der Prozessor kann bekanntlich nur zwei verschiedene Werte – 0 und 1 – verarbeiten). 10 – die Basis des Dezimalsystems – stellt dagegen keine Zweierpotenz dar. Gewissermaßen ist das Hexadezimalsystem nur eine andere Sichtweise von binären Werten (wie 01000001), genauer lassen sich hexadezimale und binäre Werte sehr leicht untereinander umrechnen – was für die Umrechnung binär nach dezimal und umgekehrt nicht gilt. Daher werden Sie bei der Programmierung oft auf das Hexadezimalsystem stoßen.

Hier ein weiteres Beispiel:

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

int main(void)
{
    cout << '\x43';
    cout << '\x2B';
    cout << '\x2B';
    cout << endl;
}
```

```
    return 0;
}
```

Ausgabe:

```
C++
```

### Hinweis

Denken Sie daran, dass es sich auch bei Escape-Sequenzen wie `\x2B` oder `\x43` um ein einziges Zeichen handelt. Daher ist die Notierung als `char`-Wert (`'\x2B'`) unproblematisch.

## 9.3.1 Ausgabe von Umlauten sowie des Zeichens »ß«

Leider verwendet die Konsole einen eigenen Zeichensatz, den so genannten OEM-Code, der aber in den ersten 128 Zeichen mit dem ASCII-Zeichensatz übereinstimmt.

Daher ist es nicht auf dem üblichen Weg möglich, Umlaute sowie den Buchstaben ß, denen im ASCII-Code Ordnungszahlen zwischen 129 und 225 zugeordnet sind, auf der Konsole auszugeben. Sie können sich natürlich dadurch behelfen, dass Sie die Umlaute mit `ae`, `ue`, `oe` bzw. mit `Ae`, `Ue`, `Oe` und das »ß« als `ss` wiedergeben. Um dennoch auf den ASCII-Zeichensatz zurückzugreifen, müssen Sie besagte Zeichen wie oben dargelegt als Escape-Sequenz mit der zugeordneten hexadezimalen Zahl und einem vorangestellten `x` in den Quellcode schreiben. Die entsprechenden Werte sind Tabelle 9.3 zu entnehmen.

Dez.	Hex.	Zeichen
129	81	ü
132	84	ä
142	8E	Ä
148	94	ö
153	99	Ö
154	9A	Ü
225	E1	ß

**Tabelle 9.3:** Umlaute und ß (erweiterter ASCII-Code)

Hierzu ebenfalls ein Beispiel:

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

int main(void)
{
```

```
cout << "\x84, \x94, \xE1" << endl;
cout << "\x8E, \x99, \x9A" << endl;
cout << "und \xE1" << endl;
return 0;
}
```

Ausgabe:

```
ä, ö, ü
Ä, Ö, Ü
und ß
```

Es macht keinen Unterschied, ob Sie bei der Notierung von Hexadezimalwerten Groß- oder Kleinbuchstaben verwenden. Statt `\xE1` ist `\xe1` ebenso möglich.

### Achtung

Bedenken Sie jedoch, dass eine Hexadezimalzahl erst mit dem ersten Zeichen, das nicht als Teil einer Hexadezimalzahl interpretiert werden kann, zu Ende ist. Um das Wort »öffentlich« auszugeben, ist es demnach unzulässig, Folgendes zu schreiben:

```
cout << "\x94ffentlich"; // FEHLER
```

Dies liegt daran, dass `f` – und auch `e` – als hexadezimale »Ziffern« Verwendung finden. Ihr Borland-Compiler beschwert sich in der obigen Anweisung lapidar über die zu hohe Anzahl von (hexadezimalen) Ziffern. Andere Compiler gehen mehr ins Detail und antworten mit der Meldung, dass `'610302'` (entspricht `'\x94ffe'`) zu groß für ein Zeichen ist.

Schaffen Sie nötigenfalls mit mehreren Ausgabeeinheiten Abhilfe:

```
cout << '\x94' << "ffentlich"; // OK
```

oder schreiben Sie

```
cout << "\x94""ffentlich"; // OK
```

Ausgabe:

```
öffentlich
```

## 9.4 Das Steuerzeichen `\n`

Im Abschnitt 8.4 (»Manipulatoren«) des letzten Kapitels haben Sie mit dem Manipulator `endl` bereits eine Möglichkeit kennen gelernt, Zeilenumbrüche zu erzeugen. Alternativ dazu können Sie das C++-Steuerzeichen `\n` verwenden. Dieses dürfen Sie auch direkt in die Zeichenkette schreiben.



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

int main(void)
{
    cout << "Dies ist eine sehr lange Anweisung."
          << "\nDeshalb ist sie auf mehrere "
          << "\nProgrammierzeilen verteilt.\n"
          << '\n';
    cout << "ENDE\n";
    return 0;
}
```

Ausgabe:

```
Dies ist eine sehr lange Anweisung.
Deshalb ist sie auf mehrere
Programmierzeilen verteilt.

ENDE
```

Die Zeile

```
<< '\n';
```

soll lediglich verdeutlichen, dass das Steuerzeichen \n ebenso als einfache Zeichenkonstante (Typ char) geschrieben werden kann. Dies gilt natürlich für jedes Zeichen, egal ob es sich um ein »normales« oder um ein Steuerzeichen handelt.

Anstelle von

```
...
    << "\nProgrammierzeilen verteilt.\n"
    << '\n';
...
```

hätten wir also mit dem gleichen Ergebnis auch schreiben können

```
...
    << "\nProgrammierzeilen verteilt.\n\n";
...
```

Jedes \n erzeugt genau einen Zeilenvorschub.

**Hinweis**

Zum Verständnis: `\n` ist ein C++-Steuerzeichen zur Darstellung des ASCII-Zeichens mit der Ordnungsnummer 10 (hexadezimaler Wert 0A). Nach dem oben Gesagten ist es natürlich auch möglich, dieses Zeichen im Code mit `\x0A` zu notieren. Was kein Programmierer tut, nicht nur wegen des minimal erhöhten Aufwands, sondern auch, weil es die Lesbarkeit des Quellcodes gegenüber `\n` bzw. `endl` nicht gerade verbessert.

**Hinweis**

Obwohl noch aus alten C-Zeiten stammend, steht die Verwendung des Steuerzeichens `\n` – anstelle des Manipulators `endl` – einem guten C++-Stil nicht entgegen.

## 9.5 Weitere Steuerzeichen

Neben dem Steuerzeichen `\n` gibt es eine Reihe weiterer Escape-Sequenzen für diejenigen der ersten 32 Zeichen der ASCII-Codetabelle, die häufig in C++-Programmen benötigt werden. Da diese Zeichen nicht druckbar sind, könnten sie auf andere Weise nicht im Code dargestellt werden – sieht man von der Möglichkeit, die Zeichen als Escape-Sequenz mit hexadezimalen Zahlen zu notieren, einmal ab (Abschnitt 9.3 »Darstellung von Zeichen mittels Escape-Sequenzen«).

Tabelle 9.4 enthält eine Auflistung der gebräuchlichsten Escape-Sequenzen.

Escape-Sequenz	Zeichen
<code>\n</code>	Zeilenvorschub (LF)
<code>\t</code>	horizontaler Tabulator (HT)
<code>\a</code>	Signalton (BEL)
<code>\b</code>	Rückwärtsschritt (BS)
<code>\r</code>	Wagenrücklauf (CR)
<code>\f</code>	Seitenvorschub (FF)
<code>\v</code>	vertikaler Tabulator (VT)
<code>\0</code>	Nullzeichen (NUL)
<code>\"</code>	"
<code>\'</code>	'
<code>\\</code>	\

Tabelle 9.4: C++-Escape-Sequenzen

Das Nullzeichen oder Nullbyte dient als Begrenzungszeichen für Zeichenketten. Wie gesagt, werden wir darauf in Kapitel 19 »Strings« näher eingehen. Die Zeichen \f (Seitenvorschub) und \v (vertikaler Tabulator) sind in der Tabelle eigentlich nur der Vollständigkeit halber erwähnt, in der Praxis sind die Zeichen nicht so bedeutend. Das Zeichen \f steuert allein die Druck-, nicht die Bildschirmausgabe; \v ist praktisch nicht mehr als vertikaler Tabulatorschritt verwendbar. Stattdessen erscheint in der Regel das Symbol \_ auf dem Bildschirm.

Das Zeichen \t bewirkt in der Ausgabe einen waagerechten Tabulatorsprung.

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

int main(void)
{
    cout << "C\t+\t+";
    cout << endl;
    return 0;
}
```

Ausgabe:

```
C      +      +
```

Wie viele Zeichen der Cursor bei einem Tabulatorschritt weitergesetzt wird, hängt von den Einstellungen des Computers ab, auf dem das Programm ausgeführt wird.

Signaltöne lassen sich mit dem Zeichen \a erzeugen.

### **Hinweis**

Signalton – andere nennen ihn Klingelzeichen, Klingelton, sogar Alarm(ton) oder schlicht Piepton. Gemessen an der Klangstärke kommt letztere Bezeichnung der Sache wohl am nächsten.

Das folgende Programm gibt jeweils nach Ausgabe eines Buchstabens einen Signalton wieder:

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

int main(void)
{
    cout << "F\aE\aH\aL\aE\aR\a";
    cout << endl;
    return 0;
}
```

Oder zur Verdeutlichung das Ganze in einer etwas umständlicheren Form:

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

int main(void)
{
    cout << 'F';
    cout << '\a';
    cout << 'E';
    cout << '\a';
    cout << 'H';
    cout << '\a';
    cout << 'L';
    cout << '\a';
    cout << 'E';
    cout << '\a';
    cout << 'R';
    cout << '\a';
    cout << endl;
    return 0;
}
```

Das Zeichen `\b` setzt den Cursor um eine Stelle zurück.

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

int main(void)
{
    cout << "C\b++";
    cout << endl;
    return 0;
}
```

Ausgabe ist nicht C++, sondern

```
++
```

da nach Ausgabe des ersten Zeichens (C) der Cursor mit `\b` zurückgesetzt und das C anschließend mit dem ersten + überschrieben wird.

Analog funktioniert `\r`, nur dass hier der Cursor an den Anfang der aktuellen Zeile gesetzt wird. Das Programm

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

int main(void)
{
    cout << "Hal\rlo";
    cout << endl;
    return 0;
}
```

erzeugt somit die Ausgabe

```
lol
```



# 10

# Variablen

Wie Sie es einrichten, dass Ihre C++-Programme Informationen an den Benutzer weitergeben, wissen Sie bereits. Jede Applikation beruht aber im Wesentlichen auf einem wechselseitigen Datenaustausch zwischen Anwender und Programm.

So müssen z.B. im Zuge von mathematischen Operationen gewöhnlich erst Daten vom Benutzer entgegengenommen werden, um ihm anschließend die gewünschten Ergebnisse auszugeben. In der Regel ist es notwendig, die Eingabedaten über einen gewissen Zeitraum hinweg festzuhalten, um gegebenenfalls mehrmals darauf zugreifen zu können. Dies gilt mitunter auch für Informationen, die vom Programm selbst erzeugt werden – denken Sie an eventuelle Zwischenergebnisse von Berechnungen.

C++ stellt hierzu das Konzept der Variablen zur Verfügung. Variablen sind Datenobjekte, die es erlauben, Informationen zu speichern, zu verändern und im Code auf diese gespeicherten Daten zuzugreifen. Das vorliegende Kapitel beschreibt den Umgang mit Variablen.

## 10.1 Variablen deklarieren

Bevor Sie eine Variable verwenden können, müssen Sie diese zunächst anlegen. Man spricht in diesem Zusammenhang vom Deklarieren einer Variablen bzw. von deren Vereinbarung. Um eine Variable zu deklarieren, geben Sie zunächst ihren Datentyp an, gefolgt von einem frei wählbaren Namen für die Variable. Und da es sich bei einer Variablendeklaration schließlich auch um eine Anweisung handelt, wird das Ganze mit einem Semikolon abgeschlossen.

```
char zeichen;
```

Mit obiger Anweisung wird demnach eine Variable namens `zeichen` zur Aufnahme von einzelnen Zeichen vereinbart. Wie Sie ja bereits erfahren haben, lautet der dafür zuständige Datentyp `char`. Der programmiersprachliche Fachausdruck für frei wählbare Namen ist *Bezeichner*. Bei `zeichen` handelt es sich also um den Bezeichner für eine Variable vom Typ `char`.

### Hinweis

Allerdings ist die grundsätzliche Freiheit in der Bezeichnerwahl gewissen Einschränkungen unterworfen (dazu mehr in Abschnitt 10.2 »Regeln zur Bezeichnerwahl«).

Was ist nun eigentlich eine Variable bzw. was geschieht im Zuge einer Variablendeklaration? Stellen Sie sich dazu den Arbeitsspeicher Ihres Computers eingeteilt in Speicherzellen vor, von denen jede exakt 1 Byte groß ist. Zur Identifizierung besitzen diese Speicherzellen fortlaufende ganzzahlige Adressen.

Im Grunde ist eine Variable nichts anderes als ein benannter Ort im Arbeitsspeicher. Benannt, weil der für die Variable reservierte Speicherbereich im weiteren Code mit Namen, nämlich mit dem bei der Deklaration gewählten Bezeichner der Variablen, ansprechbar ist.

Die Deklarationsanweisung `char zeichen;` hat bei Ausführung des Programms also zur Folge, dass

- im Arbeitsspeicher ein entsprechend dem angegebenen Datentyp großer Bereich reserviert wird, der ab sofort nur mehr von diesem Programm nutzbar ist. Werte des Typs `char` beanspruchen genau 1 Byte. Das ist der Platz, der zur internen Darstellung eines Zeichens benötigt wird.
- auf diesen Speicherort unter Angabe des Namens `zeichen` zugegriffen werden kann. Das heißt, es können dort Werte abgelegt bzw. vom Programm gelesen werden. Wie das geht, erfahren Sie in den Abschnitten 10.3 »Zuweisung« und 10.4 »Das Objekt cin«).

Zum Vergleich sei hier ein weiterer Datentyp genannt: der Datentyp `int`. Dieser dient zur Darstellung von ganzen Zahlen. Um eine Variable dieses Typs zu deklarieren, schreiben Sie

```
int zahl;
```

wobei `zahl` wiederum der Bezeichner der Variablen ist. Ebenso hätten wir als Bezeichner `nummer` bzw. das englische `number` oder `a`, `b`, `xy` usw. wählen können. Allerdings ist es im Allgemeinen sinnvoll, für Datenobjekte »sprechende« Namen zu vergeben.

Das Schlüsselwort `int` gibt den Datentyp der Variablen an. In der Regel beansprucht eine `int`-Variable 4 Byte.

### Hinweis

Diese Größe ist allerdings systemabhängig (siehe dazu Kapitel 14 »Datentypen«), auf 16-Bit-Computern werden dafür nur 2 Byte reserviert – entsprechend kleiner ist dann auch der Wertebereich.

### Hinweis

Wie Sie in Kapitel 14 sehen werden, gibt es zur Darstellung von ganzen Zahlen mehrere Datentypen. Diese unterscheiden sich ausschließlich in ihrem Wertebereich voneinander. Der Sammelbegriff für Datentypen zur Darstellung von ganzen Zahlen ist *Integer*. Dieses Wort wird in der Fachsprache häufig gebraucht. Man spricht vom *Integer*-datentyp bzw. von *Integer*-variablen. Die obige Variable `zahl` ist vom Typ `int` bzw. sie ist vom Typ *Integer* – beide Ausdrucksweisen sind demnach korrekt.



### 10.1.1 Mehrere Variablen mit einer Anweisung deklarieren

Um zwei oder mehr Variablen zu deklarieren, schreiben Sie also:

```
int zahl;  
char zeichen;  
...
```

#### Hinweis

Wie gesagt, bezeichnen Variablennamen bestimmte Orte im Arbeitsspeicher. Dies muss natürlich in eindeutiger Weise geschehen. Daraus folgt, dass sich die Bezeichner der einzelnen Variablen voneinander unterscheiden müssen. Dies gilt auch, insofern es sich um verschiedene Datentypen handelt. Die zweite Anweisung des folgenden Codestücks ist daher fehlerhaft, da der Bezeichner `donald` mit der ersten Deklaration bereits vergeben ist.

```
int donald;  
char donald; // FEHLER
```

Da C++ zwischen Groß- und Kleinschreibung unterscheidet, ließe sich der Fehler zwar wie folgt beheben, da sich ein Bezeichner `donald` gegenüber einem großgeschriebenen `Donald` ausreichend unterscheidet:

```
int donald;  
char Donald;
```

Im Sinne eines gut lesbaren Quellcodes ist jedoch davon abzuraten. Zudem ist es unter C++-Programmierern im Allgemeinen üblich, Bezeichner von lokalen Variablen (um solche handelt es sich im Weiteren) kleinzuschreiben, um sie im Code von anderen Datenobjekten besser unterscheiden zu können.

Soweit es sich um Variablen gleichen Datentyps handelt, lassen sich diese auch in einer einzigen Anweisung vereinbaren. Dazu sind die Bezeichner der einzelnen Variablen durch Kommata (,) zu trennen:

```
int zahl, nummer, nochEineZahl;
```

Mit obiger Anweisung werden demnach drei Integervariablen vereinbart. Das Schlüsselwort für den Datentyp (hier: `int`) ist dabei nur ein Mal anzugeben.

#### Hinweis

Die Datentypen heißen `int`, `char` usw. Die Schlüsselwörter, mit denen sich angegeben lässt, welcher Datentyp verwendet werden soll, haben den gleichen Namen wie die Datentypen selbst, also im vorliegenden Fall `int` und `char`.

## 10.2 Regeln zur Bezeichnerwahl

Bezeichner sind eindeutige Namen für Datenobjekte, Funktionen, Klassen, Methoden usw. Sie sind, wie gesagt, vom Programmierer grundsätzlich frei wählbar. Einschränkungen bestehen jedoch in Bezug auf folgende Regeln:

- Bezeichner dürfen sich aus Buchstaben, Ziffern und – als einzigem Sonderzeichen – dem Unterstrich ( `_` ) zusammensetzen. Umlaute, das »ß«, andere fremdsprachliche Zeichen und Leerzeichen sind nicht erlaubt.
- Das erste Zeichen darf keine Ziffer sein. Obwohl gestattet, sollten Sie keine Bezeichner mit einem führenden Unterstrich verwenden (wie `_meinevar`), da diese Schreibweise für vordefinierte Konstanten der C++-Entwicklungsumgebung vorgesehen ist. Unterstriche innerhalb des Namens sind unproblematisch und werden gerne als Ersatz für Leerzeichen verwendet. Bezeichner wie `meine_variable`, `my_var` und `um_2007` sind durchaus üblich.
- C++ ist »case-sensitiv« – es unterscheidet zwischen Groß- und Kleinschreibung. Das bedeutet, dass ein Bezeichner `xyz` auch im weiteren Quellcode so geschrieben werden muss und nicht etwa als `Xyz`, `XYZ`, `xYz` usw.
- Vom C++-Standard her gibt es bezüglich der Länge eines Bezeichners keine Vorschriften. Restriktionen bestehen allein in Bezug auf den verwendeten Compiler. Darüber müssen Sie sich aber keine Gedanken machen, da nahezu jeder Compiler weit über 200 Zeichen erlaubt.
- Auch Schlüsselwörter sind letzten Endes nichts anderes als vordefinierte Bezeichner. Daher dürfen Ihre eigenen Bezeichner nicht mit einem Schlüsselwort übereinstimmen, da ansonsten die Eindeutigkeit verloren geht.

In Tabelle 10.1 finden Sie eine Auflistung aller C++-Schlüsselwörter. Einige davon dürfen Ihnen bereits bekannt sein.

Schlüsselwörter			
<code>asm</code>	<code>auto</code>	<code>bad_cast</code>	<code>bad_typeid</code>
<code>bool</code>	<code>break</code>	<code>case</code>	<code>catch</code>
<code>char</code>	<code>class</code>	<code>const</code>	<code>const_cast</code>
<code>continue</code>	<code>default</code>	<code>delete</code>	<code>do</code>
<code>double</code>	<code>dynamic_cast</code>	<code>else</code>	<code>enum</code>
<code>except</code>	<code>explicit</code>	<code>extern</code>	<code>false</code>
<code>finally</code>	<code>float</code>	<code>for</code>	<code>friend</code>
<code>goto</code>	<code>if</code>	<code>inline</code>	<code>int</code>
<code>long</code>	<code>mutable</code>	<code>namespace</code>	<code>new</code>
<code>operator</code>	<code>private</code>	<code>protected</code>	<code>public</code>
<code>register</code>	<code>reinterpret_cast</code>	<code>return</code>	<code>short</code>

Tabelle 10.1: Vordefinierte C++-Schlüsselwörter

Schlüsselwörter			
signed	sizeof	static	static_cast
struct	switch	template	this
throw	true	try	type_info
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	while		

Tabelle 10.1: Vordefinierte C++-Schlüsselwörter (Forts.)

Eine Variablendeklaration wie folgt ist also nicht möglich, da `break` als Schlüsselwort bereits vergeben ist:

```
char break; // FEHLER
```

Weitere Beispiele für ungültige Bezeichner sind:

```
2zahl // keine Ziffer als erstes Zeichen
z-1 // Bindestrich nicht erlaubt
über // keine Umlaute
donald duck // Leerzeichen nicht erlaubt
```

Hier ein paar Beispiele für gültige Bezeichner:

```
ueber
zahl1
zahl2
zahl_3
a
a123
a_1_30
donaldDuck
donald_duck
```

Wie gesagt, sei Ihnen ans Herz gelegt, bei der Bezeichnerwahl möglichst sprechende Namen vorzuziehen – von Ausnahmen, die auch wir vereinzelt in den Beispielen dieses Buches für uns in Anspruch nehmen werden, einmal abgesehen. Ein Quelltext mit Variablennamen wie `anzahl`, `summe` und `alter` lässt sich in der Regel viel leichter nachvollziehen als derselbe Code mit den Bezeichnern `a`, `b` und `c`.

### Hinweis

Falls Variablennamen aus mehreren Wörtern bestehen, ist es üblich, diese mit einem Unterstrich zu verbinden (`donald_duck`, `wer_hat_was`) oder sich mit Großbuchstaben zu behelfen (`donaldDuck`, `werHatWas`).

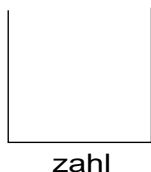
## 10.3 Zuweisung

Legen wir nun eine Integervariable an:

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

int main(void)
{
    int zahl;
    return 0;
}
```

Wie Sie ja nun wissen, bezeichnet `zahl` mithin einen Speicherort, an dem Werte abgelegt werden können. Für einen Programmierer ist es durchaus sinnvoll, sich eine Variable als Behälter vorzustellen. Dieses Bild kommt – vom praktischen Standpunkt aus betrachtet – der Wirklichkeit hinreichend nahe.



**Abbildung 10.1:** Variable ohne Inhalt

Der Behälter im Bild ist leer, weil unsere Variable `zahl` ja noch keinen Wert enthält.

### Hinweis

Genau genommen besitzt sie einen undefinierten Zufallswert und schon jetzt der Hinweis: Sie sollten tunlichst vermeiden, auf eine Variable lesend zuzugreifen, bevor diese einen wohldefinierten Wert besitzt. Ansonsten würde dies im günstigsten Fall »nur« zu falschen Berechnungsergebnissen führen. Denn der Speicherort, der im Zuge der Variablendeklaration für zukünftige Werte reserviert wird, erfährt ja vorerst keine Veränderung. Sind dort bereits Daten vorhanden, werden diese zunächst nicht gelöscht. Dies geschieht erst, wenn während des Programmlaufs neue Werte in die Variable – das heißt an diese Stelle – geschrieben werden.

In diesem Zusammenhang unterscheiden sich die Programmiersprachen. Die einen verbieten generell einen Zugriff auf Variablen, die keinen wohldefinierten Wert besitzen (Syntaxfehler). In anderen Sprachen werden numerische Variablen vom Compiler bei der Deklaration automatisch mit einem Anfangswert wie 0 versehen. Für C++ trifft weder das eine noch das andere zu<sup>1</sup>, was hundertprozentig der Philoso-

- 1 Ob eine Variable vom C++-Compiler im Zuge der Deklaration einen definierten Wert zugewiesen bekommt, hängt vom Ort der Deklaration ab. Variablen, die in Funktionen definiert werden (lokale Variablen), bekommen keinen Wert zugewiesen. Globale Variablen, die außerhalb von Funktionen (oder Klassen) deklariert werden, bekommen den Nullwert ihres Datentyps zugewiesen.

### Hinweis (Forts.)

phie von C++ entspricht. Einerseits muss der C++-Programmierer alles »zu Fuß« machen, andererseits darf er auch mehr machen als in anderen Sprachen.

Also werden wir der Variablen jetzt einen Wert zukommen lassen, sagen wir den Wert 99. Grundsätzlich stehen uns dazu zwei verschiedene Wege offen: zum einen durch Benutzereingabe (dazu mehr in Abschnitt 10.4 »Das Objekt cin«) und zum anderen durch Zuweisung. Dabei handelt es sich um eine Operation, die im Code durch das Symbol = dargestellt wird. Dieses Zeichen heißt in der Welt von C++ nicht etwa Gleichheitszeichen, sondern – nach der Operation, die es repräsentiert – *Zuweisungsoperator*.

Es gilt:

- Links vom Zuweisungsoperator steht immer der Name einer Variablen. Damit wird die Variable bezeichnet, die etwas bekommen soll (einen neuen Wert).
- Rechts vom Zuweisungsoperator steht der Wert, den diese Variable erhalten soll.

Um unserer Variablen `zahl` den Wert 99 zuzuweisen, müssen wir also schreiben:

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

int main(void)
{
    int zahl;
    zahl = 99;
    return 0;
}
```

Dass die Zuweisung erst nach der Deklaration stattfinden kann, versteht sich nach dem weiter oben Gesagten von selbst. Eine Variable muss ja erst einmal existent sein, bevor man mit ihr arbeiten kann.

### Hinweis

Sie erinnern sich? Konstante Werte vom Typ `char` werden durch einfache Anführungszeichen (`'h'`), konstante Werte vom Typ `string` durch doppelte begrenzt (`"Guten Tag"`). Mit konstanten Werten vom Typ `int` verhält es sich einfacher. Sie werden in gewohnter Weise in den Quellcode geschrieben, also z.B. 1, 2, 0, 2008 oder eben 99. Negativen Zahlen stellen Sie ebenfalls wie üblich ein Minuszeichen voran: -11, -99, -5 usw.

Dies ist auch der Grund, warum das erste Zeichen eines Bezeichners keine Ziffer sein darf (siehe Abschnitt 10.2 »Regeln zur Bezeichnerwahl«). Denn der Compiler könnte dann bei einem Ausdruck wie `123` nicht mehr feststellen, ob es sich um eine Variable dieses Namens (die vermutlich einen ganz anderen Wert eines möglicherweise anderen Datentyps als `int` beherbergt) oder um den konstanten Integerwert 123 handelt.

Mit besagter Zuweisung ergibt sich für unsere Variable nunmehr folgendes Bild:



Abbildung 10.2: Nach der Zuweisung

Beachten Sie:

- Welche Art von Werten eine Variable aufnehmen kann, ist durch ihren Datentyp vorgegeben.
- Eine Variable kann ihren Datentyp nicht ändern. Dieser wird im Zuge ihrer Deklaration ein für alle Mal festgelegt.

Im vorliegenden Fall würde der Versuch, der Variablen `zahl` etwa eine Zeichenkette zuzuweisen, als Syntaxfehler interpretiert. Es handelt sich ja um eine Integervariable, die grundsätzlich nur ganze Zahlen akzeptiert.

### Hinweis

In bestimmten Fällen führt der Compiler allerdings automatische Typumwandlungen durch. Das heißt, ein auf der rechten Seite der Zuweisung angegebener Wert, dessen Datentyp nicht mit dem der Zielvariablen übereinstimmt, wird vor der eigentlichen Zuweisung zunächst in den Datentyp der Zielvariablen konvertiert (Abschnitt 15.1 »Implizite Typumwandlungen«). Im Allgemeinen sollte man sich aber nicht auf implizite Typumwandlungen verlassen.

Um eine Variable nach deren Deklaration zu verwenden, genügt es, ihren Bezeichner im Code anzugeben, wie wir das bei der Zuweisung bereits getan haben. Sie können also den Erfolg der Zuweisung leicht nachprüfen, indem Sie den Inhalt der Variablen `zahl` mit `cout` ausgeben.

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

int main(void)
{
    int zahl;
    zahl = 99;
    cout << zahl;
    return 0;
}
```

Ausgabe: 99

Beachten Sie, dass eine Variable zur gleichen Zeit nur einen einzigen Wert enthalten kann. Eine erneute Zuweisung überschreibt den alten Wert. Die ursprüngliche Information geht dabei unwiederbringlich verloren.

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

int main(void)
{
    int zahl;
    zahl = 99;
    zahl = 17;
    cout << zahl;
    return 0;
}
```

Ausgabe: 17



A diagram consisting of a rectangular box with the number '17' centered inside. Below the box, the variable name 'zahl' is written.

**Abbildung 10.3:** Nach der Neuzuweisung

Lassen Sie uns zu Demonstrationszwecken eine zweite Integervariable – nennen wir sie `nummer` – vereinbaren. Da es sich um denselben Datentyp handelt, dürfen wir beide Deklarationen in einer Anweisung zusammenfassen:

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

int main(void)
{
    int zahl, nummer;
    zahl = 99;
    zahl = 17;
    cout << zahl;
    return 0;
}
```

## Hinweis

Anders als in der Programmiersprache C dürfen Sie eine Variable grundsätzlich an jeder Stelle im Code deklarieren, solange Sie diese erst nach ihrer Erzeugung verwenden (für uns gilt allerdings bis auf weiteres die Voraussetzung, dass dies innerhalb des Funktionsblocks von `main()` geschehen muss). Wir werden jedoch unsere Variablen in der Regel zu Beginn des Funktionsblocks deklarieren, soweit keine besonderen Gründe dagegen sprechen. Damit schafft man einen separaten Deklarationsteil und kann auf einen Blick übersehen, welche Variablen im Quellcode verwendet werden.

Auch die zweite Variable `nummer` haben wir uns zunächst als leeren Behälter vorzustellen.

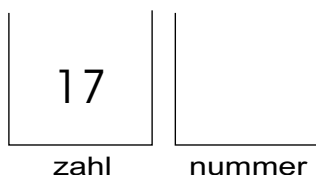


Abbildung 10.4: Zwei Variablen

Sie soll nun denselben Wert erhalten wie die Variable `zahl`. Obwohl uns der aktuelle Wert von `zahl` bekannt ist, soll dies aber nicht mit der Anweisung

```
nummer = 17;
```

geschehen. Vielmehr wollen wir so tun, als ob wir den Inhalt von `zahl` nicht kennen würden. Nehmen wir einfach an, `zahl` hätte ihren aktuellen Wert zuvor mittels einer Benutzereingabe erhalten (wie das geht, erfahren Sie weiter unten in Abschnitt 10.4 »Das Objekt cin«).

In diesem Fall verwenden wir den Bezeichner `zahl` einfach auf der rechten Seite der Zuweisung. Im Anschluss daran geben wir zur Kontrolle beide Variablenwerte – mit erklärendem Text – aus.

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

int main(void)
{
    int zahl, nummer;
    zahl = 99;
    zahl = 17;
    nummer = zahl;
    cout << "Wert von zahl: " << zahl << endl;
    cout << "Wert von nummer: " << nummer;
    return 0;
}
```



Ausgabe:

```
Wert von zahl: 17
```

```
Wert von nummer: 17
```

Aufmerksamkeit verdient die Tatsache, dass bei Ausführung der Anweisung

```
nummer = zahl;
```

zunächst die Variable `zahl` gelesen und danach die eigentliche Zuweisung des mit der Leseoperation erhaltenen Wertes an die Variable `nummer` stattfindet. Die Abarbeitung einer Zuweisung erfolgt also von rechts nach links.

### Hinweis

Die Anweisungen

```
cout << "Wert von zahl: " << zahl << endl;
```

und

```
cout << "Wert von nummer: " << nummer;
```

sollten Ihnen keine größeren Schwierigkeiten bereiten. Beispielsweise wird in der ersten Anweisung zunächst eine Zeichenkette, dann ein `int`-Wert, repräsentiert durch die Variable `zahl`, und schließlich mit `endl` ein Zeilenvorschub ausgegeben. Beachten Sie, dass jede Variable wie auch jede Konstante und auch jeder Manipulator (`endl`) eine eigene Ausgabeinheit erfordert. Dies gilt auch, wenn Sie in einer Anweisung mehrere Variablenwerte gleichen Datentyps nacheinander ausgeben wollen:

```
cout << zahl << nummer;
```

Hier unser neues Speicherbild:

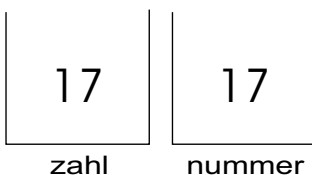


Abbildung 10.5: Nach der Zuweisung `nummer = zahl;`

### 10.3.1 Additionsoperator (+)

Wir stellten weiter oben fest, dass rechts vom Zuweisungsoperator der Wert steht, den die auf der linken Seite bezeichnete Variable erhalten soll. Das stimmt nicht ganz, zumindest bedarf diese Aussage einer gewissen Ergänzung, da auf der rechten Seite einer

Zuweisung auch ein komplexer Ausdruck stehen darf. Dann muss dieser Ausdruck erst ausgewertet werden, bevor die eigentliche Zuweisung des Ergebniswertes an die Zielvariable erfolgen kann.

### Hinweis

Die genannten Aktionen (Auswertung eines Ausdrucks und Zuweisung eines Wertes an eine Variable) geschehen natürlich unbemerkt vom Benutzer, wenn die entsprechende Anweisung beim Programmablauf zur Ausführung gelangt.

Zur Demonstration wollen wir der Variablen `zahl` die Summe ihres aktuellen Wertes plus den Wert von `nummer` zuweisen. Dabei wollen wir wiederum von einer Situation ausgehen, in der die Werte beider Variablen bei der Programmentwicklung nicht bekannt sind. Eine Zuweisung wie

```
zahl = nummer;
```

führt nicht zum gewünschten Erfolg, da der bis dato gespeicherte Wert der Variablen `zahl` dabei überschrieben wird. Folglich muss der Wert von `zahl` vor der eigentlichen Zuweisung zunächst gelesen werden, das heißt, auf der rechten Seite der Zuweisung muss die Summe aus `zahl` und `nummer` stehen. Das Symbol für die mathematische Addition ist das Pluszeichen (+). Demnach lautet die passende Anweisung:

```
zahl = zahl + nummer;
```

Bemerkenswert ist hier, dass der Bezeichner `zahl` sowohl auf der linken als auch auf der rechten Seite der Zuweisung auftaucht.

Es gilt:

- Die links vom Zuweisungsoperator befindliche Variable wird manipuliert.
- Erscheint ein Variablenbezeichner auf der rechten Seite einer Zuweisung, so bedeutet dies, dass die Variable gelesen, aber nicht verändert wird.

### Hinweis

Das heißt, es wird eine interne Kopie der Variablen erzeugt, um mit dem gelesenen Wert weiterzuarbeiten. Die Variable selbst bleibt dabei unberührt.

In unserem Fall wird die Variable `zahl` sowohl gelesen als auch verändert, da sie auf beiden Seiten der Zuweisung auftaucht.

Machen wir die Probe aufs Exempel:

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

int main(void)
{
    int zahl, nummer;
    zahl = 99;
    zahl = 17;
    nummer = zahl;
    zahl = zahl + nummer;
    cout << "Wert von zahl: " << zahl << endl;
    cout << "Wert von nummer: " << nummer;
    return 0;
}
```

Ausgabe:

```
Wert von zahl: 34
Wert von nummer: 17
```

Die Ausgabe entspricht den Erwartungen. Die Variable `zahl` wurde um den Wert von `nummer` erhöht. Die Variable `nummer` blieb unverändert.

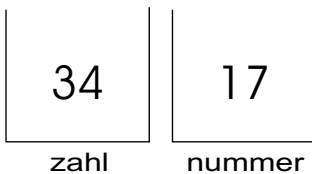


Abbildung 10.6: Nach der Zuweisung `zahl = zahl + nummer;`

Halten Sie sich bezüglich der Anweisung `zahl = zahl + nummer` die Reihenfolge des Geschehens vor Augen (hier wird besonders gut deutlich, dass die Abarbeitung einer Zuweisung von rechts nach links erfolgt).

- Zunächst werden die Werte der rechts vom Zuweisungsoperator genannten Variablen `zahl` und `nummer` bestimmt.
- Erst dann kann – mit den Werten – die Additionsoperation durchgeführt werden.
- Im letzten Schritt erfolgt die eigentliche Zuweisung des Ergebniswertes an die Variable `zahl`.

### 10.3.2 Initialisierung

Es ist möglich, einer Variablen bereits im Zuge ihrer Deklaration einen Anfangswert zuzuweisen. Man spricht in diesem Fall von der »Initialisierung« (der entsprechenden Variablen).

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

int main(void)
{
    int zahl = 99;
    cout << zahl;
    return 0;
}
```

Ausgabe: 99

Falls in einer Anweisung mehrere Variablen vereinbart sind, steht es Ihnen frei, alle bzw. nur einige davon zu initialisieren:

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

int main(void)
{
    int zahl1 = 99, zahl2, zahl3 = 1001;
    // ...
    return 0;
}
```

Die Variablen zahl1 und zahl3 sind mit den Werten 99 bzw. 1001 initialisiert, zahl2 bleibt zunächst ohne definierten Wert.

#### Hinweis

Allgemein bedeutet Initialisieren, ein Datenobjekt mit einem Anfangswert zu versehen. Im weiteren Sinne spricht man also auch dann vom Initialisieren einer Variablen, wenn dies erst unmittelbar nach ihrer Vereinbarung geschieht.

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

int main(void)
{
    int eineZahl;
    eineZahl = 44;
}
```

**Hinweis (Forts.)**

```
// ...  
return 0;  
}
```

In Bezug auf obiges Listing ist es demnach korrekt festzustellen, der Initialisierungswert der Variablen `eineZahl` sei 44 – auch wenn die Variable diesen Wert nicht mit der Deklarationsanweisung erhalten hat.

### 10.3.3 string-Variablen

Obwohl es sich bei dem Typ `string`, wie bereits erwähnt, nicht um einen elementaren Datentyp handelt, unterscheiden sich `string`-Variablen im Gebrauch nicht wesentlich von den Variablen elementarer Datentypen. Um mit `string`-Variablen arbeiten zu können, ist es allerdings erforderlich, die Headerdatei `string` mit einer entsprechenden `#include`-Direktive einzubinden:

```
#include <iostream>  
#include <string>  
using namespace std;  
  
int main(void)  
{  
    string myString = "C++";  
    cout << myString;  
    return 0;  
}
```

Ausgabe: C++

Auch für den Datentyp `string` ist eine Operation definiert, die durch das Pluszeichen (+) symbolisiert wird. Sie verkettet zwei Strings miteinander.

**Hinweis**

Als Strings (großgeschrieben) werden außer Zeichenketten auch Variablen vom Datentyp `string` bezeichnet.

```
#include <iostream>  
#include <string>  
using namespace std;  
  
int main(void)  
{  
    string myString = "C++";
```

```

string yourString = "Die Programmiersprache ";
string herString = yourString + myString;
cout << herString;
return 0;
}

```

Ausgabe:

Die Programmiersprache C++

### Hinweis

Das Aneinanderhängen bzw. Verketteten von Strings wird auch *Konkatenation* genannt.

Im obigen Listing wird das Ergebnis der Konkatenation `yourString + myString` zunächst in der Variablen `herString` gespeichert, bevor diese dann mit `cout` ausgegeben wird. Natürlich ist es auch möglich, den Ausdruck `yourString + myString` sofort an `cout` zu übergeben:

```

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

int main(void)
{
    string myString = "C++";
    string yourString = "Die Programmiersprache ";
    cout << yourString + myString;
    return 0;
}

```

Ausgabe:

Die Programmiersprache C++

Das Verketteten von Strings mit dem Plusoperator (+) funktioniert nur unter der Voraussetzung, dass mindestens ein Operand eine `string`-Variable ist. Allerdings ist der Einsatz des Plusoperators (+) auch nur in diesem Fall notwendig, wie Sie gleich sehen werden:

```

cout << yourString + "C++"; // OK,
    // yourString ist eine string-Variable
cout << "Die Programmiersprache " + "C++"; //FEHLER,
    // keine string-Variable

```

Richtig ist dagegen:

```
cout << "Die Programmiersprache " "C++"; // OK
```

In der letzten Anweisung findet die Verkettung nicht mit dem Plusoperator (+) statt. Die bloße Aneinanderreihung zweier Zeichenketten bewirkt, dass diese vom Compiler wie eine einzige behandelt werden (Kapitel 8, Abschnitt 8.3.3 »Verkettung von Strings«).

### Achtung

Die Verkettung zweier Strings ohne den Plusoperator (+) ist jedoch nur möglich, wenn es sich ausschließlich um konstante string-Werte – also Zeichenketten – handelt. Folgende Anweisung ist daher fehlerhaft.

```
cout << yourString "C++"; // FEHLER
```

Hier ist es zwingend notwendig, beide Strings (die Zeichenkette und die string-Variable) wie oben gezeigt mit dem Konkatenationsoperator (+) zu verbinden.

## 10.4 Das Objekt cin

Das Gegenstück zu cout ist cin. Dieses steht für den Datenfluss in der entgegengesetzten Richtung: vom Benutzer zum Programm. Das Objekt cin repräsentiert dabei das Standardeingabegerät, was in der Regel die Tastatur ist. Dementsprechend weist der Eingabeoperator (>>) mit der Spitze von cin – der Tastatur – weg.

```
cin >> ...;
```

Obige Anweisung ist allerdings noch unvollständig. Nach dem Eingabeoperator muss der Bezeichner einer Variablen stehen. Es muss ja festgelegt sein, wohin die Benutzereingabe gespeichert werden soll.

Das folgende Programm nimmt vom Benutzer die Eingabe eines Zeichens entgegen und speichert dieses in der char-Variablen zeich:

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

int main(void)
{
    char zeich;
    cin >> zeich;
    return 0;
}
```


Zur Kontrolle soll der Inhalt von `zeich` auf den Bildschirm ausgegeben werden. Zudem empfiehlt es sich, dem Benutzer mitzuteilen, was er zu tun hat.

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


int main(void)
{
    char zeich;
    cout << "Geben Sie bitte ein Zeichen ein: ";
    cin >> zeich;
    cout << "Sie haben das Zeichen " << zeich
        << " eingegeben." << endl;
    return 0;
}
```

Wenn Sie nach der Aufforderung das Zeichen `Y` eintippen, präsentiert sich die Ausgabe wie folgt:

```
Geben Sie bitte ein Zeichen ein: Y
Sie haben das Zeichen Y eingegeben.
```

Beachten Sie, dass `cin` nach der Eingabe auf  wartet, wenn Sie das Programm ausführen. Eine weitere Besonderheit von `cin` ist, dass es den Cursor nach der Eingabe in die nächste Zeile setzt. Daher erfolgt die Ausgabe von `Sie haben das Zeichen ...` in der folgenden Zeile.

### Hinweis

Falls Sie mehrere Zeichen eingeben, bevor Sie die -Taste drücken, werden Sie feststellen, dass nur das erste Zeichen in der Variablen `zeich` landet. Das ist an und für sich nichts Besonderes, da eine `char`-Variable – im Gegensatz zur `string`-Variablen – grundsätzlich nur ein einziges Zeichen aufnehmen kann.

Interessant ist es aber zu erfahren, was mit den übrigen Zeichen passiert. Diese landen in einem Puffer und falls `cin` in diesem Fall ein weiteres Mal zur Ausführung kommt, wartet das Programm nicht auf eine Eingabe des Benutzers, sondern nimmt sich einfach das nächste Zeichen aus dem Puffer heraus (was zu tun ist, um den Eingabepuffer zu leeren, erfahren Sie in Abschnitt 19.2.1 »`cin.get()`«).

Hier ein weiteres Beispiel. Der Benutzer wird zunächst nach seinem Vornamen gefragt, anschließend wird die Benutzereingabe in einer Begrüßungsformel verwendet:

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



```
int main(void)
{
    string vname;
    cout << "Sag mir Deinen Vornamen: ";
    cin >> vname;
    cout << "Hallo " << vname << '!';
    cout << endl;
    return 0;
}
```

### Achtung

Denken Sie immer daran, dass Sie es bei cin mit einer Dateneingabe zu tun haben. Das Objekt cin steht für das Standardeingabegerät, in der Regel die Tastatur. Schreiben Sie also nicht (!) cin <<, sondern richtig cin >>, mit der Spitze von cin (der Tastatur) wegweisend. Ein – fehlerhaftes – cin << wird Ihr Compiler mit einer oder gleich mehreren Fehlermeldung(en) quittieren (zur Exaktheit von Fehlermeldungen siehe Kapitel 5, Abschnitt 5.2 »Fehlermeldungen 'aus Compilersicht'«).

Hier die entsprechende Ausgabe (Eingabe: Hans-Peter):

```
Sag mir Deinen Vornamen: Hans-Peter
Hallo Hans-Peter!
```

Beachten Sie, dass die Eingabe mit dem ersten Zwischenraumzeichen zu Ende ist. Beispielsweise wird bei Eingabe von Hans Peter nur Hans in der Variablen vname gespeichert.

```
Sag mir Deinen Vornamen: Hans Peter
Hallo Hans!
```

Die Zeichen p, e, t, e und r (Peter) landen, wie oben dargelegt, im Eingabepuffer. Zwischenraumzeichen werden bei der Eingabe grundsätzlich ignoriert. Ob z.B. Hans Peter oder Hans Peter (drei Leerzeichen) eingegeben wurde, macht von daher keinen Unterschied.

### Hinweis

Es gibt verschiedene Methoden bzw. Funktionen, die Zwischenraumzeichen bei der Eingabe berücksichtigen (Kapitel 19 »Strings«).

Auch cin erlaubt es, mehrere Eingaben in einer Anweisung hintereinander zu schalten:

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

```
int main(void)
{
    int z1, z2, z3;
    cout << "Geben Sie drei Zahlen ein. \n";
    cin >> z1 >> z2 >> z3;
    cout << "Summe: " << z1 + z2 + z3;
    return 0;
}
```

Bei der Eingabe ist zu beachten:

- Jeweils zwei Eingabewerte müssen mit einem oder mehreren Zwischenraumzeichen (␣, ␣␣ oder Leertaste) getrennt werden. Es ist also nicht notwendig, jede Eingabe mit ↵ abzuschließen.
- Die gesamte Eingabe muss aber in jedem Fall mit ↵ beendet werden.
- Jedes ↵ setzt den Cursor in die nächste Zeile.

Ausgabe (Eingabe 37, 15 und 48):

```
Geben Sie drei Zahlen ein.
37  15  48
Summe: 100
```

### Hinweis

Dass vom Benutzer auch wirklich ganzzahlige Werte eingegeben werden, setzen wir hier – wie in den weiteren Beispielen dieses Buches – großzügig voraus. Später werden Sie Möglichkeiten kennen lernen, Eingaben auf ihre Korrektheit zu überprüfen.

## 10.5 Dreieckstausch

Zum Abschluss sei hier noch ein Problem genannt, das Ihnen in der einen oder anderen Form immer wieder begegnen wird.

Es seien *a* und *b* zwei Integervariablen, die wohldefinierte Werte besitzen. Nehmen wir für *a* den Wert 11 und für *b* den Wert 3 an:

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

int main(void)
{
    int a = 11, b = 3;
    return 0;
}
```

Die Werte der Variablen sollen nun miteinander vertauscht werden.

### Hinweis

Dabei soll natürlich wieder angenommen werden, dass die aktuellen Variablenwerte nicht bekannt sind. Sonst könnten wir ja einfach schreiben

```
b = 11;  
a = 3;
```

und damit wäre der Tausch vollzogen. Ganz so einfach wollen wir es uns aber nicht machen, zumal die Werte in der Regel nicht bekannt sind, wenn dieses Problem in der Praxis auftaucht. Wenn wir in einem fortgeschritteneren Stadium daran gehen, eine Sortierfunktion für Arrays selbst zu schreiben, dann werden wir dabei auf das hier vorgestellte Verfahren zurückgreifen (siehe hierzu Kapitel 21, Abschnitt 21.2 »Funktion `sortiereArr()`«, zu Arrays siehe Kapitel 18 »Arrays«).

Die Variable `a` soll also den Wert der Variablen `b` erhalten und umgekehrt. Nun könnte man auf den Gedanken kommen, diesen der Variablen `a` einfach zuzuweisen – was ja letzten Endes auch geschehen muss, nur eben nicht sofort, wie wir gleich sehen werden:

```
#include <iostream>  
using namespace std;  
  
int main(void)  
{  
    int a = 11, b = 3;  
    a = b;  
    return 0;  
}
```

Das Problem liegt darin, dass mit dieser Zuweisung der alte Wert von `a` verloren geht. Diesen soll aber die Variable `b` erhalten. Die Lösung besteht nun darin, dass man vor der Zuweisung (`a = b`) den alten Wert von `a` erst einmal in einer zusätzlichen Variablen – die wir sinnigerweise `temp` nennen – zwischenspeichert:

```
#include <iostream>  
using namespace std;  
  
int main(void)  
{  
    int a = 11, b = 3;  
    int temp;  
    temp = a;  
    a = b;  
    b = temp;  
    return 0;  
}
```

Zuletzt erhält b den in der Variablen temp gespeicherten Wert, der nunmehr dem Ausgangswert von a entspricht. Somit ist nach der Zuweisung

```
b = temp;
```

der Tausch vollzogen. Geben wir zur Kontrolle die Variablenwerte a und b – vor dem Tausch und nach dem Tausch – auf den Bildschirm aus:

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

int main(void)
{
    int a = 11, b = 3;
    int temp;
    cout << "Vor dem Tausch: " << endl;
    cout << "Wert von a: " << a << endl;
    cout << "Wert von b: " << b << endl;
    temp = a;
    a = b;
    b = temp;
    cout << "Nach dem Tausch: " << endl;
    cout << "Wert von a: " << a << endl;
    cout << "Wert von b: " << b << endl;
    return 0;
}
```

Die Ausgabe zeigt das erwartete Ergebnis (Abbildung 10.7).

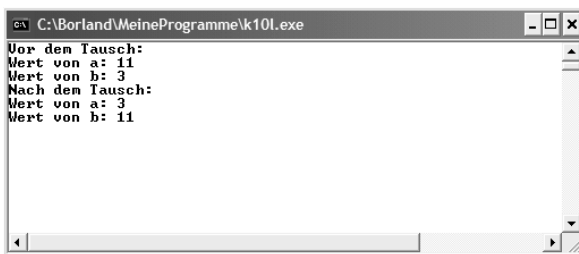


Abbildung 10.7: Werte von Variablen vertauschen

# 11

# Ausdrücke und Operatoren

Wie bereits die Beispiele in Verbindung mit dem Eingabe- und Ausgabeoperator (<< und >>), dem Zuweisungs- und dem Additionsoperator gezeigt haben, kommen den Operatoren funktionelle Bedeutung zu: Operatoren dienen der Verarbeitung bzw. Manipulation von Daten.

Operatoren basieren auf Operanden, die diese Daten repräsentieren. So gibt die Streamoperation – die über den Ausgabeoperator (<<) symbolisiert wird – den Operanden auf den Bildschirm aus, die durch das Pluszeichen repräsentierte Operation addiert beide Operanden, die Zuweisungsoperation manipuliert den linken Operanden usw.

Die meisten Operatoren arbeiten mit zwei Operanden, es gibt aber auch Operatoren, die nur einen Operanden erwarten. Allerdings bestehen Operanden oft aus komplexen Ausdrücken, von denen jeder wiederum weitere Operatoren enthalten darf usw.

## 11.1 Was ist ein Ausdruck?

Im programmiersprachlichen Sinne gilt jeder Teil einer Anweisung, der einen Wert repräsentiert, als Ausdruck. So gesehen fallen hierunter auch einzelne Variablenbezeichner und so genannte Literale.

### Hinweis

Konstante Werte wie z.B. 'g', 234, "Wie geht es Dir?", die direkt in den Quellcode geschrieben werden, bezeichnet man auch als unbenannte Konstanten oder Literale, um sie von den benannten Konstanten abzugrenzen (zu benannten Konstanten siehe Kapitel 12, Abschnitt 12.4 »Konstanten mit const deklarieren«).

Demnach kann in der Anweisung

```
cout << zahl + 23;
```

der Term `zahl + 23`, aber auch der Bezeichner `zahl` sowie das Literal `23` für sich alleine betrachtet als Ausdruck angesehen werden.

Bezeichner von Variablen können selbstverständlich nur dann als Ausdrücke angesehen werden, wenn sie im Lesekontext auftreten. Auf der linken Seite einer Zuweisung oder in einer Deklarationsanweisung repräsentieren sie keinen Wert, sondern den Speicherort selbst.

### 11.1.1 Mehrere Zuweisungen hintereinander schalten

Allerdings steht eine Zuweisung zugleich für den Wert der Variablen, an die zugewiesen wurde. Daher handelt es sich bei der ganzen Zuweisung sehr wohl um einen Ausdruck. Das wird besonders deutlich, wenn man mehrere Zuweisungen hintereinander schreibt:

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

int main(void)
{
    int z1, z2, z3;
    cin >> z1;
    z3 = z2 = z1;
    cout << z3;
    return 0;
}
```

Nach der Mehrfachzuweisung `z3 = z2 = z1` besitzen alle Variablen den gleichen Wert, der zuvor vom Benutzer eingegeben wurde. Es sei darauf hingewiesen, dass nach dem oben Gesagten die Teile `z1`, `z2 = z1`, `z2`, `z3 = z2`, `z3` und auch der ganze Term `z3 = z2 = z1` als Ausdruck zu bewerten sind.

Die eben gezeigte Art der Notation mag Ihnen etwas ungewöhnlich erscheinen und letzten Endes ist sie es auch. Schreiben Sie also die Zuweisungen ruhig einzeln in den Quellcode, wenn Ihnen das lieber ist:

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

int main(void)
{
    int z1, z2, z3;
    cin >> z1;
    z2 = z1;
    z3 = z2;
    cout << z3;
    return 0;
}
```

Oft dürfen Sie gar nicht anders verfahren, als einzelne Zuweisungen zu verwenden. Anweisungen wie die folgende sind z.B. fehlerhaft:

```
z3 = z2 + 1 = z1;
```

Der Compiler wird sich mit der folgenden oder einer ähnlichen Fehlermeldung beschweren:

```
Linker Operand muss ein L-Wert sein
```

### L-Werte/R-Werte

L-Werte sind Datenobjekte, die Platz im Speicher beanspruchen, z.B. Variablen. Nur solche können auf der linken Seite einer Zuweisung auftreten. Beispiele für R-Werte sind Literale. Diese können aus bekannten Gründen ausschließlich rechts vom Zuweisungsoperator verwendet werden. Sie repräsentieren ja stets einen Wert, nie aber einen Speicherort. Anweisungen wie

```
9 = nummer + 11; // FEHLER
```

und auch

```
z2 + 1 = z1; // FEHLER
```

bzw.

```
1 + z2 = z1; // FEHLER
```

sind daher ohne Sinn. Es gibt zwar einen Speicherort z2 (vorausgesetzt, es handelt sich um den Bezeichner einer Variablen), nicht aber einen Speicherort  $z2 + 1$  oder  $1 + z2$ .

## 11.1.2 Komplexe Ausdrücke

Komplexe Ausdrücke wie  $zahl + 23$  oder auch  $z3 = z2 = z1$  müssen während der Programmausführung zunächst ausgewertet werden, bevor mit dem Ergebniswert weitergearbeitet werden kann.

### Hinweis

Im übertragenen Sinne gilt dies auch für einfache Werte (dargestellt durch Literale oder Bezeichner von Variablen), da diese zumindest ein Mal »angefasst« bzw. gelesen werden müssen.

In diesem Zusammenhang spricht man vom Rückgabewert eines Ausdrucks. Man sagt, ein Ausdruck gibt bzw. liefert einen Wert zurück:

```
int anzahl;  
anzahl = 8 + 17;
```

So ist es durchaus üblich, bezüglich der letzten Anweisung zu sagen: »Der Ausdruck rechts vom Zuweisungsoperator gibt den Wert 25 zurück«. Ein Nicht-Programmierer würde sich natürlich ganz anders ausdrücken (»... ergibt den Wert 20«, »Das Ergebnis ...«).

**Hinweis**

Bedenken Sie, dass auch Strings im programmiersprachlichen Sinne Werte sind:

```
string myString = "Die Programmiersprache ";
cout << myString + "C++";
```

Entsprechend lässt sich bezüglich des Ausdrucks `yourString + "C++"` feststellen, er besitze den Rückgabewert "Die Programmiersprache C++".

Komplexe Ausdrücke werden von Operatoren und ihren Operanden gebildet. Dabei unterscheidet man binäre und unäre Operatoren. Binäre Operatoren, wie etwa der Additionoperator (+), erfordern zwei Operanden, unäre Operatoren dagegen nur einen. Beispiele für unäre Operatoren sind die Inkrement- und Dekrementoperatoren sowie das Minusvorzeichen. Die Inkrement- und Dekrementoperatoren werden Sie gleich kennen lernen (Abschnitt 11.4 »Inkrement- und Dekrementoperatoren«).

**Hinweis**

Im Gegensatz zum unären Minuszeichen handelt es sich beim Subtraktionsoperator – beide Zeichen werden mit dem gleichen Symbol (-) dargestellt – um einen binären Operator (zur arithmetischen Subtraktion siehe den folgenden Abschnitt).

Nun könnte man an der Einordnung des Minusvorzeichens als Operator zweifeln, da es dem Anschein nach zum Literal (im Beispiel 33) gehört:

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

int main(void)
{
    cout << -33;
    return 0;
}
```

Ausgabe: -33

Dass es sich beim Minusvorzeichen tatsächlich um einen Operator handelt, lässt sich leicht belegen, indem man zwischen dem »-«-Zeichen und der 33 ein oder mehrere Leerzeichen einfügt:

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

int main(void)
{
    cout << - 33;
    return 0;
}
```



Obiges Listing lässt sich auch in diesem Fall problemlos kompilieren und zeigt bei Ausführung die Ausgabe -33. Dies wäre nicht möglich, würde es sich bei -33 um einen einzigen Textbaustein handeln. Denn wie Sie wissen, dürfen innerhalb von Textbausteinen keine Leerzeichen eingefügt werden (Kapitel 7, Abschnitt 7.4 »Leerräume«).

**Hinweis**

Es ist dennoch üblich, einen Ausdruck wie -1 oder -927 als Literal bzw. konstanten Wert zu bezeichnen, obwohl es sich streng genommen um einen Operator (-) mit dem zugehörigen Operanden (1 oder 927) handelt.

**Hinweis**

Wie Sie vielleicht schon bemerkt haben, gehört auch der Zuweisungsoperator (=) zur Gruppe der binären Operatoren (der rechte Operand ist ein einfacher oder komplexer Ausdruck, der linke Operand die Zielvariable).

## 11.2 Arithmetische Operatoren

Zur Darstellung von arithmetischen Operationen stellt C++ eine Reihe von Symbolen zur Verfügung, die Ihnen aus der Mathematik her geläufig sein dürften.

Operation	Symbol
Addition	+
Subtraktion	-
Multiplikation	*
Division	/
Modulo	%

**Tabelle 11.1:** Arithmetische Operatoren

Den Operator zur Darstellung der mathematischen Addition kennen Sie bereits (+). Beachten Sie, dass das »+«-Zeichen zur Darstellung zweier unterschiedlicher Operationen herangezogen wird – der mathematischen Addition und der Stringkonkatenation (Kapitel 10, Abschnitt 10.3.3 »string-Variablen«). Seine Bedeutung ist also von vornherein nicht eindeutig und muss bei der Übersetzung des Quellcodes vom Compiler aus dem Kontext ermittelt werden. Entscheidend hierbei ist der Datentyp der Operanden. Handelt es sich um numerische Datentypen, repräsentiert das »+«-Zeichen die mathematische Addition. Sind die Operanden vom Datentyp string, wird die Konkatenation durchgeführt.

### Hinweis

Eigentlich fungiert das Symbol + sogar in dreifacher Bedeutung. Analog zum Minusvorzeichen existiert es auch als unäres Plusvorzeichen. Allerdings kommt es in dieser Funktion nur selten zum Einsatz, da numerische Literale vom Compiler per Default als positive Werte behandelt werden.

Der Operator für die Subtraktion (-) sollte Ihnen ebenfalls keine Probleme bereiten.

Die beiden verbleibenden Symbole zur Darstellung der Grundrechenarten:

- Der Stern (\*) dient der Multiplikation.
- Der Schrägstrich (/) repräsentiert die Division.

Folgendes Programm nimmt zwei Zahlen vom Benutzer entgegen und gibt als Ergebnis das Produkt beider Zahlen auf den Bildschirm aus:

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

int main(void)
{
    int zahl1, zahl2;
    cout << "1. Zahl: ";
    cin >> zahl1;
    cout << "2. Zahl: ";
    cin >> zahl2;
    cout << "Ergebnis: " << zahl1 * zahl2;
    return 0;
}
```

Ausgabe:

```
1. Zahl: 13
2. Zahl: 21
Ergebnis: 273
```

Für die Zahleneingaben 13 und 21 erhalten Sie als Ergebnis 273, was dem Produkt beider Zahlen entspricht.

### 11.2.1 Der Modulo-Operator

Der so genannte Modulo-Operator wird mit dem Prozentzeichen (%) dargestellt. Wie die Operatoren für die Grundrechenarten besitzt auch der Modulo-Operator zwei Operanden.

Die Modulo-Operation gibt den Rest einer Ganzzahldivision als Wert zurück. Das bedeutet nicht, dass bei dieser Operation tatsächlich eine Division durchgeführt wird. Es wird lediglich ermittelt, wie oft der rechte Operand in dem linken vollständig enthalten ist, und der verbleibende Restwert wird zurückgegeben. Beispielsweise liefert der Ausdruck

```
9 % 4
```

den Rückgabewert 1, da nach Division beider Zahlen (9 geteilt durch 4) ein Rest von 1 verbleibt.

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

int main(void)
{
    cout << 9 % 4;
    return 0;
}
```

Ausgabe: 1

Weitere Beispiele:

```
cout << 17 % 3; // Ausgabe: 2
cout << 27 % 10; // Ausgabe: 7
cout << 2 % 7; // Ausgabe: 2
```

Die letzte Anweisung erzeugt die Ausgabe 2, da 7 in 2 zwar 0 Mal enthalten ist, aber bei der Operation nach wie vor ein Rest von 2 übrig bleibt.

### Achtung

Die Modulo-Operation ist ausschließlich auf ganzzahlige Werte anwendbar.

Die Modulo-Operation wird oft angewendet, um festzustellen, ob eine Zahl durch eine andere ohne Rest teilbar ist. Auf diese Weise lässt sich z.B. ermitteln, ob eine – möglicherweise vom Anwender eingegebene – Zahl gerade oder ungerade ist (eine Zahl ist dann gerade, wenn sie durch 2 ohne Rest teilbar ist). Für solche Prüfungen benötigen wir allerdings zusätzlich die Kenntnis von Kontrollstrukturen. Mit diesen werden wir uns in Kapitel 16 »Verzweigungen« und Kapitel 17 »Wiederholungsanweisungen« beschäftigen. Wir werden im weiteren Verlauf noch des Öfteren Gelegenheit haben, den sinnvollen Einsatz der Modulo-Operation zu demonstrieren.

## 11.3 Zusammengesetzte Zuweisungsoperatoren

Wie Sie im letzten Kapitel in Abschnitt 10.3 »Zuweisung« gesehen haben, muss eine Variable im Rahmen einer Zuweisung auf beiden Seiten des Zuweisungsoperators genannt werden, um ihren Wert in Abhängigkeit des bereits bestehenden zu erhöhen.

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

int main(void)
{
    int z = 6;
    z = z + 3;
    cout << z;
    return 0;
}
```

Ausgabe: 9

Die Variable `z` wird mit der Anweisung `z = z + 3;` um den Betrag von 3 erhöht, sodass die Ausgabe den Wert 9 zeigt.

Das eben Gesagte gilt natürlich grundsätzlich, wenn man einer Variablen einen neuen Wert zuweisen und dabei den aktuell gespeicherten in irgendeiner Weise berücksichtigen möchte:

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

int main(void)
{
    int z = 6;
    z = z / 2;
    cout << z;
    return 0;
}
```

Ausgabe: 3

In der Zuweisung `z = z / 2;` wird der bis dato gespeicherte Wert von `z` zunächst halbiert (`z / 2`), bevor er mit dem Ergebnis der Division überschrieben wird.

Für solche Fälle stellt C++ eine Reihe von zusammengesetzten Operatoren bereit, die Sie Tabelle 11.2 entnehmen können.

Operator	Bedeutung
<code>+=</code>	Addition und Zuweisung
<code>-=</code>	Subtraktion und Zuweisung
<code>*=</code>	Multiplikation und Zuweisung
<code>/=</code>	Division und Zuweisung
<code>%=</code>	Modulo und Zuweisung

**Tabelle 11.2:** Zusammengesetzte Zuweisungsoperatoren

Anstelle von `z = z + 3` kann man also ebenso `z += 3` schreiben:

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

int main(void)
{
    int z = 6;
    z += 3;
    cout << z;
    return 0;
}
```

Ausgabe: 9

Analog lässt sich Folgendes schreiben:

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

int main(void)
{
    int z = 6;
    z /= 2;
    cout << z;
    return 0;
}
```

Ausgabe: 3

Genau genommen repräsentieren zusammengesetzte Operatoren zwei Operationen in einer. Andererseits lässt sich vereinfachend feststellen: Der Operator `+=` addiert den Wert des rechten Operanden mit dem Betrag der Variablen (linker Operand), der Operator `/=` dividiert den Wert der Variablen mit dem rechten Operanden als Divisor, der Operator `*=` multipliziert den Wert der Variablen mit dem des rechten Operanden usw.

**Achtung**

Zusammengesetzte Operatoren werden vom Compiler als ein Symbol angesehen. Sie dürfen also zwischen den beiden Zeichen kein Leerzeichen einfügen.

## 11.4 Inkrement- und Dekrementoperatoren

Beispiele für unäre Operatoren sind der Inkrement- und der Dekrementoperator. Der Inkrementoperator (++) bewirkt eine Erhöhung des Wertes einer numerischen Variablen um den Betrag von 1. Der Dekrementoperator (--) verringert dagegen den Wert einer numerischen Variablen um den gleichen Betrag. Man sagt, eine Variable wird inkrementiert bzw. dekrementiert.

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

int main(void)
{
    int zahl = 3;
    zahl++;
    cout << zahl;
    return 0;
}
```

Ausgabe: 4

Im folgenden Beispiel wird die Variable zahl dekrementiert:

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

int main(void)
{
    int zahl = 3;
    zahl--;
    cout << zahl;
    return 0;
}
```

Ausgabe: 2

Die Notationen `zahl++` und `zahl = zahl + 1` (bzw. `zahl += 1`) sowie `zahl--` und `zahl = zahl - 1` (oder eben `zahl -= 1`) sind also einander gleichwertig.

## Hinweis

Möglicherweise haben Sie sich schon einmal gefragt, was das »++« im Namen C++ zu bedeuten hat. Ursprünglich sollte die Sprache C+ heißen, um anzugeben, dass es sich um eine verbesserte Version von C handelt. Das Pluszeichen wurde auch schon damals gerne verwendet, um zu kennzeichnen, dass es sich bei einem Produkt um eine verbesserte Version handelt. Stroustrup, der Erfinder der Sprache, hatte aber dann den lustigen Einfall mit den zwei Pluszeichen, da man schließlich in der Programmiersprache C syntaktisch korrekt mit ++ inkrementiert. So war der Name C++ geboren. C++ ist also gewissermaßen ein inkrementiertes C (C plus 1).

Beide Operatoren kommen sowohl in der Postfix- (zahl++) als auch in der Präfixnotation (++zahl) vor. Dies macht jedoch nur dann einen Unterschied, falls der Ausdruck, den die Operatoren mit ihren Variablen bilden, nicht alleine in einer Anweisung steht. Ist Letzteres der Fall, kommen also noch andere Operatoren vor, so wird in Verbindung mit der Präfixnotation zuerst inkrementiert bzw. dekrementiert, bevor weitere Operationen mit der Variablen stattfinden. Umgekehrt haben bei der Postfixnotation die anderen Operationen Vorrang, die Inkrementierung bzw. Dekrementierung wird erst danach durchgeführt.

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

int main(void)
{
    int zahl = 3, nummer;
    nummer = zahl++;
    cout << "Wert von nummer: ";
    cout << nummer << endl;
    cout << "Wert von zahl: " << zahl;
    return 0;
}
```

Ausgabe:

```
Wert von nummer: 3
Wert von zahl: 4
```

Im obigen Beispiel wird zunächst die Zuweisung ausgeführt und danach inkrementiert. Die Variable `nummer` erhält also den Wert 3.

Anders bei der Präfixnotation. Hier wird die Variable `zahl` vor der Zuweisung inkrementiert:

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

```
int main(void)
{
    int zahl = 3, nummer;
    nummer = ++zahl;
    cout << "Wert von nummer: ";
    cout << nummer << endl;
    cout << "Wert von zahl: " << zahl;
    return 0;
}
```

Ausgabe:

```
Wert von nummer: 4
Wert von zahl: 4
```

Hier ein Beispiel zum Dekrement in der Postfixnotation:

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

int main(void)
{
    int x = 10;
    cout << x--;
    cout << endl << x;
    return 0;
}
```

Ausgabe:

```
10
9
```

In der Anweisung `cout << x--;` wird zunächst der Wert der Variablen `x` ausgegeben, danach dekrementiert. Daher zeigt erst die darauf folgende Ausgabeanweisung den verringerten Wert von `x` an.

Anders in der Präfixnotation:

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

int main(void)
{
    int x = 10;
    cout << --x;
    cout << endl << x;
    return 0;
}
```



Ausgabe:

```
9
9
```

Inkrement- bzw. Dekrementoperatoren kommen besonders häufig im Zusammenhang mit Wiederholungsanweisungen zur Anwendung.

### Referenz

Zu Wiederholungsanweisungen siehe Kapitel 17 »Wiederholungsanweisungen«.

## 11.5 Priorität von Operatoren

Es liegt auf der Hand, dass es bei Ausdrücken wie

```
zahl = 20 + 12 / 4 * 3
```

irgendwie geregelt sein muss, in welcher Reihenfolge die einzelnen Operationen ausgeführt werden.

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

int main(void)
{
    int zahl;
    zahl = 20 + 12 / 4 * 3;
    cout << zahl;
    return 0;
}
```

Ausgabe: 29

Die Ausgabe zeigt für `zahl` den Wert 29, was Sie vermutlich vorausgesehen haben. Denn Sie wissen, dass eine Zuweisung erst nach Auswertung des rechten Operanden erfolgt (der hier von dem komplexen Ausdruck `20 + 12 / 4 * 3` gebildet wird). Außerdem ist Ihnen aus der Mathematik vermutlich die Regel »Punkt vor Strich« geläufig, die in C++ ebenfalls gilt.

Beachtung verdient die Tatsache dennoch, dass mit Ausführung der Zuweisung

```
zahl = 20 + 12 / 4 * 3;
```

insgesamt vier Operationen stattfinden: außer der Zuweisung noch eine Addition, eine Division und eine Multiplikation.

**Hinweis**

Wie Sie sicher schon bemerkt haben, trägt eine Anweisung, in der ein Zuweisungsoperator vorkommt, in der Regel den Namen besagter Operation (Zuweisung), auch wenn in derselben Anweisung mehrere Operatoren genannt sind. »Zuweisungsanweisung« klingt wohl etwas umständlich, ist aber nichtsdestoweniger korrekt.

Damit die Reihenfolge der Ausführung von Operatoren festgelegt ist, sind alle C++-Operatoren in eine mehrstufige Prioritätsrangfolge eingeordnet, wobei 17 die höchste, 1 die niedrigste Prioritätsstufe bezeichnet. Es gilt:

- Operatoren mit hoher Priorität werden vor Operatoren mit niedriger Priorität ausgewertet.
- Operatoren mit gleicher Prioritätsstufe werden entsprechend ihrer Assoziativität von links nach rechts bzw. von rechts nach links ausgewertet.

Hat ein Operator eine höhere Prioritätsstufe, so bedeutet dies also, dass die durch ihn repräsentierte Operation zuerst ausgeführt wird, wenn er zusammen mit anderen Operatoren in einem Ausdruck auftritt.

Falls sich dagegen mehrere Operatoren gleicher Prioritätsstufe in einem Ausdruck befinden, entscheidet deren Assoziativität.

Da die arithmetischen Operatoren linksassoziativ sind, wird z.B. in dem Ausdruck

```
4 + 2 + 7
```

zunächst die Addition  $4 + 2$  durchgeführt, bevor die Addition des so erhaltenen Wertes mit 7 erfolgt – was zwar hier keine Rolle spielt, in einem anderen Kontext kann die Reihenfolge der Ausführung aber durchaus von Bedeutung sein.

Aus dem gleichen Grund wird in dem Teilausdruck

```
12 / 4 * 3
```

der oben genannten Anweisung (`zahl = 20 + 12 / 4 * 3;`) die Division vor der Multiplikation ausgeführt.

**Hinweis**

Dass der Divisionsoperator `/` und der Multiplikationsoperator `*` die gleiche Priorität besitzen, können Sie auch Tabelle 11.3 entnehmen. Beiden ist die Rangstufe 13 zugeordnet, Additions- und Subtraktionsoperator haben dagegen Prioritätsstufe 12.

Anders verhält es sich dagegen beim Zuweisungsoperator. Dieser ist rechtsassoziativ.

```
zahl1 = zahl2 = 19;
```

Hier findet zunächst die Zuweisung des Wertes 19 an die Variable `zahl2` statt, im Anschluss daran die Zuweisung `zahl1 = zahl2` (dass es sich mit `zahl1` und `zahl2` um Integervariablen handelt, setzen wir natürlich voraus).

In Tabelle 11.3 sind alle bis jetzt besprochenen C++-Operatoren nach ihrer Priorität geordnet, wobei der am weitesten oben stehende Operator die höchste Priorität besitzt. Operatoren mit übereinstimmender Prioritätskennzahl befinden sich auf gleicher Prioritätsstufe.

Priorität	Operator	Bedeutung	Assoziativität
15	<code>++</code>	Inkrement	keine
15	<code>--</code>	Dekrement	keine
15	<code>-</code>	unäres Minuszeichen	keine
15	<code>+</code>	unäres Pluszeichen	keine
13	<code>*</code>	Multiplikation	von links nach rechts
13	<code>/</code>	Division	von links nach rechts
13	<code>%</code>	Modulo	von links nach rechts
12	<code>+</code>	Addition	von links nach rechts
12	<code>-</code>	Subtraktion	von links nach rechts
2	<code>=</code>	Zuweisung	von rechts nach links
2	<code>*=</code>	Multiplikation und Zuweisung	von rechts nach links
2	<code>/=</code>	Division und Zuweisung	von rechts nach links
2	<code>%=</code>	Modulo und Zuweisung	von rechts nach links
2	<code>+=</code>	Addition und Zuweisung	von rechts nach links
2	<code>-=</code>	Subtraktion und Zuweisung	von rechts nach links

**Tabelle 11.3:** Prioritätsreihenfolge der bekannten Operatoren

### Hinweis

Eine vollständige Tabelle aller C++-Operatoren finden Sie im Anhang. Dort sind auch Operatoren mit den Prioritätsstufen 17 und 1 genannt.

### Hinweis

Bei den Streamoperatoren `<<` und `>>` handelt es sich um überladene Operatoren vordefinierter Klassen. Sie gehören daher nicht unmittelbar zur Programmiersprache und sind deshalb nicht aufgeführt (zum Überladen von Operatoren siehe Kapitel 31). Beide besitzen die Prioritätsstufe 11, was den mit den gleichen Symbolen dargestellten Schiebeoperatoren entspricht (siehe Anhang).

## Hinweis

Bei den Operatoren, bei denen in der Tabelle keine Assoziativität angegeben ist, spielt diese tatsächlich keine Rolle, z.B. bei unären Operatoren oder Operatoren, die grundsätzlich nicht zusammen mit anderen auftreten. Bei den bekannten Operatoren trifft dies nur auf den Inkrement- bzw. Dekrementoperator und das Minus- bzw. Plusvorzeichen zu. In der Tabelle im Anhang sind weitere solcher Operatoren aufgelistet, die meisten davon werden Sie noch kennen lernen.

Es ist nicht erforderlich, die gesamte Rangfolge aller Operatoren auswendig zu wissen. Einige Orientierungshilfen sind jedoch nützlich:

- Arithmetische Operatoren stehen in der Rangfolge weit oben.
- Wie in der Mathematik gilt: Punkt vor Strich. Das heißt, die arithmetischen Operatoren `*` (Multiplikation) und `/` (Division) binden stärker als `+` (Addition) und `-` (Subtraktion). Zu Ersteren kommt der Modulo-Operator `%`, er hat also dieselbe Rangstufe wie `*` und `/`.
- Der Zuweisungsoperator besitzt von allen Operatoren die niedrigste Prioritätsstufe (abgesehen vom Sequenzoperator und dem Operator `throw`, siehe hierzu die Tabelle im Anhang).

Sie können in jedem Ausdruck durch Setzen von Klammern die Reihenfolge ändern:

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

int main(void)
{
    int zahl;
    zahl = (20 + 12) / 4 * 3;
    cout << zahl;
    return 0;
}
```

Ausgabe: 24

Im Ausdruck

```
(20 + 12) / 4 * 3
```

erfolgt demnach zuerst die Addition  $20 + 12$ . Der Ergebniswert dieser Operation wird dann als linker Operand in der folgenden Division verwendet ( $32 / 4$ ). Erst dann wird der somit erzeugte Wert mit 3 multipliziert ( $8 * 3$ ).

## Tipp

Regeln Sie in Zweifelsfällen die Priorität durch Setzen von Klammern. Mitunter ist dies auch im Hinblick auf eine bessere Lesbarkeit des Quellcodes empfehlenswert.

# 12

## Zahlen mit Nachkommastellen

In diesem Kapitel erfahren Sie, wie Zahlen mit Nachkommastellen im Quellcode dargestellt werden und welche Datentypen C++ zur Verarbeitung solcher Zahlen bereitstellt. Nebenbei lernen Sie den `sizeof`-Operator kennen, mit dem überprüft werden kann, wie viel Platz ein Datenobjekt im Arbeitsspeicher einnimmt. Im letzten Abschnitt (12.4 »Konstanten mit `const` deklarieren«) werden mit dem Schlüsselwort `const` deklarierte Konstanten vorgestellt und aufgezeigt, wann deren Verwendung im Quellcode sinnvoll ist.

### 12.1 Die Datentypen `double` und `float`

Zur Deklaration von Variablen, die Zahlen mit Nachkommastellen aufnehmen sollen, stellt C++ die Datentypen `double` und `float` zur Verfügung. Die Schlüsselwörter zur Angabe des Datentyps bei der Deklaration haben den gleichen Namen wie die Datentypen.

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

int main(void)
{
    float myFloat;
    double myDouble;
    return 0;
}
```

Eine Variable des Typs `float` beansprucht 4 Byte im Arbeitsspeicher, eine Variable des Typs `double` 8 Byte, entsprechend höher sind bei `double` der darstellbare Wertebereich und die Genauigkeit.

#### Hinweis

Tatsächlich sind die Größen von `float` und `double` compilerabhängig. Der C++-Standard schreibt für `float` eine Mindestgröße von 4 Byte vor, der Datentyp `double` muss mindestens so groß sein wie der von `float` (heißt: eine `float`-Variable darf nie größer sein als eine `double`-Variable). Als Höchstgrenze gilt für `float` der Wert von `double`, für `double` der Wert von `long double` (zu `long double` siehe Kapitel 14, Abschnitt 14.1.1 »Datentypqualifizierer«). In der Regel belegen `float` und `double` allerdings 4 bzw. 8 Byte im Arbeitsspeicher.

### 12.1.1 sizeof-Operator

Den Platz, den ein Datenobjekt im Arbeitsspeicher einnimmt, können Sie mithilfe des `sizeof`-Operators leicht nachprüfen. Beim `sizeof`-Operator handelt es sich um einen unären Operator. Der einzige Operand ist der Bezeichner des gewünschten Datenobjekts und ist auf der rechten Seite – am besten in Klammern – anzugeben. So gibt beispielsweise der Ausdruck

```
sizeof (myFloat)
```

die Größe der Variablen `myFloat` in Byte zurück.

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

int main(void)
{
    float myFloat;
    double myDouble;
    cout << "myFloat: ";
    cout << sizeof (myFloat);
    cout << " Byte" << endl;
    cout << "myDouble: ";
    cout << sizeof (myDouble);
    cout << " Byte" << endl;
    return 0;
}
```

Ausgabe:

```
myFloat: 4 Byte
myDouble: 8 Byte
```

Alternativ können Sie als Operanden auch den Namen eines Datentyps angeben (in Klammern gesetzt):

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

int main(void)
{
    float myFloat;
    double myDouble;
    cout << "Datentyp float: ";
    cout << sizeof (float);
}
```

```
cout << " Byte" << endl;  
cout << "Datentyp double: ";  
cout << sizeof (double);  
cout << " Byte" << endl;  
return 0;  
}
```

Ausgabe:

```
Datentyp float: 4 Byte  
Datentyp double: 8 Byte
```

Der Rückgabewert eines Ausdrucks

```
sizeof (Datentyp)
```

gibt also allgemein an, wie viele Byte ein Datenobjekt des angegebenen Typs im Arbeitsspeicher beansprucht. Da das eine Mal die Datentypnamen `float` bzw. `double` angegeben wurden, das andere Mal Variablen dieser Typen, stimmen die erhaltenen Ergebnisse natürlich überein.

### Achtung

Wenn als Operand des `sizeof`-Operators eine Datentypbezeichnung angegeben ist, dann ist das Setzen von Klammern zwingend vorgeschrieben.

Schreiben Sie also nicht

```
cout << sizeof double; // FEHLER
```

sondern

```
cout << sizeof (double); // OK
```

Dagegen ist sowohl

```
cout << sizeof (myDouble); // OK
```

als auch

```
cout << sizeof myDouble; // OK
```

erlaubt (wobei davon ausgegangen wird, dass es sich bei `myDouble` um eine zuvor bzw. um die oben deklarierte Variable handelt).

## 12.2 Literale zur Darstellung von Zahlen mit Nachkommastellen

Zurück zu unseren Variablen `myFloat` und `myDouble`. Diese sollen nun mittels einfacher Zuweisung einen Wert erhalten, sagen wir 10,5 für `myFloat` und -1936,99 für `myDouble`.

Dazu müssen Sie allerdings wissen, dass im Quellcode zur Trennung von Vor- und Nachkommastellen der Punkt zu verwenden ist. Da C++ schließlich wie die meisten anderen Programmiersprachen aus den USA stammt und dort ein Dezimalpunkt und kein Dezimalkomma üblich ist, ist dies nicht weiter verwunderlich.

Um also der Variablen `myDouble` den oben genannten Wert zuzuweisen, müssen Sie diesen im Code wie folgt schreiben:

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

int main(void)
{
    float myFloat;
    double myDouble;
    myDouble = -1936.99;
    cout << "Wert von myDouble: ";
    cout << myDouble << endl;
    return 0;
}
```

Ausgabe:

```
Wert von myDouble: -1936.99
```

### Hinweis

Die meisten Compiler warnen, falls Variablen zwar deklariert, aber im weiteren Code nicht verwendet werden, was im obigen Listing auf die Variable `myFloat` zutrifft.

Warnungen des Compilers sind zwar keine Fehlermeldungen, das heißt, der Quellcode ist trotzdem kompilierbar. In der Regel deutet eine Warnung des Compilers aber darauf hin, dass etwas nicht in Ordnung ist. Sie sollten solche Warnungen im Allgemeinen ernst nehmen und der Sache auf den Grund gehen.

Im vorliegenden Fall brauchen Sie das jedoch nicht zu tun, da wir ja beabsichtigen, die Variable `myFloat` später noch zu verwenden – ihr einen Wert zuzuweisen. Sie ist also nicht überflüssig, wie Ihr Compiler vermutlich annimmt.



Zahlen mit Nachkommastellen werden »Gleitkommawerte« genannt, die Datentypen `float` und `double` entsprechend Gleitkommadatentypen (oder kürzer: Gleitkommatypen).

### Hinweis

Weitere Bezeichnungen für Gleitkommawerte sind Gleitkommazahlen, Gleitpunkt-  
werte, Gleitpunktzahlen, Fließkommawerte und Fließkommazahlen.

Der Begriff »Gleitkommawert« kommt daher, dass der Punkt in Abhängigkeit von der Darstellung solcher Zahlen »gleitet«. So lässt sich z.B. der Wert

234.89

ebenso mit

$23489 * 10^{-2}$

oder z.B. mit

$0.23489 * 10^3$

darstellen.

### Hinweis

Tatsächlich verwendet C++ beim Speichern von Gleitkommawerten einen Teil des Speicherplatzes für die Mantisse, einen Teil für den Exponenten und ein Bit für das Vorzeichen. Bei der Mantisse handelt es sich um den Wert, mit dem die Zehnerpotenz multipliziert wird, um auf den darzustellenden Gleitkommawert zu kommen. Bei  $23489 * 10^{-2}$  ist 23489 die Mantisse.

Gleitkommawerte werden von C++ grundsätzlich in der so genannten »Normalform« gespeichert. Dabei wird der zu speichernde Wert in Exponentialschreibweise dargestellt und so formuliert, dass unmittelbar vor dem Dezimalpunkt eine Null steht. Für den Wert 234.89 aus dem Beispiel ergibt sich

$0.23489 * 10^3$

Die Mantisse beträgt hier 23489 und der Exponent 3.

Die Speicherung erfolgt nicht im Dezimal-, sondern im binären Format. Falls es sich um einen negativen Gleitkommawert handelt, wird das Vorzeichenbit auf 1 gesetzt.

Sie wissen nun, dass es zur Darstellung von Gleitkommawerten die Datentypen `float` und `double` gibt. Wie Gleitkommawerte direkt in den Quellcode geschrieben werden, ist Ihnen auch bekannt – mit dem Punkt als Trennzeichen.

Welchen Datentyp besitzt aber nun ein Literal wie 2004.55, ist es vom Typ float oder double? Es gilt:

- Gleitkommalliterale werden standardmäßig als double angenommen.
- Sie können jedoch den Datentyp float forcieren, indem Sie ein F anhängen.

Dabei darf entgegen den sonstigen Gepflogenheiten von C++ das F auch kleingeschrieben werden.

Folglich ist

```
2004.55
```

ein Literal vom Typ double. Die Literale

```
2004.55F
```

bzw.

```
2004.55f
```

sind dagegen vom Datentyp float.

### Achtung

Das F bzw. f gehört zum Literal. Sie dürfen zwischen der letzten Ziffer und dem F bzw. f kein Leerzeichen einfügen:

```
304.77 F // FEHLER
```

```
304.77F // OK
```

Nun soll auch die Variable myFloat einen Wert erhalten:

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

int main(void)
{
    float myFloat;
    double myDouble;
    myDouble = -1936.99;
    cout << "Wert von myDouble: ";
    cout << myDouble << endl;
    myFloat = 10.5f;
    cout << "Wert von myFloat: ";
    cout << myFloat << endl;
    return 0;
}
```

Ausgabe:

```
Wert von myDouble: -1936.99
Wert von myFloat: 10.5
```

Bezüglich des Wertebereichs und der Genauigkeit von float und double gilt:

- Der Datentyp float umfasst Zahlen mit Nachkommastellen im Wertebereich von  $\pm 3.4 \cdot 10^{-38}$  bis  $\pm 3.4 \cdot 10^{38}$  mit einer Genauigkeit von sechs Stellen.
- Der Wertebereich des Datentyps double erstreckt sich von  $\pm 1.7 \cdot 10^{-308}$  bis  $\pm 1.7 \cdot 10^{308}$  mit einer Genauigkeit von 15 Stellen.

»Genauigkeit von sechs bzw. 15 Stellen« heißt, Rundungsfehler treten erst auf, wenn die Zahl mehr als insgesamt sechs (für float) bzw. 15 (für double) signifikante Ziffern besitzt – das Trennzeichen nicht mitgerechnet. Es spielt also bezüglich der Genauigkeit keine Rolle, ob sich die Ziffern vor oder nach dem Trennzeichen befinden.

### Hinweis

Beispielsweise besteht das Gleitkommalliteral

00456.887

aus sechs signifikanten Ziffern, da die führenden Nullen keinen Einfluss auf den repräsentierten Wert und damit dessen Darstellung im Speicher haben.

Ebenso besitzt das Literal

799.5000

vier signifikante Ziffern.

## 12.3 Ein Programm zur Berechnung der Mehrwertsteuer

Im Folgenden soll schrittweise ein Programm entwickelt werden, das den Benutzer zur Angabe eines Nettobetrags auffordert und anschließend die daraus resultierende Mehrwertsteuer sowie den Gesamtbetrag anzeigt.

Dies ist natürlich eine verhältnismäßig leichte Aufgabe. Dennoch ist es im Allgemeinen sinnvoll, sich zu überlegen, in welcher Reihenfolge die einzelnen Aktionen ausgeführt werden müssen, insbesondere welche Informationen in welcher Richtung auszutauschen sind.

Um aus einem Nettobetrag die Mehrwertsteuer zu errechnen, ist dieser mit dem entsprechenden Faktor zu multiplizieren. Für einen Mehrwertsteuersatz von 19 % gilt also:

*Nettobetrag \* 0.19 = Mehrwertsteuerbetrag*

Der Gesamtbetrag errechnet sich damit zu

$$\text{Nettobetrag} + \text{Mehrwertsteuerbetrag} = \text{Gesamtbetrag}$$

Für den Datenfluss zwischen Programm und Benutzer heißt das:

- Allein der Nettobetrag muss eingegeben werden. Der Faktor für die Mehrwertsteuer ist bekannt.
- Auszugeben sind der Mehrwertsteuerbetrag und der Gesamtbetrag.

### Hinweis

In der Programmierung spricht man vom Entwickeln eines Algorithmus. Gewissermaßen handelt es sich dabei um das Rezept, anhand dessen anschließend die Kodierung erfolgt.

Als Erstes müssen wir dem Benutzer aber mitteilen, was er zu tun hat. Also setzen wir das Programmgerüst auf und zeigen zunächst einen Text an, der den Benutzer zur Eingabe des Nettobetrags auffordert:

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

int main(void)
{
    cout << "Nettobetrag: ";
    return 0;
}
```

Nun muss die Benutzereingabe mit `cin` entgegengenommen werden. Allerdings ist dazu eine Variable anzugeben, die den Eingabewert aufnehmen soll.

Als Bezeichner dieser Variablen ist `netto` sicherlich gut gewählt. Was den Datentyp angeht, scheidet `Integer` aus, da wir ja mit Gleitkommawerten rechnen (\* 0.19). Also ist die Wahl zwischen `float` und `double` zu treffen. Eine `float`-Variable benötigt zwar gegenüber einer `double`-Variablen weniger Speicherplatz, aber diese Überlegung soll keine Rolle spielen. Die heutigen Computersysteme sind so großzügig mit Speicher ausgestattet, dass die 8 Byte, die `double` belegt, gegenüber den 4 Byte für `float` nicht ins Gewicht fallen. Für `double` sprechen die höhere Genauigkeit und der weitaus größere Wertebereich. Wir entscheiden uns deshalb für den `double`, aber natürlich können Sie auch `float` den Vorzug geben:

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

```
int main(void)
{
    double netto;
    cout << "Nettobetrag: ";
    return 0;
}
```

In dieser Variablen speichern wir nun die Benutzereingabe:

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

int main(void)
{
    double netto;
    cout << "Nettobetrag: ";
    cin >> netto;
    return 0;
}
```

Nun gilt es, den Mehrwertsteuerbetrag auszuweisen bzw. erst einmal zu berechnen. Der entsprechende Ausdruck lautet:

```
netto * 0.19
```

Es gibt nun zwei Möglichkeiten.

Die erste ist, den Ausdruck direkt in der Ausgabeanweisung zu verwenden:

```
cout << netto * 0.19;
```

Die Alternative dazu ist, den Rückgabewert dieses Ausdrucks zunächst in einer Variablen zu speichern und in der Ausgabeanweisung den Bezeichner dieser Variablen zu verwenden:

```
double mbetrag;
mbetrag = netto * 0.19;
cout << mbetrag;
```

Falls ein Wert später noch einmal benötigt wird, empfiehlt es sich, der letztgenannten Variante den Vorzug zu geben. Im Beispiel muss noch einmal auf den Wert zurückgegriffen werden, um den Gesamtbetrag zu berechnen. Daher wird hier die zweite Variante eingesetzt.

## Hinweis

Die Anweisung

```
cout << "Gesamtbetrag: " << netto + mbetrag;
```

ist sicherlich etwas leichter nachzuvollziehen als die folgende, die ohne zusätzliche Variable auskommt:

```
cout << "Gesamtbetrag: " << netto + (netto * 0.19);
```

Am Ende dieses Kapitels werden wir aber auch diese Variante vorstellen.

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

int main(void)
{
    double netto, mbetrag;
    cout << "Nettobetrag: ";
    cin >> netto;
    mbetrag = netto * 0.19;
    cout << "Mwst: " << mbetrag;
    return 0;
}
```

Den Ausdruck `netto + mbetrag` für den Gesamtbetrag verwenden wir jetzt direkt in der Ausgabe:

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

int main(void)
{
    double netto, mbetrag;
    cout << "Nettobetrag: ";
    cin >> netto;
    mbetrag = netto * 0.19;
    cout << "Mwst: " << mbetrag;
    cout << endl << "Gesamtbetrag: ";
    cout << netto + mbetrag << endl;
    return 0;
}
```

## CD-ROM

Das Programm finden Sie als *k12e* im Ordner *Beispiele/K12* auf der Buch-CD.

Bei Eingabe eines Nettobetrags von z.B. 100.10 weist das Programm korrekterweise für die Mehrwertsteuer den Betrag 19.019 und als Gesamtbetrag 119.119 aus (Abbildung 12.1).



Abbildung 12.1: Programm zur Berechnung der Mehrwertsteuer

### Achtung

Verwenden Sie bei der Eingabe unbedingt einen Punkt zur Trennung der Nachkommastellen, wenn Sie das Programm ausführen.

### Hinweis

Was Sie tun müssen, um die Ausgabe auf zwei oder eine andere Anzahl von Nachkommastellen zu begrenzen, erfahren Sie im nächsten Kapitel (Abschnitt 13.4.2 »Anzeige von Nachkommastellen begrenzen«).

## 12.4 Konstanten mit const deklarieren

Wie Sie bereits wissen, nennt man Literale auch unbenannte Konstanten. Der Grund dafür ist, dass die entsprechenden Werte direkt im Code stehen, also nicht durch bestimmte Namen symbolisiert werden. Anders verhält es sich bei den benannten Konstanten, die daher auch als »symbolische Konstanten« bezeichnet werden. Ihr Name steht jeweils für einen bestimmten – unveränderlichen – Wert.

### Hinweis

Aus diesem Grunde gehört eine Variable eben nicht zur Gruppe der benannten Konstanten. Eine Variable symbolisiert zwar im Code einen Wert, dieser ist aber im Weiteren veränderbar – variabel.

Beispiele für vordefinierte benannte Konstanten, die fest zur Programmiersprache C++ gehören, sind die Wahrheitswerte `true` und `false` (Kapitel 14, Abschnitt 14.1 »Welche Datentypen gibt es noch?«).

### Hinweis

Es sei darauf hingewiesen, dass die Einordnung von `true` und `false` als benannte Konstanten nichts mit ihrer Eigenschaft als Schlüsselwörter zu tun hat. Was sie von anderen Schlüsselwörtern unterscheidet, ist eben, dass sie im Code für einen Wert stehen.

Aber auch der Programmierer kann im Quellcode benannte Konstanten selbst erzeugen. Dies kann auf zweierlei Weise geschehen:

- mit der `#define`-Anweisung (siehe dazu Kapitel 27, Abschnitt 27.2 »Symbolische Konstanten mit `#define` vereinbaren«)
- Deklaration mit dem Schlüsselwort `const`

Paradoxerweise handelt es sich bei solchen – mit `const` deklarierten – Konstanten eigentlich um Variablen – was genau genommen davon abhängt, wie sie vom Compiler behandelt werden (dazu gleich mehr). Entsprechend tauchen in der Literatur verschiedene Begriffe auf. Außer `const`-Konstanten werden sie mitunter auch als `const`-Variablen bzw. konstante Variablen bezeichnet.

Zumindest ihre Deklaration verläuft ähnlich der von Variablen, bis auf die Tatsache, dass dabei das Schlüsselwort `const` verwendet wird und sie noch innerhalb der Deklaration initialisiert werden müssen:

```
const int MEINEZAHL = 99;
```

### Hinweis

Es empfiehlt sich, einer guten alten Gewohnheit aus der C-Programmierung zu folgen und benannte Konstanten in Versalien zu schreiben. Man kann sie auf diese Weise im Quelltext auf einen Blick von Variablen unterscheiden.

Der Wert von solcherart deklarierten »Variablen« darf im weiteren Code nicht mehr geändert werden (was eigentlich im Widerspruch zur Bezeichnung »Variable« steht). Ein entsprechender Versuch wird als Syntaxfehler interpretiert:

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

int main(void)
{
    const int MEINEZAHL = 99;
    MEINEZAHL = 17; // FEHLER
    // ...
    return 0;
}
```

### Hinweis

Beachten Sie, dass eine mit `const` deklarierte Konstante/Variable ein R-Wert – kein L-Wert – ist, da sie ja nur auf der rechten Seite einer Zuweisung stehen kann (zu L-Werten und R-Werten siehe Kapitel 11, Abschnitt 11.1.1 »Mehrere Zuweisungen hintereinander schalten«).



Auf welche Weise der Manipulationsversuch stattfindet, ob über eine einfache Zuweisung oder eine Benutzereingabe, spielt dabei keine Rolle. Folgende Eingabeaufforderung ist also ebenfalls fehlerhaft:

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

int main(void)
{
    const int MEINEZAHL = 99;
    cin >> MEINEZAHL; // FEHLER
    // ...
    return 0;
}
```

Das Schlüsselwort `const` hat zudem einen speziellen Namen. Da es den Typ eines Datenobjekts verändert, nennt man es auch Typmodifizierer.

### Hinweis

Allerdings ist die Bezeichnung »Typ« hier nur im weiteren Sinne zu verstehen. Sie ist insofern gerechtfertigt, als `const` Einfluss darauf nimmt, wie ein Datenobjekt gespeichert wird bzw. ob es überhaupt gespeichert wird, wie wir gleich sehen werden. Auch der Begriff »Datenobjekt« ist in diesem Sinne nicht ganz korrekt, da damit eigentlich nur solche Elemente bezeichnet werden, die Platz im Arbeitsspeicher beanspruchen.

Es stellt sich nun die Frage, ob eine `const`-Konstante Platz im Arbeitsspeicher belegt oder nicht. Die Antwort ist: Dies ist von Fall zu Fall verschieden. Es hängt zum einen davon ab, ob der Wert einer `const`-Konstanten zur Übersetzungszeit bekannt ist, und zum anderen von der »Intelligenz« Ihres Compilers.

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

int main(void)
{
    const int XYZ = 35;
    cout << XYZ;
    return 0;
}
```

Der Wert von `XYZ` ist hier zur Übersetzungszeit bekannt. Wenn Ihr Compiler »schlau« genug ist, dann wird er `XYZ` wie eine Konstante behandeln – so als ob an jeder Stelle im Code, an der der Bezeichner `XYZ` verwendet wird, das Literal `35` stehen würde. Aber wie gesagt, das ist von Compiler zu Compiler verschieden.

Anders verhält es sich dagegen, falls der Wert einer `const`-Konstanten zur Übersetzungszeit nicht ermittelt werden kann. Dann bleibt Ihrem Compiler gar nichts anderes übrig, als für die Konstante Speicherplatz zu reservieren:

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

int main(void)
{
    int zahl;
    cin >> zahl;
    const int XYZ = zahl;
    cout << XYZ;
    return 0;
}
```

Die Konstante `XYZ` wird hier bei der Deklaration mit dem Wert der Variablen `zahl` initialisiert. Da die Variable `zahl` ihren Wert aber erst zur Laufzeit erhält, kann der Wert von `XYZ` zur Übersetzungszeit nicht feststehen.

### Hinweis

»Der Compiler macht dies, der Compiler macht das ...«. Dies ist eine in der Programmierliteratur allgemein übliche – wenngleich nicht ganz korrekte – Vereinfachung, der auch wir uns in diesem Buch bedienen. Tatsächlich erzeugt der Compiler die Maschinenbefehle, die dann beim Programmlauf besagte Aktionen bewirken. Oben müsste es also exakter heißen: »... bleibt Ihrem Compiler gar nichts anderes übrig, als Maschinencode zu erzeugen, der – wenn ausgeführt – für die Konstante Speicherplatz reserviert.«

Die Verwendung von symbolischen Konstanten erfolgt meist unter folgenden Gesichtspunkten:

- Man kann erreichen, dass Werte nicht versehentlich geändert werden. So wäre es z.B. unsinnig (und würde Berechnungsfehler zur Folge haben), wenn die Kreiszahl `PI` zur Laufzeit einen anderen Wert erhält.
- Bezeichner können so gewählt werden, dass sie selbsterklärend sind. Dies erhöht die Lesbarkeit des Quelltextes. So besitzen benannte Konstanten wie `MWST` oder `PI` weit mehr Aussagekraft als die Literale `19` oder `3.14`.
- Ein großer Vorteil von Konstanten ist, dass eventuelle Änderungen nur an einer einzigen Stelle im Quellcode erfolgen müssen.

Nehmen wir an, der Entwickler einer Buchhaltungssoftware müsste sein Programm überarbeiten. Der Anlass wäre eine Erhöhung der Mehrwertsteuer von 19 auf 20 Prozent. Bei Verwendung einer `const`-Konstanten müsste er den neuen Wert nur in der Deklarationsanweisung austauschen (`const int MWST = 20;`). Im weiteren Quelltext wird ja nur

mehr der Bezeichner MWST verwendet. Andernfalls müsste das Literal 19 an jeder Stelle im Code geändert werden, an der es auftritt – was bei einem Code von mehreren hundert Zeilen bereits einen erheblichen Arbeitsaufwand bedeuten würde.

In diesem Sinne lässt sich unser Mehrwertsteuerprogramm des letzten Abschnitts wie folgt umsetzen:

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

int main(void)
{
    const double MWST = 0.19;
    double netto, mbetrag;
    cout << "Nettobetrag: ";
    cin >> netto;
    mbetrag = netto * MWST;
    cout << "Mwst: " << mbetrag;
    cout << endl << "Gesamtbetrag: ";
    cout << netto + mbetrag << endl;
    return 0;
}
```

Zum Abschluss hier noch die weiter oben genannte Alternative, ohne den Mehrwertsteuerbetrag ( $\text{netto} * 0.19$  bzw.  $\text{netto} * \text{MWST}$ ) in der Variablen mbetrag zwischenspeichern – zur Verdeutlichung einmal ohne (erstes Listing) und einmal mit Verwendung der Konstanten MWST:

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

int main(void)
{
    double netto;
    cout << "Nettobetrag: ";
    cin >> netto;
    cout << "Mwst: " << netto * 0.19;
    cout << endl << "Gesamtbetrag: ";
    cout << netto + (netto * 0.19) << endl;
    return 0;
}
```

Hier das entsprechende Listing mit der const-Konstanten MWST:

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

```
int main(void)
{
    const double MWST = 0.19;
    double netto;
    cout << "Nettobetrag: ";
    cin >> netto;
    cout << "Mwst: " << netto * MWST;
    cout << endl << "Gesamtbetrag: ";
    cout << netto + (netto * MWST) << endl;
    return 0;
}
```

## CD-ROM

Es handelt sich um die Quelldateien *k12g.cpp* und *k12h.cpp* im Ordner *Beispiele/K12* der Buch-CD.

## Hinweis

Die Klammern im Ausdruck

`netto + (netto * MWST)`

sind natürlich nicht zwingend notwendig, da der Multiplikationsoperator (\*) eine höhere Priorität besitzt als der Additionsoperator (+). Sie dienen hier ausschließlich der besseren Lesbarkeit.

# 13

## Ausgabe mit Manipulatoren formatieren

Grundsätzlich sind es zwei verschiedene Dinge, wie Werte gespeichert und in welchem Format diese in der Ausgabe dargestellt werden. Falls vom Programmierer nichts anderes angegeben, verwendet `cout` bei der Ausgabe von Daten bestimmte Vorgabeeinstellungen. Diese können allerdings mittels Angabe von Manipulatoren verändert werden (zu Manipulatoren siehe auch Abschnitt 8.4 »Manipulatoren«). So können Sie z.B. im Programm zur Berechnung der Mehrwertsteuer des letzten Kapitels die Ausgabe des Mehrwertsteuerbetrags und des Gesamtbetrags auf zwei Nachkommastellen begrenzen.

### Hinweis

Der Vollständigkeit halber werden in diesem Kapitel alle Aspekte der Ausgabeformatierung – speziell auch der Exponentialschreibweise – beschrieben. Zum weiteren Verständnis sind diese Ausführungen nicht unbedingt erforderlich (und manche hier beschriebenen Dinge werden Sie nur selten brauchen), Sie können daher auch Teile überspringen und bei Kapitel 14 weitermachen. Bei Bedarf kommen Sie dann einfach noch einmal auf dieses Kapitel zurück.

## 13.1 Standardeinstellungen

Im Folgenden werden die Grundeinstellungen dargelegt, die bei der Ausgabe von Gleitkommawerten mit `cout` verwendet werden, falls im Code vom Programmierer nichts anderes bestimmt ist. Eine dieser Einstellungen bezieht sich auf die Genauigkeit der Darstellung.

### 13.1.1 Genauigkeit

Per Voreinstellung gilt die Regelung, dass Gleitkommawerte mit einer Genauigkeit von sechs signifikanten Stellen ausgegeben werden. Trennzeichen, eventuelle Vorzeichen oder der Exponent bleiben dabei außer Betracht (in welchem Zusammenhang bei der Darstellung von Gleitkommawerten Exponenten auftreten, erfahren Sie gleich weiter unten).

Der Wert

```
-12.3456
```

besitzt also in diesem Sinne sechs signifikante Stellen (123456) und wird somit noch ohne Rundungsfehler ausgegeben:

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

int main(void)
{
    cout << -12.3456;
    return 0;
}
```

Ausgabe: 12.3456

Für die Genauigkeit spielt, wie gesagt, weder das Vorzeichen noch die Position des Trennzeichens eine Rolle:

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

int main(void)
{
    cout << 1.23456;
    return 0;
}
```

Ausgabe: 1.23456

Falls der Defaultwert von sechs Stellen jedoch überschritten wird, treten in der Ausgabe Rundungsfehler auf. So besitzt z.B. der Wert 1.234567 unbestritten sieben signifikante Stellen. Daher wird bei der Ausgabe auf sechs Stellen gerundet:

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

int main(void)
{
    cout << 1.234567;
    return 0;
}
```

Ausgabe: 1.23457

An diesem Ergebnis ändert sich natürlich auch nichts, wenn besagter Wert in einer Variablen gespeichert ist.

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

```
int main(void)
{
    double aus = 1.234567;
    cout << aus;
    return 0;
}
```

Ausgabe: 1.23457

Beachten Sie, dass die Variable `aus` den Wert 1.234567 ohne Rundungsfehler speichert. Wie Sie wissen, lassen sich mit dem Datentyp `double` im positiven Bereich Werte bis  $1.7 \cdot 10^{308}$  darstellen und dies mit einer Genauigkeit von 15 Stellen. Da kann es bei der Speicherung des Wertes 1.234567 keine Probleme geben. Wie schon eingangs erwähnt, sind es prinzipiell zwei verschiedene Dinge, wie Daten gespeichert und wie diese in der Ausgabe dargestellt werden.

Nicht signifikant sind auch solche Vor- und Nachkommastellen, die keinen Einfluss auf den tatsächlichen Wert besitzen.

Demnach besitzen z.B. die Werte

```
-012.345600
```

und

```
000.1234560
```

jeweils sechs signifikante Stellen (123456).

### Hinweis

Beachten Sie, dass für den Wert 000.1234560 allein die Position des Punktes sowie die ersten sechs Ziffern nach diesem ausschlaggebend sind, keine der Nullen vor oder nach dem Punkt. Daher könnten Sie anstelle von 0.123456 – bzw. 000.1234560 – im Quellcode ebenso gut das Literal .123456 (mit Punkt – ohne Null) verwenden.

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

```
int main(void)
{
    cout << .123456;
    return 0;
}
```

Ausgabe: 0.123456

Die Ausgabe erfolgt jedoch wie gewohnt mit einer Null vor dem Komma.

Dementsprechend treten auch bei den genannten Werten keine Rundungsfehler auf.

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

int main(void)
{
    cout << -012.345600 << endl;
    cout << 000.1234560;
    return 0;
}
```

Ausgabe:

```
-12.3456
0.123456
```

Wie Sie am obigen Beispiel erkennen können, werden per Default Nullen, die für den Wert einer Gleitkommazahl ohne Bedeutung sind, nicht mit ausgegeben – abgesehen von einer allein stehenden Null vor dem Trennzeichen, was der gewohnten Darstellung entspricht.

### 13.1.2 Ausgabeformat

Eine weitere Vorgabeeinstellung bezieht sich auf das bei der Ausgabe verwendete Format. Insgesamt gibt es drei Ausgabeformate für Gleitkommawerte:

- Vorgabeformat
- Fixed-Format
- Scientific-Format

Falls im Code nicht explizit die Formate `fixed` oder `scientific` angegeben sind, gilt die Standardeinstellung, also das, was in der Aufzählung als Vorgabeformat bezeichnet ist.

Den aufgezählten Formaten liegen zwei verschiedene Schreibweisen für Zahlen mit Nachkommateil zugrunde, einmal die gewohnte in der Form

```
XXX.YYY
```

wobei *X* eine Vorkommaziffer und *Y* eine Nachkommaziffer bezeichnet – eventuell mit einem führenden Minuszeichen zur Darstellung negativer Werte.

Beispiele:

```
2749.878
0.123456
-177.52
```



Die andere ist Ihnen möglicherweise aus der Mathematik unter der Bezeichnung wissenschaftliche Schreibweise bzw. Exponentialschreibweise bekannt. In dieser Schreibweise werden die genannten Werte wie folgt dargestellt:

```
2.749878e+3  
1.23456e-1  
-1.7752e+2
```

e+2 bedeutet Exponent 2 zur Basis 10, analog e-1 Exponent -1 zur Basis 10 und e+3 Exponent 3 zur Basis 10. In der üblichen mathematischen Schreibweise sehen die Werte so aus:

```
2.749878 * 103  
1.23456 * 10-1  
-1.7752 * 102
```

Das Fixed-Format bedient sich der »normalen«, das Scientific-Format der wissenschaftlichen Schreibweise. In den Vorgabeeinstellungen verwendet cout in Abhängigkeit vom Betrag des darzustellenden Wertes eine dieser beiden Darstellungsweisen.

Dabei gilt: Ist – bezogen auf die Exponentialschreibweise mit einer Vorkommastelle – der Exponent kleiner als -4 oder größer als 5, so erfolgt die Ausgabe in dieser, andernfalls in der gewohnten Form.

### Hinweis

Mit anderen Worten: Ist der Betrag des auszugebenden Wertes (unabhängig vom Vorzeichen!) sehr groß, so erfolgt die Ausgabe in Exponentialschreibweise.

Beachten Sie, dass Gleitkommawerte in der Standardeinstellung stets mit einer Genauigkeit von sechs signifikanten Stellen ausgegeben werden. Dies gilt unabhängig davon, in welcher Form die Ausgabe erfolgt. Wie bereits erwähnt, zählen Trennzeichen, Vorzeichen und Exponent nicht zu den signifikanten Stellen.

Hierzu einige Beispiele: Der Wert

```
1.234567
```

stellt sich in der Exponentialschreibweise mit einer Stelle vor dem Komma (eigentlich müssten wir sagen »vor dem Punkt«) dar als

```
1.234567e+0
```

was gleichbedeutend ist mit  $1.234567 * 10^0$ . Der Exponent, der ja Null ist, ist weder kleiner als -4 noch größer als 5. Daher erfolgt die Ausgabe dieses Wertes in der gewohnten Form.

Andererseits hat dieser Wert sieben Stellen, das Trennzeichen nicht mitgerechnet. Gemäß der Vorgabe stehen für die Ausgabe eines Gleitkommawertes aber nur sechs Stellen zur Verfügung. Daraus ergibt sich: Der Wert wird auf sechs Stellen gerundet ausgegeben:

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

int main(void)
{
    cout << 1.234567;
    return 0;
}
```

Ausgabe: 1.23457

Dabei spielt es selbstverständlich keine Rolle, in welcher Form einzelne Werte im Programmcode notiert sind:

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

int main(void)
{
    cout << 123.4567e-2;
    return 0;
}
```

Ausgabe: 1.23457

Nehmen wir als weiteres Beispiel den Wert

```
-1234567.8
```

In Exponentialschreibweise sieht dieser so aus

```
-1.2345678e+6
```

was dem Wert  $-1.2345678 \cdot 10^6$  entspricht.

Da in der Exponentialschreibweise bei einer Vorkommastelle als Exponent 6 anzugeben ist, wird also der Wert -1234567.8 von cout in der Standardeinstellung in Exponentialform ausgegeben. Dabei treten allerdings ebenfalls Rundungsfehler auf, nachdem die Anzahl der auszugebenden Stellen per Default grundsätzlich auf sechs begrenzt ist:

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

```
int main(void)
{
    cout << -1234567.8;
    return 0;
}
```

Ausgabe: -1.23457e+06

### Hinweis

Soweit sich cout der exponentiellen Darstellung bedient, ob im Vorgabeformat oder im Scientific-Format, erfolgt die Anzeige des Exponenten in der Regel zwei- oder dreistellig und mit Vorzeichen.

### Hinweis

Wie gesagt, bleiben Vorzeichen, Trennzeichen und Exponent außen vor. In diesem Sinne bleiben bezüglich der Exponentialdarstellung des Wertes -1234567.8 immer noch acht Stellen.

**-1.2345678e+06**

Auf sechs Stellen gerundet ergibt dies

-1.23457e+06

Anders dagegen bei

```
1234560.0
```

Exponentialschreibweise:

```
1.23456e+6
```

Die Ausgabe erfolgt ebenfalls in Exponentialform, aber ohne Rundung:

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

int main(void)
{
    cout << 1234560.0;
    return 0;
}
```

Ausgabe: 1.23456e+06

**Hinweis**

Die weiter oben genannten Beispielwerte 2749.878 bzw. 0.123456 bzw. -177.52 werden also per Default alle in der üblichen Form ausgegeben (Exponent 3, -1 bzw. 2), 2749.878 allerdings mit Rundungsfehlern, da die Anzahl der signifikanten Stellen sieben beträgt (Ausgabe: 2749.88).

## 13.2 Die Manipulatoren `setiosflags()` und `resetiosflags()`

Nun zu den Formaten `fixed` bzw. `scientific`. Wie oben erwähnt, entspricht das `Fixed-Format` der gewohnten Darstellungsweise, das `Scientific-Format` stellt die Werte in `Exponentialform` dar. Allerdings gibt es einen Unterschied zur Standardeinstellung: Der Defaultwert von 6 bezüglich der Genauigkeit bezieht sich bei diesen Formaten allein auf die Nachkommastellen.

Das hat einerseits den Vorteil, dass Rundungsfehler erst sehr viel später auftreten.

Beispielsweise wird der Wert 3.123456 im `Fixed-Format` wie im `Scientific-Format` ohne Rundungsfehler dargestellt.

**Hinweis**

Für das `Fixed-Format` macht es bezüglich der Genauigkeit generell keinen Unterschied, wie viele Stellen sich vor dem Punkt befinden. An den Nachkommastellen ändert das ja nichts. Folglich treten im `Fixed-Format` auch bei dem Wert

```
33.123456
```

keine Rundungsfehler auf. Anders verhält es sich aber mit dem `Scientific-Format`, nachdem die Darstellung ja mit nur einer Stelle vor dem Trennzeichen erfolgt, also eventuell Stellen in den Nachkommasteil übergehen. Aus

```
33.123456
```

wird ja in exponentieller Schreibweise

```
3.3123456e+1
```

und dieser Wert wird, da sich somit sieben signifikante Stellen im Nachkommasteil befinden, auf sechs Nachkommastellen gerundet – sowie mit dreistelligem Exponenten – ausgegeben:

```
3.312346e+01
```

Andererseits erscheinen im `Fixed-` wie im `Scientific-Format` unter Umständen überflüssige Nullen in der Ausgabe.

So ergibt sich z.B. für den Wert 12.1 im Fixed-Format die Ausgabe:

```
12.100000
```

und im Scientific-Format:

```
1.200000e+01
```

### Hinweis

Dabei ist hier natürlich der Defaultwert von 6 für die Anzahl der auszugebenden Stellen vorausgesetzt. Dieser lässt sich jedoch mit `setprecision()` ändern (dazu gleich in Abschnitt 13.4 »`setprecision()`«).

Mit anderen Worten: Gleitkommawerte werden im Fixed- und im Scientific-Format mit einer konstanten Anzahl von Nachkommastellen – die den Einstellungen für die Genauigkeit entspricht – angezeigt.

Was müssen Sie nun tun, um eines der beiden Formate einzustellen? Dazu stehen Ihnen die Manipulatoren `setiosflags()`, `fixed` und `scientific` zur Verfügung. Beschäftigen wir uns zunächst mit `setiosflags()`.

Wie der Name schon andeutet, erlaubt dieser Manipulator, bestimmte Flags in Bezug auf Ein-/Ausgabeoperationen zu setzen – »ios« steht für »input output String«. Flags sind in der Regel einzelne Bits von Variablen, die einen bestimmten Zustand signalisieren, je nachdem, ob sie auf den Wert 1 oder 0 gesetzt sind. Das Objekt `cout` besitzt eine solche Variable und die Bits dieser Variablen regeln, wie `cout` Daten ausgibt. Das heißt, jedes einzelne Bit ist für eine ganz spezielle Einstellung diesesbezüglich zuständig.

Der Programmierer hat zwar keine direkte Zugriffsmöglichkeit auf die besagte Variable, aber über `setiosflags()` kann auf diese Einfluss genommen werden. Um z.B. das Format Scientific einzustellen, übergeben Sie `setiosflags()` den Wert `ios_base::scientific`<sup>1</sup>:

```
setiosflags(ios_base::scientific)
```

Dieser Ausdruck wird nun in den Ausgabestrom von `cout` eingebaut und da es sich bei `setiosflags()` wie bei dem Ihnen schon bekannten `endl` um einen Manipulator handelt, muss dieser in einer separaten Ausgabeeinheit verwendet werden; eine separate Anweisung ist nicht notwendig (siehe Kapitel 8, Abschnitt 8.4 »Manipulatoren«). Natürlich soll dies vor der gewünschten Ausgabe eines Gleitkommawertes geschehen.

Zudem muss zusätzlich die Headerdatei *ioomanip* – bei älteren Compilern *ioomanip.h* – eingebunden werden. Dies gilt für alle Manipulatoren, denen bei Verwendung Werte zu

1 Hinter `ios_base::scientific` verbirgt sich letzten Endes nichts anderes als eine in der C++-Standardbibliothek mit `const` definierte Konstante, vgl. Kapitel 12, Abschnitt 12.4 »Konstanten mit `const` deklarieren«. Die etwas ungewöhnliche Syntax mit dem doppelten Doppelpunkt erklärt sich daher, dass die Konstante in einer Klasse (`ios_base`) als statisches Element deklariert ist, siehe Kapitel 25.

übergeben sind, also z.B. für `setiosflags()`. Wir werden im Weiteren sehen, dass es sich mit `setprecision()` genauso verhält.

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

int main(void)
{
    cout << setiosflags(ios_base::scientific);
    cout << 7.53;
    return 0;
}
```

Ausgabe: 7.530000e+00

Denken Sie daran, dass es sich um Einstellungen handelt, die so lange bestehen bleiben, bis sie geändert werden. Nach der Angabe von `setiosflags(ios_base::scientific)` verwendet `cout` daher für alle folgenden Ausgaben von Gleitkommawerten das Scientific-Format.

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

int main(void)
{
    cout << setiosflags(ios_base::scientific);
    cout << 7.53 << endl << 22.4;
    return 0;
}
```

Ausgabe:

```
7.530000e+00
2.240000e+01
```

Auf die gleiche Weise lässt sich das Fixed-Format einstellen. In diesem Fall muss `setiosflags()` der Wert `ios_base::fixed` übergeben werden:

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

int main(void)
{
    cout << setiosflags(ios_base::fixed);
```

```
cout << 7.53 << endl << 22.4;
return 0;
}
```

Ausgabe:

```
7.530000
22.400000
```

Dass es sich hier um das Fixed-Format und nicht um die Grundeinstellung handelt, können Sie an der Ausgabe daran erkennen, dass der Nachkommateil – obwohl hinsichtlich des tatsächlichen Wertes nicht notwendig – sechsstellig dargestellt wird, also mit Nullen aufgefüllt ist.

### Hinweis

Eine mit `setiosflags()` gesetzte Einstellung lässt sich mit dem Manipulator `resetiosflags()` wieder rückgängig machen (»reset«, deutsch »zurücksetzen«). Dabei ist `resetiosflags()` genau mit dem Wert zu verwenden, mit dem die Einstellung zuvor über `setiosflags()` getroffen wurde.

Um also im vorletzten Beispiel vom Scientific-Format, das zuvor mit `setiosflags(ios_base::scientific)` eingestellt wurde, wieder zur Grundeinstellung zurückzukehren, ist an `resetiosflags()` ebenfalls dieser Wert zu übergeben:

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

int main(void)
{
    cout << setiosflags(ios_base::scientific);
    cout << 7.53 << endl;
    cout << resetiosflags(ios_base::scientific) << 22.4;
    return 0;
}
```

Ausgabe:

```
7.530000e+00
22.4
```

Mit `setiosflags(ios_base::fixed)` und `resetiosflags(ios_base::fixed)` verhält es sich natürlich genauso:

```
#include <iostream>
#include <iomanip>
```

```
using namespace std;

int main(void)
{
    cout << setiosflags(ios_base::fixed);
    cout << 7.53 << endl;
    cout << resetiosflags(ios_base::fixed) << 22.4;
    return 0;
}
```

Ausgabe:

```
7.530000
22.4
```

Der Versuch, ohne vorherige Verwendung von `resetiosflags()` auf das jeweils andere Format umzuschalten – per `setiosflags(ios_base::scientific)` oder `setiosflags(ios_base::fixed)` –, scheitert übrigens. Vielmehr wird die Grundeinstellung wiederhergestellt; der eine Aufruf setzt den anderen außer Kraft – im Unterschied zu den Manipulatoren `fixed` und `scientific` (mehr dazu im nächsten Abschnitt):

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

int main(void)
{
    cout << setiosflags(ios_base::fixed);
    cout << 7.53 << endl;
    cout << setiosflags(ios_base::scientific) << 22.4;
    return 0;
}
```

Ausgabe:

```
7.530000
22.4
```

Daher wird die Zahl 22.4 nicht im Scientific-Format ausgegeben, sondern entsprechend den Standardeinstellungen. Auch eine zweimalige Verwendung von `setiosflags()` mit dem Wert `ios_base::scientific` ändert daran nichts:

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



```
int main(void)
{
    cout << setiosflags(ios_base::fixed);
    cout << 7.53 << endl;
    cout << setiosflags(ios_base::scientific);
    cout << setiosflags(ios_base::scientific) << 22.4;
    return 0;
}
```

Ausgabe:

```
7.530000
22.4
```

Um im obigen Beispiel das Scientific-Format einzustellen, muss zunächst mit `resetiosflags(ios_base::fixed)` die Einstellung `setiosflags(ios_base::fixed)` rückgängig gemacht werden:

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

int main(void)
{
    cout << setiosflags(ios_base::fixed);
    cout << 7.53 << endl;
    cout << resetiosflags(ios_base::fixed);
    cout << setiosflags(ios_base::scientific) << 22.4;
    return 0;
}
```

Ausgabe:

```
7.530000
2.240000e+01
```

## 13.3 Die Manipulatoren `fixed` und `scientific`

Etwas einfacher als `setiosflags()` und `resetiosflags()` sind die Manipulatoren `fixed` und `scientific` zu handhaben. Sie besitzen keine Parameter (heißt: Sie müssen Ihnen bei Gebrauch keine Informationen mitteilen), daher ist auch die Einbindung der Include-Datei *iomanip* nicht erforderlich.

**Hinweis**

Es sei hier gesagt, dass es grundsätzlich keine Nachteile mit sich bringt, Header-dateien einzubinden. Solange die in ihnen definierten Elemente nicht im Code verwendet werden, hat dies keinen Einfluss auf die Größe der ausführbaren Datei.

Wie der Name schon sagt, bewirkt `fixed` eine Umstellung ins Fixed-Format, entsprechend `scientific` ins Scientific-Format:

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

int main(void)
{
    cout << fixed;
    cout << 71.99;
    return 0;
}
```

Ausgabe: 71.990000

Analog:

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

int main(void)
{
    cout << scientific;
    cout << 71.99;
    return 0;
}
```

Ausgabe: 7.199000e+01

Im Unterschied zu den Manipulatoren `setiosflags()` und `resetiosflags()` bewirkt eine Verwendung des jeweils anderen Manipulators (`fixed` bzw. `scientific`) eine Umstellung auf das entsprechende Format:

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

int main(void)
{
    cout << scientific;
    cout << 71.99 << endl;
    cout << fixed << 71.99 << endl;
}
```

```
cout << scientific << 71.99;
return 0;
}
```

Ausgabe:

```
7.199000e+01
71.990000
7.199000e+01
```

Um zur Grundeinstellung zurückzukehren, müssen Sie allerdings auf `setiosflags()` bzw. `resetiosflags()` zurückgreifen.

## 13.4 setprecision()

Wie schon angedeutet ist es möglich, die Voreinstellung bezüglich der Anzahl von Stellen, die für die Ausgabe verwendet wird, zu ändern. Dies geschieht mit dem Manipulator `setprecision()`. Diesem ist bei Verwendung die Anzahl der gewünschten Stellen zu übergeben:

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

int main(void)
{
    cout<< 1.234567 << endl;
    cout << setprecision(7);
    cout << 1.234567;
    return 0;
}
```

Ausgabe:

```
1.23457
1.234567
```

### Achtung

Alle Manipulatoren, denen beim Aufruf Werte zu übergeben sind, sind in der Headerdatei *iomanip* definiert. Daher müssen Sie diese Headerdatei mit der `#include`-Direktive einbinden, falls Sie `setprecision()` im Code verwenden wollen.

Per Default ist eine Genauigkeit von sechs Stellen eingestellt. Daher erfolgt die erste Ausgabe des Wertes 1.234567 auf sechs Stellen gerundet. `setprecision(7)` setzt die Genauig-

keit auf sieben Stellen. Das heißt, es werden nunmehr sieben Stellen für die Ausgabe verwendet. Daher treten bei der folgenden Ausgabe des Wertes 1.234567 keine Rundungsfehler auf.

Beachten Sie, dass es sich auch hier wiederum um eine Einstellung handelt, die so lange bestehen bleibt, bis explizit etwas anderes festgelegt wird:

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

int main(void)
{
    cout<< 1.234567 << endl;
    cout << setprecision(7);
    cout << 1.234567 << endl;
    cout << 1.234567 << endl;
    cout << setprecision(6);
    cout << 1.234567 << endl;
    return 0;
}
```

Ausgabe:

```
1.23457
1.234567
1.234567
1.23457
```

Natürlich lässt sich der Wert für die Genauigkeit auch verringern:

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

int main(void)
{
    cout << setprecision(2);
    cout << 1.234567;
    return 0;
}
```

Ausgabe: 1.2

Falls weder das Format `fixed` noch das Format `scientific` eingestellt ist, also das Vorgabeformat verwendet wird, richtet sich die Wahl bezüglich der Exponentialdarstellung nach der Einstellung für die Genauigkeit (Abschnitt 13.1 »Standardeinstellungen«). Da

wir diesbezüglich bis dato von der Grundeinstellung (Genauigkeit von sechs Stellen) ausgegangen sind, war die im ersten Abschnitt genannte Regel hinreichend. Sie sei zur besseren Übersicht hier noch einmal wiedergegeben:

Ist – bezogen auf die Exponentialschreibweise mit einer Vorkommastelle – der Exponent kleiner als -4 oder **größer als 5**, erfolgt die Ausgabe in dieser, andernfalls in der gewohnten Form.

Im Hinblick auf eine Änderung der Genauigkeitseinstellung müssen wir diese Regel nun wie folgt ändern:

Ist – bezogen auf die Exponentialschreibweise mit einer Vorkommastelle – der Exponent kleiner als -4 oder **größer bzw. gleich der Genauigkeit**, erfolgt die Ausgabe in dieser, andernfalls in der gewohnten Form.

### Hinweis

Wobei mit »Genauigkeit« die Anzahl von Stellen gemeint ist, die von cout für die Ausgabe verwendet werden.

Für eine Genauigkeit von drei Stellen, heißt das also:

... der Exponent kleiner als -4 oder **größer bzw. gleich 3**, ...

bzw.

... der Exponent kleiner als -4 oder **größer als 2**, ...

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

int main(void)
{
    cout << setprecision(3);
    cout << 9901.79;
    return 0;
}
```

Ausgabe: 9.9e+03

Der Wert

9901.79

besitzt in Exponentialschreibweise bei einer Vorkommastelle den Exponenten 3:

9.90179 \* 10<sup>3</sup>

Da die Genauigkeit zuvor mit

```
cout << setprecision(3);
```

auf 3 eingestellt wurde, wird der Wert in Exponentialschreibweise – auf drei Stellen gerundet – dargestellt und würde folglich so aussehen:

```
9.90e+03
```

Da in dieser Form die Null keine Bedeutung für den tatsächlichen Wert besitzt, wird sie bei der Ausgabe ebenfalls nicht berücksichtigt (Abschnitt 13.1 »Standardeinstellungen«), sodass die Ausgabe tatsächlich folgendes Aussehen hat:

```
9.9e+03
```

### 13.4.1 setprecision() in Verbindung mit Scientific-/Fixed-Format

Im Fixed- oder im Scientific-Format bezieht sich die Genauigkeit allein auf die Nachkommastellen. Das heißt, mit »Genauigkeit« ist die Anzahl von Stellen gemeint, die von cout für die Ausgabe des Nachkommateils verwendet werden:

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

int main(void)
{
    cout << setprecision(3);
    cout << scientific << 9901.79;
    return 0;
}
```

Ausgabe: 9.90e+03

Daher wird hier auf drei Nachkommastellen (!) gerundet, nicht auf drei Stellen insgesamt wie im Beispiel zuvor.

Für das Fixed-Format ergeben sich bezüglich des Wertes 9901.79 unter der Genauigkeits-einstellung 3 keine Rundungsfehler:

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

int main(void)
{
    cout << setprecision(3);
```

```
cout << fixed << 9901.79;
return 0;
}
```

Ausgabe: 9901.790

Wie im Beispiel zu erkennen, werden – im Fixed- bzw. im Scientific-Format – fehlende Stellen im Nachkommateil mit Nullen aufgefüllt. Ausgegeben wird also nicht

```
9901.79
```

sondern

```
9901.790
```

### 13.4.2 Anzeige von Nachkommastellen begrenzen

Die Anzahl der auszugebenden Nachkommastellen ist daher im Fixed-Format – wie auch im Scientific-Format – konstant, solange sich an der Genauigkeit nichts ändert. Dies lässt sich z.B. bei der Ausgabe von Eurobeträgen ausnutzen, die in der Regel mit zwei Stellen nach dem Komma (Punkt) erfolgen soll:

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

int main(void)
{
    const double preis = 9.999;
    int anzahl;
    cout << "Anzahl: ";
    cin >> anzahl;
    cout << setprecision(2) << fixed;
    cout << "Gib mir " << preis * anzahl
        << " Euro" << endl;
    return 0;
}
```

Bei einer Stückzahl von 7 verlangt das Programm 69.99 Euro (siehe Abbildung 13.1) – und nicht etwa 69.993 Euro.



Abbildung 13.1: Ausgabe des Eurobetrags mit zwei Nachkommastellen

An welcher Stelle und in welcher Reihenfolge die beiden Manipulatoren `fixed` und `setprecision()` im Code verwendet werden, spielt natürlich keine Rolle, solange dies vor der Ausgabe des Gesamtpreises (`preis * anzahl`) geschieht. In diesem Sinne lassen sich beide Manipulatoren sogar direkt in den letzten Ausgabestring einbauen:

```
cout << "Gib mir " << fixed << setprecision(2)
    << preis * anzahl << " Euro" << endl;
```

Nun können Sie auch die Ausgabe Ihres Mehrwertsteuerprogramms vom letzten Kapitel auf zwei Nachkommastellen beschränken:

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

int main(void)
{
    const double MWST = 0.19;
    double netto, mbetrag;
    cout << fixed << setprecision(2);
    cout << "Nettobetrag: ";
    cin >> netto;
    mbetrag = netto * MWST;
    cout << "Mwst: " << mbetrag;
    cout << endl << "Gesambetrag: ";
    cout << netto + mbetrag << endl;
    return 0;
}
```

Die Ausgabe für den Eingabewert 100.10 ist Abbildung 13.2 zu entnehmen. Mehrwertsteuer und Gesamtbetrag sind gegenüber dem entsprechenden Beispiel des letzten Kapitels – dort zeigte die Ausgabe 19.019 und 119.119 – auf zwei Stellen gerundet.



Abbildung 13.2: Programm zur Berechnung der Mehrwertsteuer

So viel zu Gleitkommawerten. Denken Sie daran, dass sich die obigen Ausführungen auch nur auf solche beziehen. Ganzzahlige Datentypen – gemeint sind Werte dieser Typen, also z.B. `int`-Werte – werden z.B. nie in exponentialer Form dargestellt.



## 13.5 Feldbreite setzen

Hier noch einige weitere Formatierungsmöglichkeiten. Eine davon bezieht sich auf die Feldbreite. Das ist die Anzahl von Stellen, die für die Ausgabe eines Wertes vorgesehen ist. Wenn nichts anderes bestimmt ist, nimmt jeder Wert in der Ausgabe genau so viel Platz ein, wie er eben von seiner Größe her beansprucht, also die Zahl 123 exakt drei Stellen, die Zeichenkette "C++" ebenfalls drei Stellen (die Begrenzungszeichen erscheinen ja nicht in der Ausgabe), 123.45 genau sechs Stellen. Ebenso gut kann man sagen, die entsprechenden Felder sind drei bzw. sechs Stellen groß (breit).

Die Zeichenkette

```
"    Weiter mit C++    "
```

nimmt bei der Ausgabe 20 Stellen in Anspruch, da die drei führenden und die drei abschließenden Leerzeichen ebenfalls ausgegeben werden. In Anbetracht dessen ließe sich eine Erweiterung der Feldbreite auch mittels Leerzeichen erreichen – was allerdings etwas aufwändig bzw. umständlich erscheint.

Einfacher geht es mit dem Manipulator `setw()`. Übergeben Sie diesem die gewünschte Feldbreite als `int`-Wert. Mit

```
setw(8)
```

wird also die Feldbreite auf acht Stellen eingestellt. Die Ausgabe erfolgt rechtsbündig.

### Tipp

Eine feste Feldbreite mit rechtsbündiger Ausgabe ist beispielsweise zur Ausgabe von Zahlenwerten in tabellarischer Form nützlich.

Zu berücksichtigen ist, dass sich `setw()` nur auf den nächsten auszugebenden Wert bezieht. Falls Sie die Feldbreite für darauf folgende Ausgabewerte ebenfalls neu definieren wollen, müssen Sie `setw()` für jeden Ausgabewert erneut verwenden:

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

int main(void)
{
    cout << setw(8) << "ABC";
    cout << setw(6) << "DEF";
    cout << setw(8) << "G"
        << endl << endl;
    cout << setw(8) << "Fix" << endl;
    cout << setw(14) << "und" << endl;
```

```
cout << setw(22) << "Foxi" << endl;
return 0;
}
```

## Hinweis

Es sei noch einmal darauf hingewiesen, dass für Manipulatoren mit Parametern, also auch für `setw()`, die Headerdatei *iomanip* einzubinden ist.

Die Ausgabe entspricht Abbildung 13.3.



Abbildung 13.3: Ausgabe mit veränderter Feldbreite

Sollte die angegebene Größe für den auszugebenden Wert nicht ausreichen, werden die überzähligen Stellen nicht etwa abgeschnitten, sondern die Feldbreite wird automatisch erweitert:

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

int main(void)
{
    cout << setw(3) << "Donald Duck";
    return 0;
}
```

Ausgabe: Donald Duck

Obiges Listing erzeugt daher die Ausgabe Donald Duck, nicht etwa Don. Der Manipulator `setw(3)` wird praktisch ignoriert. Der Parameter 3 gibt also die Mindestbreite des Feldes an.

### 13.5.1 Füllzeichen festlegen mit `setfill()`

In diesem Zusammenhang sei ein weiterer Manipulator genannt: `setfill()`. Mit diesem lassen sich die freien Stellen eines Feldes mit Füllzeichen ausgeben. Das gewünschte Füllzeichen wird dazu `setfill()` als `char`-Wert übergeben, z.B. in der Form

```
setfill('*')
```

Hier handelt es sich dagegen um eine Einstellung, die so lange bestehen bleibt, bis im Code etwas anderes festgelegt wird, z.B. mit der Anweisung

```
cout << setfill(' ');
```

um wieder auf Leerzeichen umzuschalten. Dieses ist schließlich das Standardfüllzeichen.

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

int main(void)
{
    cout << setfill('.');
    cout << setw(8) << "ABC";
    cout << setw(6) << "DEF";
    cout << setw(8) << "G"
        << endl << endl;
    cout << setw(8) << "Fix" << endl;
    cout << setw(14) << "und" << endl;
    cout << setw(22) << "Foxi" << endl;
    return 0;
}
```

Die Ausgabe sehen Sie in Abbildung 13.4.



Abbildung 13.4: Ausgabe mit dem Punkt als Füllzeichen

### 13.5.2 Ausgaben linksbündig/rechtsbündig ausrichten mit left/right

Mit den Manipulatoren `left` und `right` können Sie die Ausrichtung der Ausgabe innerhalb eines Feldes bestimmen:

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

```
int main(void)
{
    cout << setfill('*') << left;
    cout << setw(15) << "A" << endl;
    cout << setw(14) << "AB" << endl;
    cout << setw(13) << "ABC" << endl;
    cout << setw(12) << "ABCD" << endl;
    cout << setw(11) << "ABCDE" << endl;
    cout << setw(10) << "ABCDEF" << endl;
    cout << setw(9) << "ABCDEFG" << endl;
    cout << "ABCDEFGH" << endl;
    cout << setw(9) << "ABCDEFG" << endl;
    cout << setw(10) << "ABCDEF" << endl;
    cout << setw(11) << "ABCDE" << endl;
    cout << setw(12) << "ABCD" << endl;
    cout << setw(13) << "ABC" << endl;
    cout << setw(14) << "AB" << endl;
    cout << setw(15) << "A" << endl;
    return 0;
}
```

Der Manipulator `left` bewirkt, dass alle nachfolgenden Ausgaben linksbündig ausgerichtet werden.

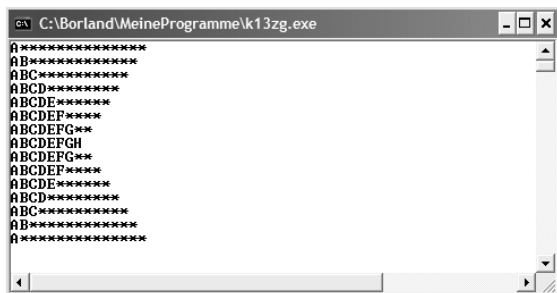


Abbildung 13.5: Die Ausgabe erfolgt innerhalb der Felder linksbündig

Eine anschließende Verwendung des Manipulators `right` würde die Ausrichtung innerhalb der Felder wieder auf rechtsbündig umstellen.

## Hinweis

Wie Sie weiter oben gesehen haben, erfolgt die Ausgabe innerhalb eines Feldes standardmäßig rechtsbündig. Insofern die Feldbreite den auszugebenden Zeichen entspricht – was in der Grundeinstellung der Fall ist –, spielt die Ausrichtung natürlich keine Rolle.

### Hinweis

Falls Ihr Compiler mit den Manipulatoren `left` und `right` nicht zurechtkommt, sei Ihnen die Methode `setf()` von `cout` empfohlen, auf die hier ansonsten nicht weiter eingegangen werden soll. Verwenden Sie diese in einer separaten Anweisung wie folgt:

Um die Ausgabe in den Feldern rechtsbündig auszurichten:

```
cout.setf(ios_base::right, ios_base::adjustfield);
```

Um die Ausgabe in den Feldern linksbündig auszurichten:

```
cout.setf(ios_base::left, ios_base::adjustfield);
```

Bei weniger aktuellen Compilern kann es sein, dass `ios` statt `ios_base` geschrieben werden muss:

```
cout.setf(ios::right, ios::adjustfield);
```

bzw.

```
cout.setf(ios::left, ios::adjustfield);
```



# 14

## Datentypen

Sie haben in den letzten Kapiteln schon einiges über Datentypen und Konstanten erfahren. Darauf aufbauend finden Sie in diesem Kapitel eine abschließende Darstellung aller Standarddatentypen von C++ sowie noch einiges Wissenswerte zum Thema Literale.

### 14.1 Welche Datentypen gibt es noch?

Die Antwort lautet: Nicht mehr viele. Die meisten haben Sie bereits kennen gelernt. Vor allem ist hier noch der Datentyp `bool` – nach dem Mathematiker George Boole, man spricht auch vom booleschen Datentyp – zu nennen. Sein Wertebereich umfasst ausschließlich die beiden Wahrheitswerte `true` (für wahr) und `false` (für falsch).

Wie an anderer Stelle bereits erwähnt, sind `true` und `false` vordefinierte Konstanten, die Sie im Code bei Bedarf verwenden können, z.B. um eine boolesche Variable zu initialisieren:

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

int main(void)
{
    bool aktiv = true;
    cout << aktiv;
    return 0;
}
```

Ausgabe: 1

Die Deklaration einer booleschen Variablen erfolgt in der gewohnten Form. Zur Angabe des Datentyps wird das Schlüsselwort `bool` verwendet, `aktiv` ist hier der Bezeichner der Variablen.

Die interne Repräsentation der beiden Wahrheitswerte ist 1 (für wahr) und 0 (für falsch). Diese Werte werden auch von `cout` bei der Ausgabe benutzt. Dementsprechend lautet die Ausgabe im obigen Beispiel 1, nicht `true`. Tatsächlich handelt es sich bei `bool` ebenfalls um einen ganzzahligen Datentyp – der sich eben nur auf zwei Werte erstreckt.

**Tipp**

Daher ließe sich im Beispiel die Variable `aktiv` ebenso mit

```
bool aktiv = 1;
```

initialisieren.

Es ist jedoch ratsam, im Quellcode zur Darstellung von Wahrheitswerten die vordefinierten Konstanten `true` und `false` zu benutzen. Diese besitzen gegenüber numerischen Literalen den Vorteil der Eindeutigkeit, was sich positiv auf die Nachvollziehbarkeit des Quellcodes auswirkt.

Damit kennen Sie die fünf Grundtypen `bool`, `char`, `int`, `float` und `double`.

Datentyp	Beschreibung
<code>bool</code>	Wahrheitswerte <code>true</code> (wahr) oder <code>false</code> (falsch)
<code>char</code>	einzelne Zeichen, z.B. <code>'A'</code> , <code>'\$'</code> , <code>'\n'</code>
<code>int</code>	ganze Zahlen
<code>float</code>	Gleitkommawerte einfacher Genauigkeit
<code>double</code>	Gleitkommawerte doppelter Genauigkeit

**Tabelle 14.1:** Grundtypen

**Hinweis**

Der Datentyp `string` ist nicht aufgeführt, da es sich bei diesem, wie Sie ja wissen, nicht um einen elementaren Datentyp, sondern um eine vordefinierte Klasse handelt.

**Hinweis**

Dem booleschen Datentyp kommt im Zusammenhang mit Kontrollstrukturen eine besondere Bedeutung zu. Kontrollstrukturen bilden das Thema des 16. und 17. Kapitels.

### 14.1.1 Datentypqualifizierer

Alle weiteren Standarddatentypen sind von einigen der genannten Grundtypen abgeleitet und zwar mit den Qualifizierern `short`, `long` bzw. `signed` oder `unsigned`.

Als Beispiel sei der Datentyp `long double` genannt, auf den wir an anderer Stelle bereits hingewiesen haben. Es handelt sich wie bei `float` und `double` um einen Datentyp zur Darstellung von Gleitkommawerten. Wie am Attribut `long` zu erkennen ist, besitzt der Datentyp `long double` einen größeren Wertebereich und auch eine größere Genauigkeit als `double`, was allerdings vom verwendeten Compiler abhängt. Der C++-Standard



(siehe hierzu Kapitel 5, Abschnitt 5.3.2 »ANSI/ISO-Standard«) schreibt lediglich vor, dass der Datentyp `long double` mindestens so groß sein muss wie der Datentyp `double`.

Tatsächlich behandeln viele Compiler `long double` genauso wie `double`, was z.B. auch auf den C++-Compiler des Visual Studio zutrifft. Dort sind Variablen des Datentyps `double` wie auch Variablen des Datentyps `long double` 8 Byte groß. Andere Compiler, z.B. der von Borland, reservieren für eine `long double`-Variable 10 Byte – im Gegensatz zu 8 Byte für `double`. Dementsprechend erstreckt sich dort der Wertebereich von `long double` von  $\pm 1.2 \cdot 10^{-4932}$  bis  $\pm 1.2 \cdot 10^{4932}$  mit einer Genauigkeit von 18 Stellen.

### Hinweis

Der Wertebereich von `double` bei einer Größe von 8 Byte:

$\pm 1.7 \cdot 10^{-308}$  bis  $\pm 1.7 \cdot 10^{308}$  mit einer Genauigkeit von 15 Stellen (Kapitel 12, Abschnitt 12.2 »Literele zur Darstellung von Zahlen mit Nachkommastellen«).

Die Deklaration einer Variablen vom Typ `long double` erfolgt unter Angabe von beiden Schlüsselwörtern – schließlich heißt der Datentyp ja auch `long double`.

Im folgenden Listing wird eine Variable `betrag` vom Typ `long double` deklariert und mit dem Wert 9872.789 initialisiert. Anschließend wird die Variable `betrag` an `cout` übergeben:

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

int main(void)
{
    long double betrag = 9872.789;
    cout << betrag;
    return 0;
}
```

Ausgabe: 9872.79

Damit sind alle Standarddatentypen zur Darstellung von Gleitkommawerten genannt.

Datentyp	Größe	Wertebereich
<code>float</code>	4 Byte	$\pm 3.4 \cdot 10^{-38}$ bis $\pm 3.4 \cdot 10^{38}$
<code>double</code>	8 Byte	$\pm 1.7 \cdot 10^{-308}$ bis $\pm 1.7 \cdot 10^{308}$
<code>long double</code>	10 Byte	$\pm 1.2 \cdot 10^{-4932}$ bis $\pm 1.2 \cdot 10^{4932}$

**Tabelle 14.2:** Gleitkommatypen

Bei den Größenangaben handelt es sich um Regelwerte. Für `double` schreibt der C++-Standard als Mindestgröße die von `float`, als Obergrenze die Größe von `long double` vor; `float` muss mindestens so groß sein wie `int`. Das Übrige hängt vom Compiler ab. Die

Datentypen `float` und `double` sind jedoch durchweg 4 bzw. 8 Byte groß. Hinsichtlich der Größe von `long double` gilt das weiter oben Gesagte (meist 8 oder 10 Byte).

Kommen wir zu den ganzzahligen Datentypen, zu denen außer `bool` und `char` – Zeichen werden ja intern ebenfalls als ganzzahlige Werte gespeichert – vor allem der Datentyp `int` zählt. In Verbindung mit dem Schlüsselwort `int` ergibt der Qualifizierer `long` den Typ `long int`. Für diesen ist eine Größe von 4 Byte vorgeschrieben.

Eine `int`-Variable ist je nach System und verwendetem Compiler 2 bzw. 4 Byte groß. In einer 32-Bit-Umgebung, die heutzutage die Regel ist, kann man jedoch 4 Byte annehmen. Um aber ganz sicherzugehen, lässt sich auch der Datentyp `long int` verwenden. Ein Datenobjekt vom Typ `long int` ist – unabhängig vom System – immer 4 Byte groß.

Alternativ zu `long int` können Sie auch nur `long` schreiben. Die Deklaration kann also lauten:

```
long int zahl;
```

oder

```
long zahl;
```

Vielfach spricht man auch nur vom Datentyp `long`. Gemeint ist dasselbe. Wie gesagt, alles, was Sie über den Datentyp `int` wissen, gilt auch für `long` – mit dem einzigen Unterschied, dass für diesen Datentyp eine Größe von 4 Byte sichergestellt ist.

Analoges gilt für den Datentyp `short int`. Ein Datenobjekt dieses Typs ist stets 2 Byte groß, nie 4 Byte. Damit sind 65536 ganze Zahlen darstellbar. Für einen vorzeichenbehafteten Datentyp `short int` ergibt sich daraus ein Wertebereich von -32768 bis +32767 (auf eine nicht vorzeichenbehaftete Abwandlung dieses Datentyps wie auch anderer Integer-typen bzw. des Typs `char` werden wir gleich zu sprechen kommen).

Auch hier gilt: Der Datentyp heißt `short int` oder kurz `short`. Dementsprechend können Sie eine `short`-Variable deklarieren mit

```
short int nummer;
```

oder

```
short nummer;
```

### Hinweis

Die Qualifizierer `short` und `long` treten also in folgenden Datentypbezeichnungen auf:

```
short int (alternativ: short)
long int (alternativ: long)
long double
```

Nun zu den Qualifizierern `signed` und `unsigned`. Falls bei der Deklaration nicht explizit `unsigned` angegeben ist, werden die Datentypen `short`, `int`, `long` und in der Regel auch `char` vom Compiler als `signed` – vorzeichenbehaftet – behandelt. Das heißt, die Hälfte des zur Verfügung stehenden Speicherplatzes wird für die Darstellung von negativen Zahlen verwendet.

### Hinweis

Die Zahl 0 wird bei dieser Betrachtung zu den positiven Zahlen gerechnet. Daher ist die kleinste darstellbare Zahl im Betrag immer um eins größer (für den Datentyp `short` ist das die Zahl -32768) als ihr Gegenstück im positiven Bereich (bei `short` der Wert 32767).

Wie gesagt, gilt dies nur für ganzzahlige Datentypen – vom Datentyp `bool` einmal abgesehen, der immer `unsigned` ist (Werte 0 und 1 bzw. `true` und `false`). Gleitkommawerte (Typen `float`, `double` und `long double`) werden intern anders dargestellt als ganzzahlige Werte.

Allerdings könnten Sie das Schlüsselwort `signed` – obwohl dies nicht notwendig ist – bei der Deklaration mit angeben:

```
signed int z;
```

ebenso wie

```
int z;
```

deklariert also eine Variable `z` vom Datentyp `int`.

Analog

```
signed short int x;
```

bzw.

```
signed short x;
```

bzw.

```
short x;
```

sowie

```
signed long int y;
```

bzw.

```
signed long y;
```

bzw.

```
long y;
```

### Achtung

Beachten Sie, dass die Qualifizierer `signed` und `unsigned` nur in Verbindung mit Integertypen (`short`, `int`, `long`) und `char` verwendet werden dürfen.

Zwar besitzen Gleitkommatypen immer einen negativen Bereich, dennoch dürfen Sie nicht schreiben

```
signed float myFloat; // FEHLER
```

und natürlich noch weniger

```
unsigned float myFloat; // FEHLER
```

Der Borland-Compiler reagiert mit Fehlermeldungen. Bei anderen Compilern hat dies Warnmeldungen und möglicherweise fehlerhafte Berechnungsergebnisse zur Folge – was die ungünstigere Variante darstellt.

Die Bezeichnungen `signed short`, `signed short int`, `signed int` usw. sind also lediglich alternative Namen für die entsprechenden Datentypen `short` bzw. `int`.

Anders verhält es sich dagegen mit den Datentypen `unsigned short` (entspricht `unsigned short int`), `unsigned int`, `unsigned long` (entspricht `unsigned long int`) und `unsigned char`. Diese sind nicht vorzeichenbehaftet. Das heißt, der zur Verfügung stehende Speicherplatz wird ausschließlich zur Darstellung von positiven Werten verwendet. Dementsprechend sind mit diesen Typen doppelt so viele positive Zahlen darstellbar. So besitzt der Datentyp `unsigned short` einen Wertebereich von 0 bis 65535 (zum Vergleich der Wertebereich des Datentyps `short`: -32768 bis 32767).

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

int main(void)
{
    unsigned short zahl = 65535;
    cout << zahl;
    return 0;
}
```

Ausgabe: 65535

Tabelle 14.3 enthält eine Zusammenstellung aller ganzzahligen Datentypen mit alternativen Namen für einen Datentyp (z.B. `short int` bzw. `signed short` bzw. `signed short int` für `short`). Die alternativen Namen beziehen sich jeweils auf die obere Zeile.

Datentyp	Größe	Wertebereich
bool	1 Byte	Wahrheitswerte true bzw. 1, false bzw. 0
char	1 Byte	-128 bis 127
signed char		
unsigned char	1 Byte	0 bis 255
short	2 Byte	-32768 bis 32767
short int, signed short, signed short int		
unsigned short	2 Byte	0 bis 65535
unsigned short int		
int	2 bzw. 4 Byte	wie short bzw. long
signed int		
unsigned int	2 bzw. 4 Byte	wie unsigned short bzw. unsigned long
long	4 Byte	-2147483648 bis 2147483647
long int, signed long, signed long int		
unsigned long	4 Byte	0 bis 4294967295
unsigned long int		

**Tabelle 14.3:** Ganzzahlige Datentypen

### Hinweis

Es sei noch erwähnt, dass einige wenige Compiler den Datentyp char auch als unsigned behandeln – was weder auf den Borland- noch auf den C++-Compiler des Visual Studios zutrifft.

## 14.2 Literale

Literale sind Konstanten, die auf direkte Weise – nicht in symbolischer Form – im Quellcode einen Wert repräsentieren (siehe auch Kapitel 11, Abschnitt 11.1 »Was ist ein Ausdruck?«). Auch jedem Literal ist einer der oben angegebenen Datentypen zuzuordnen – wie grundsätzlich jedem Wert, unabhängig davon, wie dieser im Code repräsentiert ist.

Das Wesentliche diesbezüglich ist Ihnen inzwischen bekannt, z.B. dass char-Literale vom Compiler an den einfachen Anführungszeichen (z.B. 'A') und Zeichenketten an den doppelten Anführungszeichen ("ABC") erkannt werden.

### Hinweis

In Kapitel 19, Abschnitt 19.1 »Wie Zeichenketten dargestellt werden« werden Sie erfahren, dass Zeichenketten eigentlich char-Arrays mit dem Endezeichen \0 sind.

Erwähnung verdient noch die Tatsache, dass zwischen den doppelten Anführungszeichen auch nur ein einziges Zeichen stehen darf. Das Literal

```
"A"
```

ist also ebenfalls eine Zeichenkette. Entscheidend sind allein die Begrenzungszeichen (""). Genau genommen handelt es sich um zwei Zeichen, da zur internen Darstellung noch das Endezeichen `\0` angefügt wird. Aus dem gleichen Grund besteht die Ihnen bekannte leere Zeichenkette (""), siehe Kapitel 9, Abschnitt 9.1 »Die Escape-Sequenzen `\`", `'` und `\\`«) tatsächlich aus einem Zeichen. Wie gesagt, dazu mehr in Kapitel 19.

Was die Ausgabe angeht, macht es keinen Unterschied, ob Sie ein einzelnes Zeichen als char-Literal oder als Zeichenkette in der Ausgabeanweisung schreiben:

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

int main(void)
{
    cout << 'G' << endl;
    cout << "G" << endl;
    return 0;
}
```

Ausgabe:

```
G
G
```

### 14.2.1 Literale zur Darstellung von ganzzahligen Werten

Ganzzahlige Literale bestehen aus einer Folge von Ziffern, wobei es sich, nebenbei gesagt, nicht nur um dezimale, sondern auch um hexadezimale (Basis 16) und sogar oktale (Basis 8) Ziffern handeln kann. In der Regel werden Sie jedoch in Ihren Quelltexten dezimale Ziffern verwenden.

Beachten Sie aber, dass jede Zahl, die mit einer 0 beginnt, von Ihrem Compiler als oktale Zahl angesehen wird.

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

int main(void)
{
    cout << 08; // FEHLER
    return 0;
}
```

Obiges Listing ist nicht kompilierbar. Ihr Compiler wird einen entsprechenden Versuch mit der Fehlermeldung beantworten, dass 8 keine gültige Ziffer für eine Oktalzahl ist.

### Oktalsystem

Das Oktalsystem ist ein Zahlensystem zur Basis 8 und wird häufig als Alternative zum Hexadezimalsystem (Basis 16) verwendet. Da 8 eine direkte Zweierpotenz ist ( $2^3$ ), harmonisiert das Oktalsystem ebenso wie das Hexadezimalsystem (auch 16 ist eine direkte Zweierpotenz, konkret  $2^4$ ) hervorragend mit der binären Arbeitsweise des Computers. Ein Vorteil des Oktalsystems ist, dass es nur acht Ziffern (0 bis 7) kennt und daher im Unterschied zum Hexadezimalsystem ohne die Einführung zusätzlicher Buchstaben (A-F) auskommt. Ein Nachteil ist, dass längere Zahlwerte entstehen: So lässt sich jeder 1-Byte-Wert im Hexadezimalsystem mit maximal zwei Ziffern ausdrücken (0 bis FF), im Oktalsystem benötigt man dagegen bis zu drei (0 bis 377) wie auch übrigens im Dezimalsystem (0 bis 255). Zum Hexadezimalsystem siehe auch Kapitel 9, Abschnitt 9.3 »Darstellung von Zeichen mittels Escape-Sequenzen«.

In anderen Fällen, falls die Zahl als gültige Oktalzahl interpretierbar ist, führt dies meist zu unerwünschten Ergebnissen – es sei denn, die Notation in oktaler Form ist beabsichtigt:

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

int main(void)
{
    cout << 071;
    return 0;
}
```

Ausgabe: 57

Da der Oktalzahl 71 der dezimale Wert 57 entspricht, erscheint dieser in der Ausgabe.

Möglicherweise haben Sie sich schon gefragt, welchen Datentyp ein ganzzahliges Literal hat. Schließlich stehen ja drei zur Auswahl: `short`, `int` und `long`. Mit `unsigned short`, `unsigned int` und `unsigned long` sind es sogar sechs. Es gilt:

- Solange eine ganzzahlige Konstante von der Größe her in den Datentyp `int` passt, hat sie diesen Datentyp.
- Passt sie nicht in den Datentyp `int`, aber in den Datentyp `unsigned int` (weil es sich um eine etwas größere, positive Zahl handelt), besitzt sie den Datentyp `unsigned int`.
- Passt sie weder in `int` noch in `unsigned int`, hat sie den Datentyp `long`.
- Handelt es sich um eine positive Zahl und ist deren Wert so groß, dass er auch nicht vom Datentyp `long` darstellbar ist, so besitzt die Konstante den Datentyp `unsigned long`.

Die Reihenfolge ist also `int` -> `unsigned int` -> `long` -> `unsigned long`. In der Regel haben Sie es jedoch mit `int`-Literalen zu tun, da dieser Datentyp in der 4-Byte-Variante schließlich einen Wertebereich besitzt, der mit dem von `long` identisch ist.

Die Literale

```
-3
2147483647
0
-99898765
25
```

sind z.B. alle vom Datentyp `int`. Dagegen besitzt das Literal

```
3147411649
```

den Datentyp `unsigned int`.

Demzufolge treten in dieser Form Literale vom Typ `long` im Code überhaupt nicht auf, falls der `int`-Datentyp 4 Byte groß ist. Es ist jedoch möglich, durch Anhängen des Suffixes `L` oder `l` den Datentyp `long` zu erzwingen. Die Literale

```
-3L
0L
25l
-3l
```

sind also alle vom Datentyp `long`:

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

int main(void)
{
    long zahl = 12L;
    cout << zahl;
    return 0;
}
```

Ausgabe: 12

Das Suffix `U` bzw. `u` forciert den Datentyp `unsigned int`, gegebenenfalls den Datentyp `unsigned long`, falls der Wert nicht in den Datentyp `unsigned int` passt:

```
unsigned int zahl = 2004U;
```



Die Suffixe L bzw. l und U bzw. u können auch in beliebiger Kombination auftreten. Die Literale

```
3u1
1777821u
0Lu
99898765UL
25uL
```

besitzen somit alle den Datentyp `unsigned long`.

### Hinweis

Natürlich ergibt es keinen Sinn, das Suffix U bzw. u auf ein vorzeichenbehaftetes Literal anzuwenden.

`-2005U` // ???

In der Regel wird das Suffix U bzw. u im Zusammenhang mit negativen Zahlen vom Compiler einfach ignoriert. Der Borland-Compiler tut dies mit einer Warnmeldung kund.

## 14.2.2 Gleitkommaliterale

Was Gleitkommaliterale angeht, haben Sie das meiste schon gehört (Kapitel 12, Abschnitt 12.2 »Literale zur Darstellung von Zahlen mit Nachkommastellen«). Entscheidend für die Qualifizierung als Gleitkommaliteral ist allein der Dezimalpunkt. Daher handelt es sich bei den Literalen

```
2.0
2.
.11
0.11
```

durchgehend um Gleitkommaliterale, von denen die ersten beiden (2.0 und 2.) genauso wie die letzten beiden (.11 und 0.11) jeweils denselben Wert repräsentieren. Erwähnenswert ist die Tatsache, dass 2.0 bzw. 2. Gleitkommaliterale sind, obwohl diese für einen ganzzahligen Wert stehen. Ausschlaggebend hierfür ist, wie gesagt, allein der Punkt.

Gleitkommaliterale besitzen den Datentyp `double`, unter bestimmten Voraussetzungen `long double` (falls der Wert zu groß für `double` und der Datentyp `long double` entsprechend groß ist).

Durch Anhängen des Suffixes F bzw. f kann allerdings der Datentyp `float` forciert werden (Kapitel 12, Abschnitt 12.2 »Literale zur Darstellung von Zahlen mit Nachkommastellen«).

```
6739.22F
-75.656f
```

So viel zu Literalen bzw. deren Zuordnung zu einem bestimmten Datentyp. Zur Verdeutlichung des oben Dargelegten hier noch ein paar Beispiele:

```
"1"
'1'
1
1L
1.
1.0
1.f
1.0f
```

Das erste Literal ist eine Zeichenkette ("1"), das zweite besitzt den Datentyp `char` ('1'), das dritte `int` (1), das vierte `long` (1L), das fünfte (1.) und sechste (1.0) sind vom Datentyp `double`, das siebte (1.f) und auch das achte (1.0f) weisen den Datentyp `float` auf.

# 15

## Typumwandlungen

Dass C++ eine stark typisierte Programmiersprache ist, ist Ihnen mittlerweile hinreichend bekannt und im Wesentlichen wissen Sie auch, wie mit den elementaren Datentypen umzugehen ist. In diesem Kapitel geht es um Typumwandlungen. Diese können sowohl implizit als auch explizit erfolgen. Was das zu bedeuten hat, erfahren Sie jetzt.

### 15.1 Implizite Typumwandlungen

Wie Ihnen bekannt ist, kann eine Variable entsprechend ihrer Vereinbarung nur Werte eines bestimmten Typs aufnehmen. Daher dürfen z.B. in einer Variablen des Datentyps `double` ausschließlich Werte dieses Typs gespeichert werden, und eine `char`-Variable ist allein zur Aufnahme von einzelnen Zeichen vorgesehen, also von Werten des Typs `char`.

Demnach ist z.B. bei einer Zuweisung grundsätzlich darauf zu achten, dass der Wert rechts vom Zuweisungsoperator den Datentyp der Zielvariablen besitzt:

```
double x;  
x = 275.99
```

Dennoch lässt sich folgendes Listing problemlos kompilieren und ausführen, obwohl der Variablen `zahl`, die vom Typ `double` ist, scheinbar ein `int`-Wert zugewiesen wird:

```
#include <iostream>  
using namespace std;  
  
int main(void)  
{  
    double zahl = 19;  
    cout << zahl;  
    return 0;  
}
```

Ausgabe: 19

Es ist so, dass in bestimmten Fällen, in denen C++ es als vertretbar ansieht, automatische Typumwandlungen stattfinden. C++ ist diesbezüglich allerdings sehr großzügig – die Verantwortung liegt daher beim Programmierer.

**Hinweis**

Eine Umwandlung eines Datentyps in einen anderen wird auch »Konvertierung« genannt. Falls diese automatisch erfolgt, spricht man von »impliziter Typkonvertierung«. Das Gegenstück dazu ist die »explizite Konvertierung«, die der Programmierer im Code forcieren kann (siehe dazu Abschnitt 15.4 »Explizite Typumwandlungen«).

Die Variable `zahl` hat also nicht etwa ihren Datentyp geändert, sondern der Integerwert 19 wurde vor der eigentlichen Zuweisung in den `double`-Wert 19.0 umgewandelt.

Falls auf der rechten Seite der genannten Zuweisung anstelle des Literals eine Variable steht, verhält sich das nicht anders:

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

int main(void)
{
    int myInt = 19;
    double zahl = myInt;
    cout << zahl;
    return 0;
}
```

Ausgabe: 19

Bei Ausführung der Zuweisung

```
double zahl = myInt;
```

wird die Variable `myInt` zunächst gelesen. Anschließend wird der somit erhaltene Wert (19) in den passenden Datentyp (19.0) umgewandelt.

In der Regel sind solche Konvertierungen unproblematisch, wenn der Wertebereich des Zieldatentyps größer ist als der des Quelldatentyps. Für das obige Beispiel trifft dies zu (Quelldatentyp ist `int`, Zieldatentyp ist `double`). Dennoch sollten Sie sich im Allgemeinen nicht auf implizite Typumwandlungen verlassen. In der Regel sind diese vermeidbar. Niemand hindert Sie schließlich daran, im Code anstelle von

```
double zahl = 19;
```

gleich den passenden Datentyp zu verwenden:

```
double zahl = 19.0;
```

## Hinweis

Allerdings stehen keine Literale des Datentyps `short` zur Verfügung, sodass Sie sich nötigenfalls mit Literalen des Datentyps `int` behelfen müssen – was aber gänzlich unproblematisch ist.

```
short kleineZahl = 45;
```

Außerdem können Sie dem Compiler auch ausdrücklich mitteilen, was Sie tun wollen (dazu gleich mehr in Abschnitt 15.4 »Explizite Typumwandlungen«):

```
short kleineZahl = static_cast<short> (45);
```

## 15.1.1 Konvertierung von `char` nach `int`

Besondere Erwähnung verdient die Konvertierung zwischen `char` und `int`. Wie Sie ja wissen, zählt `char` ebenfalls zu den ganzzahligen Datentypen. Der Grund dafür ist, dass Zeichen intern als ganzzahlige Werte repräsentiert werden. Nach der Zuweisung

```
char c = 'A';
```

landet z.B. nicht der Buchstabe 'A' in der Variablen `c`, sondern der numerische Wert 65, der der Ordnungszahl des Zeichens 'A' im ASCII-Code entspricht. Beim Zugriff auf die Variable im Zuge von Ausgabe- oder anderen Operationen wird der gespeicherte Zahlenwert nötigenfalls wieder in das korrespondierende Zeichen umgewandelt.

Andererseits ist es durchaus möglich, mit einer `char`-Variablen Rechenoperationen durchzuführen (dazu mehr im nächsten Abschnitt) oder ihren Wert z.B. einer Integervariablen zuzuweisen:

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

int main(void)
{
    char c = 'B';
    int ord = c;
    cout << ord;
    return 0;
}
```

Ausgabe: 66

Nach der Zuweisung `ord = c` enthält `ord` den Zahlenwert, der dem Zeichen 'B' entspricht. Letzteres ist zu diesem Zeitpunkt in der Variablen `c` enthalten. Da in der Ausgabeanweisung die Integervariable `ord` verwendet wird, wird dieser Wert als Zahl angezeigt. Gibt man dagegen dort die Variable `c` an, erscheint dieser Wert in der Ausgabe als Zeichen.

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

int main(void)
{
    char c = 'B';
    int ord = c;
    cout << c;
    return 0;
}
```

Ausgabe: B

Es stellt sich die Frage, wie `cout` Kenntnis darüber erlangt, dass die Variable `ord` bzw. deren Wert als Zahl, der Wert von `c` dagegen als Zeichen behandelt werden muss. Letzten Endes sind ja alle Werte in binärer Form gespeichert, also mit einer Folge aus einzelnen Bits, von denen jedes entweder den Wert 0 oder 1 annehmen kann.

Für die Interpretation eines Wertes als Zeichen, als Ganzzahl oder als Gleitkommazahl ist letztlich dessen Bitmuster entscheidend. So besteht z.B. eine `char`-Variable aus 8 Bit (entspricht 1 Byte), eine `int`-Variable in der Regel aus 32 Bit (4 Byte). Eine `float`-Variable beansprucht ebenfalls 32 Bit, von denen ein Teil als Mantisse, ein Teil als Exponent und ein Bit als Vorzeichen verwendet wird. Im Unterschied dazu wird bei ganzzahligen Datentypen die gesamte Bitfolge als ein numerischer Wert interpretiert.

### Hinweis

Ergänzend sei gesagt, dass bei `signed`-Datentypen das linke Bit, wenn es auf 1 gesetzt ist, einen negativen Wert anzeigt. Für `unsigned`-Typen steht dieses Bit zusätzlich zur Verfügung. Daher besitzen `unsigned`-Typen einen doppelt so großen positiven Wertebereich wie ihre entsprechenden `signed`-Typen.

Wenn ein Wert konvertiert wird, bedeutet das eigentlich, dass sich die Interpretation seines Bitmusters ändert. Das heißt in Bezug auf die Anweisung

```
int ord = c;
```

nichts anderes, als dass die in der `char`-Variablen `c` enthaltene Information auf eine Weise in der Variablen `ord` (an dem entsprechenden Speicherort im Arbeitsspeicher) abgelegt wird, die der Datentyp `int` vorschreibt. Wenn Ihr Compiler das für vertretbar hält, wird er die Zuweisung durchführen. Das heißt, er wird das Bitmuster des zuzuweisenden Wertes in einer Weise interpretieren, die dem Datentyp der Zielvariablen entspricht. Mit anderen Worten: Ihr Compiler wird die Typkonvertierung durchführen und zwar implizit, das heißt ohne dass Sie ihm dies durch entsprechende Sprachmittel im Code ausdrücklich mitgeteilt hätten.

Hier ein kleines Programm, das den Benutzer zur Eingabe eines Zeichens auffordert und ihm anschließend die Ordnungsnummer anzeigt, die diesem Zeichen im ASCII-Code entspricht:

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

int main(void)
{
    char c;
    int ord;
    cout << "Geben Sie ein Zeichen ein: ";
    cin >> c;
    ord = c;
    cout << "Ordnungsnummer: " << ord << endl;
    return 0;
}
```

Bei Eingabe des Zeichens »A« gibt das Programm die Ordnungsnummer 65, bei Eingabe des Zeichens »9« die Ordnungsnummer 57 aus. Beides ist korrekt, da dem Zeichen '9' tatsächlich die Ordnungsnummer 57 im ASCII-Code entspricht.

Jedoch sollte obiges Programm auf die Eingabe des Zeichens »ß« hin eigentlich die Ordnungszahl 225 anzeigen, was leider nicht der Fall ist. Hingegen weist das Programm für dieses Zeichen den Wert -31 aus.

Das kommt daher, dass der char-Datentyp standardmäßig als signed behandelt wird. Somit werden die Zeichen des erweiterten ASCII-Codes als negative Werte abgelegt (zum ASCII-Code bzw. erweiterten ASCII-Code siehe Kapitel 9, Abschnitt 9.2 »ASCII-Code« sowie Abschnitt 9.3.1 »Ausgabe von Umlauten sowie des Zeichens ß«). Um das zu ändern, reicht es aus, bei der Deklaration der char-Variablen das Schlüsselwort `unsigned` zu verwenden:

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

int main(void)
{
    unsigned char c;
    int ord;
    cout << "Geben Sie ein Zeichen ein: ";
    cin >> c;
    ord = c;
    cout << "Ordnungsnummer: " << ord << endl;
    return 0;
}
```

Nach Eingabe des Zeichens »ß« weist das Programm nun richtigerweise die Ordnungszahl 225 aus.

## CD-ROM

Das Programm – .cpp und .exe-Datei – finden Sie als *k15f* im Ordner *Beispiele/K15* auf der Buch-CD.

Selbstverständlich sind auch Konvertierungen in die andere Richtung möglich, von `int` nach `char`. In Abwandlung des obigen Beispiels erwartet folgendes Programm die Eingabe der Ordnungszahl und gibt im Anschluss das zugehörige Zeichen aus:

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

int main(void)
{
    unsigned char c;
    int ord;
    cout << "Geben Sie die ASCII-Code-Nummer an: ";
    cin >> ord;
    c = ord;
    cout << "Das Zeichen mit der Ordnungsnummer "
         << ord << " ist " << c << endl;
    return 0;
}
```

Ausgabe:

```
Geben Sie die ASCII-Code-Nummer an: 65
Das Zeichen mit der Ordnungsnummer 65 ist A
```

Nach Eingabe von 65 gibt das Programm erwartungsgemäß das Zeichen A aus (bei 225 das Zeichen 8).

## Hinweis

Hier wird vorausgesetzt, dass der Benutzer nur Zahlen eingibt, die darstellbaren Zeichen des ASCII-Codes entsprechen.

## 15.2 Wann gehen bei der Konvertierung Informationen verloren?

Nicht alle Konvertierungen verlaufen jedoch so problemlos wie die eben gezeigten. Häufig kommt es dabei zu Informationsverlusten. Dies ist z.B. bei der Konvertierung von `double` bzw. `float` nach `int` der Fall, da eine Integervariable bekanntlich keine Nachkommastellen aufnehmen kann:



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

int main(void)
{
    int num = 2.9;
    cout << num;
    return 0;
}
```

Ausgabe: 2

Die Konvertierung findet zwar statt, eventuelle Nachkommastellen werden dabei aber abgeschnitten. Es wird also nicht etwa gerundet. Demzufolge wird der Gleitkommawert 2.9 vor der eigentlichen Zuweisung in den `int`-Wert 2 konvertiert.

Falls der Zieldatentyp größer ist als der Quelldatentyp, ist sichergestellt, dass der zuzuweisende Wert tatsächlich in den Zieldatentyp passt. Auf der anderen Seite führt C++ Konvertierungen auch dann durch, wenn der Quelldatentyp größer ist als der Zieldatentyp, was z.B. in der Richtung von `int` nach `short` oder von `double` nach `float`, aber auch von `double` oder `float` nach `int` zutrifft.

### Hinweis

Sowohl `double` als auch `float` besitzen bekanntlich einen weitaus größeren Wertebereich als `int`.

Hier ist besondere Vorsicht geboten, da eine implizite Konvertierung sogar dann erfolgt, wenn der zuzuweisende Wert den Wertebereich des Zieldatentyps tatsächlich überschreitet:

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

int main(void)
{
    short kleineZahl = 65539;
    cout << kleineZahl;
    return 0;
}
```

Ausgabe: 3

Der Wert 65539 wird also in den Wert 3 konvertiert und dieser dann der Variablen `kleineZahl` zugewiesen.

Eine `short`-Variable kann im positiven Bereich Werte bis 32767 aufnehmen (Tabelle 14.3). Falls diese Grenze wie im obigen Beispiel überschritten wird, beginnt die Zählung wieder von vorn, das heißt beim kleinsten Wert dieses Datentyps. Eine Zuweisung des Wer-

tes 32768 hat demnach die Konvertierung dieses Wertes zu -32768 zur Folge, 32769 wird zu -32767 konvertiert, 32770 zu -32766 usw.

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

int main(void)
{
    short kleineZahl = 32768;
    cout << kleineZahl;
    return 0;
}
```

Ausgabe: -32768

## 15.3 Welchen Datentyp hat ein bestimmter Ausdruck?

In Kapitel 11, Abschnitt 11.1.2 »Komplexe Ausdrücke« haben Sie erfahren, dass grundsätzlich jeder Ausdruck – also auch jeder Teilausdruck – einen Rückgabewert besitzt. Dieser entspricht dem Wert, den der besagte Ausdruck nach Auswertung besitzt. So ist in der Zuweisung

```
x = 17 + 3;
```

einmal die ganze Zuweisung als Ausdruck anzusehen (wir setzen voraus, dass es sich bei `x` um eine Integervariable handelt); Rückgabewert ist der Wert, den die Variable `x` nach der Zuweisung besitzt (Kapitel 11, Abschnitt 11.1.1 »Mehrere Zuweisungen hintereinander schalten«). Aber auch der Term `17 + 3` und im weiteren Sinne sogar die Literale `17` und `3` als solche sind definitionsgemäß Ausdrücke mit den Rückgabewerten `20` bzw. `17` bzw. `3`.

### Hinweis

Jeder Teil einer Anweisung, der einen Wert repräsentiert, ist ein Ausdruck (Kapitel 11, Abschnitt 11.1 »Was ist ein Ausdruck?«). Mitunter fällt darunter auch die ganze Anweisung selbst, wie es etwa in der obigen Zuweisung der Fall ist (genau genommen ohne das abschließende Semikolon, das allein als Trennzeichen zwischen Anweisungen fungiert).

Aus dem Gesagten ergibt sich, dass auch jeder Ausdruck einen Datentyp besitzt, nämlich den Datentyp seines Rückgabewertes. So besitzen in der Anweisung

```
x = 17 + 3;
```

die Ausdrücke

```
x = 17 + 3
```

und

```
17 + 3
```

beide den Datentyp `int`, was dem Datentyp aller auftretenden Operanden entspricht. Beide Operanden der arithmetischen Addition (`17 + 3`) sind vom Datentyp `int`, daher besitzt der Ergebniswert dieses Ausdrucks den gleichen Datentyp. Genauso verhält es sich mit den Operanden der Zuweisung. Die Variable `x` ist vom Datentyp `int` (was wir vorausgesetzt haben) und der rechte Operand (`17 + 3`) ebenfalls, wie wir gerade festgestellt haben.

### Hinweis

Beachten Sie, dass der Term

```
17 + 3
```

für sich betrachtet ein Ausdruck ist, da er ja für einen Wert steht. Auf der anderen Seite bildet er den rechten Operanden der Zuweisung

```
x = 17 + 3
```

ist also wiederum Teil eines komplexeren Ausdrucks.

Was geschieht nun aber, falls die Operanden eines Ausdrucks unterschiedlichen Datentyps sind? Dann wird der Compiler versuchen, die entsprechenden Datentypen umzuwandeln, wie es in den Beispielen des letzten Abschnitts gezeigt wurde. So wird in der Zuweisung

```
double zahl = 19;
```

zunächst der rechte Operand konvertiert (von `19` nach `19.0`), also dem Datentyp des linken Operanden (der Zielvariablen `zahl`) angepasst.

Der genannte Sachverhalt lässt sich mit anderen Worten wie folgt wiedergeben: Operationen können grundsätzlich nur mit Operanden gleichen Datentyps durchgeführt werden. Gegebenenfalls müssen die Datentypen einander angepasst werden.

### Hinweis

Wenn eine solche Anpassung allerdings nicht möglich ist, wird der Compiler die Arbeit verweigern:

```
double zahl = "Ich darf nicht rein"; // FEHLER
```

**Hinweis (Forts.)**

Dies ist ebenso der Fall, wenn er die Konvertierung für sehr kritisch hält:

```
double zahl = "19"; // FEHLER
```

Obwohl der String "19" als Zahl interpretierbar ist, von daher eine Konvertierung also möglich wäre, wird eine solche nicht durchgeführt. C++ bzw. Ihr Compiler nimmt in diesem Fall nicht mehr an, dass Sie wissen, was Sie tun, und will Sie so vor eventuellen Fehlern schützen.

Einige der Regeln, nach denen die Anpassung (Konvertierung) geschieht, haben Sie im letzten Abschnitt bereits erfahren. Hier einige weitere Beispiele:

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

int main(void)
{
    int zahl;
    zahl = 17 + 3.3;
    cout << zahl;
    return 0;
}
```

Ausgabe: 20

In der Zuweisung

```
zahl = 17 + 3.3;
```

wird zunächst der Ausdruck rechts vom Zuweisungsoperator ausgewertet.

Dieser besteht aus zwei Operanden, die an der durch das »+«-Zeichen symbolisierten arithmetischen Addition beteiligt sind. Die Operanden besitzen verschiedene Datentypen, der linke (17) den Typ `int`, der rechte (3.3) den Typ `double`. Also sollte eine Konvertierung erfolgen, entweder des linken Operanden von `int` (17) nach `double` (17.0) oder des rechten von `double` (3.3) nach `int` (3), wobei im letzten Fall die Nachkommastellen verloren gehen (siehe Abschnitt 15.2 »Wann gehen bei der Konvertierung Informationen verloren?«).

Es gilt die Regel:

Abgesehen von der Zuweisung, in der sich die Konvertierung stets nach der Zielvariablen zu richten hat, und einigen weiteren Ausnahmen, die noch zu nennen sind, erfolgt in Ausdrücken die Konvertierung in den größten Datentyp.

### Hinweis

Ausnahmen von der genannten Regel treten unter anderem in Verbindung mit Funktionsaufrufen auf (zu Funktionen siehe Kapitel 20). Soweit beim Funktionsaufruf die Argumente im Datentyp nicht mit den formalen Parametern übereinstimmen, erfolgt gegebenenfalls eine Konvertierung der Argumente nach dem Datentyp der formalen Parameter. Dasselbe gilt für den Rückgabewert einer Funktion, der gegebenenfalls dem in der Definition angegebenen Datentyp angepasst wird. Da es sich bei formalen Parametern von Funktionen wie bei Funktionsrückgabewerten um Variablen handelt – bei Funktionsrückgabewerten um temporäre –, verläuft die Konvertierung analog dem bereits bekannten Prinzip, sie richtet sich nach dem Datentyp der Zielvariablen.

Nach dem Gesagten erfolgt im Ausdruck

```
17 + 3.3
```

die Konvertierung des linken Operanden nach `double`, von 17 zu 17.0, sodass der Rückgabewert dieses Ausdrucks (20.3) – und damit der Ausdruck als solcher – ebenfalls vom Typ `double` ist.

### Hinweis

Man sagt also, der Ausdruck `17 + 3.3` ist vom Datentyp `double`. Denken Sie daran, dass der Datentyp eines Ausdrucks vom Datentyp seines Rückgabewertes bestimmt wird.

Nach erfolgter Auswertung des Ausdrucks `17 + 3.3` ergibt sich für die Zuweisung

```
zahl = 20.3;
```

wobei der rechte Wert dem Datentyp der Zielvariablen angepasst werden muss. Dieser ist `int`, also erfolgt die Konvertierung des `double`-Wertes 20.3 nach `int` (20), wobei die Nachkommastelle verloren geht. Daher lautet die Ausgabe 20.

### Hinweis

Der Datentyp des Ausdrucks `zahl = 17 + 3.3` ist also `int`, ebenso der Datentyp seines Rückgabewertes.

## 15.3.1 Ausdrücke als Operanden des `sizeof`-Operators

In Kapitel 12, Abschnitt 12.1.1 haben Sie den `sizeof`-Operator kennen gelernt. Dort wurde diesem der Name eines Datentyps bzw. der Bezeichner einer Variablen übergeben. Rückgabewert ist die Größe des Datentyps bzw. die Größe der bezeichneten Variablen. Es ist auch möglich, dem `sizeof`-Operator beliebige Ausdrücke zu übergeben, also

z.B. den Ausdruck  $17 + 3.3$ . Der `sizeof`-Operator liefert dann die Größe des Ausdrucks als Wert zurück, wobei die Größe eines Ausdrucks wiederum dem Datentyp seines Rückgabewertes entspricht.

Somit können Sie leicht nachprüfen, dass die Konvertierung tatsächlich in der oben beschriebenen Weise stattfindet und nicht etwa zunächst nach `int`, also  $17 + 3.3$  zu  $17 + 3$ , wonach eine erneute Anpassung in Bezug auf die folgende Zuweisung des Ergebnisses an die Variable `zahl` gar nicht mehr notwendig wäre. Denn wäre der Ausdruck  $17 + 3.3$  vom Typ `int`, würde `sizeof(17 + 3.3)` den Wert 4 (Byte) zurückliefern, andernfalls 8 für `double`. Letzteres ist der Fall, was die obigen Ausführungen bestätigt:

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

int main(void)
{
    cout << sizeof(17 + 3.3);
    return 0;
}
```

Ausgabe: 8

Ebenfalls in Übereinstimmung mit dem oben Dargelegten erhält man den Wert 4, falls der ganze Ausdruck `zahl = 17 + 3` an `sizeof` übergeben wird:

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

int main(void)
{
    int zahl;
    cout << sizeof(zahl = 17 + 3.3);
    return 0;
}
```

Ausgabe: 4

Beachten Sie, dass `sizeof` allein einen ihm übergebenen Ausdruck bewertet, jedoch die symbolisierten Operationen nicht tatsächlich durchführt. Bezogen auf obiges Listing sollten Sie also die Variable `zahl` nicht in einer Ausgabeanweisung verwenden, solange diese keinen wohldefinierten Wert besitzt:

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

int main(void)
{
    int zahl;
    cout << sizeof(zahl = 17 + 3.3);
    cout << endl;
}
```

```
cout << zahl; // ???(PROBLEMATISCH)
return 0;
}
```

Dass sizeof die Zuweisung tatsächlich nicht durchführt, lässt sich nachweisen, indem man die Variable zahl mit einem Anfangswert versieht und nach der sizeof-Operation (der Bewertung des übergebenen Ausdrucks) wieder ausgibt:

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

int main(void)
{
    int zahl = 0;
    cout << sizeof(zahl = 17 + 3.3);
    cout << endl;
    cout << "Wert von zahl: " << zahl;
    cout << endl;
    return 0;
}
```

Ausgabe:

```
4
Wert von zahl: 0
```

Ausgegeben wird also der Initialisierungswert 0 und nicht etwa das Ergebnis einer eventuellen Zuweisung `zahl = 17 + 3.3`.

### 15.3.2 Datentypen werden nicht konvertiert

Es sei darauf hingewiesen, dass eine Variable ihren Datentyp grundsätzlich nicht ändern kann. Allenfalls erfolgt die Konvertierung einer Kopie:

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

int main(void)
{
    int zahl, ganzeZahl = 17;
    zahl = ganzeZahl + 3.3;
    cout << zahl;
    return 0;
}
```

Ausgabe: 20

Die Konvertierung in den Ausdrücken

```
ganzeZahl + 3.3
```

und

```
zahl = ganzeZahl + 3.3
```

findet wie oben beschrieben statt. Dass hier anstelle eines Literals (17) der Bezeichner einer Variablen steht, macht insofern keinen Unterschied. Außerdem wird nicht die Variable `ganzeZahl` selbst konvertiert, sondern eine Kopie ihres Wertes.

### Hinweis

Im Zuge eines Lesezugriffs auf eine Variable wird der in ihr enthaltene Wert kopiert. Alle weiteren Operationen finden dann mit diesem kopierten Wert statt, die Variable selbst ist daran nicht beteiligt.

Die Variable `ganzeZahl` besitzt daher auch nach der Konvertierung (ihrer Kopie) den Datentyp `int` und nimmt somit im Arbeitsspeicher weiterhin 4 Byte in Anspruch, nicht etwa 8 Byte, was dem Speicherbedarf einer `double`-Variablen entsprechen würde.

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

int main(void)
{
    int zahl, ganzeZahl = 17;
    zahl = ganzeZahl + 3.3;
    cout << "Wert von zahl: " << zahl << endl;
    cout << "Wert von ganzeZahl: " << ganzeZahl << endl;
    cout << "Gr\x94\xE1" << "e von ganzeZahl: ";
    cout << sizeof(ganzeZahl) << " Byte" << endl;
    return 0;
}
```

Ausgabe:

```
Wert von zahl: 20
Wert von ganzeZahl: 17
Größe von ganzeZahl: 4 Byte
```



**Hinweis**

Beachten Sie, dass in der Anweisung

```
cout << "Gr\x94\xE1" << "e von ganzeZahl: ";
```

die Ausgabe auf zwei Einheiten verteilt ist, da ansonsten E1e und nicht wie beabsichtigt E1 als Hexadezimalwert interpretiert wird – die Escape-Sequenz \xE1 steht für das Zeichen ß, \x94 für das Zeichen ö (siehe dazu Kapitel 9, Abschnitt 9.3.1 »Ausgabe von Umlauten sowie des Zeichens ß«).

**15.3.3 Reihenfolge der Konvertierungen**

Anders als im oben gezeigten Beispiel ist es für das Ergebnis eines Ausdrucks häufig entscheidend, in welcher Reihenfolge die Konvertierungen stattfinden. Hier ist besonders bei der Integerdivision Vorsicht geboten:

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

int main(void)
{
    int z1 = 7, z2 = 2;
    double ergebnis, gleit = 2.0;
    ergebnis = z1 / z2 * gleit;
    cout << ergebnis;
    return 0;
}
```

Ausgabe: 6

Die Variable `ergebnis` bekommt nicht etwa den Wert 7.0 zugewiesen, wie Sie möglicherweise erwartet haben, sondern den Wert 6.0, wie an der Ausgabe zu sehen ist. Das kommt daher, dass zunächst der Teilausdruck

```
z1 / z2
```

ausgewertet wird, da Division und Multiplikation gleiche Priorität besitzen und links-assoziativ sind (Kapitel 11, Abschnitt 11.5 »Priorität von Operatoren«). In diesem Ausdruck (`z1 / z2`) sind keine Konvertierungen nötig, da beide Operanden gleichen Datentyps – `int` – sind. Infolgedessen wird die Integerdivision

```
7 / 2
```

durchgeführt, wobei die Nachkommastellen wegfallen. Der Ergebniswert dieser Operation ist also 3 – nicht 3.5. Dieser Wert (3) ist anschließend an der Multiplikation

```
3 * gleit
```

beteiligt. Hier wird 3 zunächst nach 3.0 konvertiert, was für den Betrag des Rückgabewertes (6.0) bezüglich des Ausdrucks

```
z1 / z2 * gleit
```

aber keine Bedeutung mehr hat.

Anders verläuft die Konvertierung, wenn man im Code – durch entsprechende Klammerung – festlegt, dass die Multiplikation zuerst ausgeführt wird:

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

int main(void)
{
    int z1 = 7, z2 = 2;
    double ergebnis, gleit = 2.0;
    ergebnis = z1 / (z2 * gleit);
    cout << ergebnis;
    return 0;
}
```

Ausgabe: 1.75

Nun wird zunächst der Ausdruck

```
z2 * gleit
```

ausgewertet, dabei vor der eigentlichen Multiplikation `z2` von `int` nach `double` konvertiert (2 zu 2.0), der Regel folgend, dass in Ausdrücken mit Operanden unterschiedlichen Datentyps in den größeren Datentyp konvertiert wird. Das Ergebnis, das ebenfalls vom Typ `double` ist, bildet nun den rechten Operanden der folgenden Division:

```
7 / 4.0
```

Hier wird der linke Operand ebenfalls in den größeren Datentyp – `double` – umgewandelt. Beachten Sie, dass es dabei nicht auf den tatsächlichen Wert der Operanden ankommt – die hier ja beide ganzzahlig sind –, sondern allein auf deren Datentyp. Somit wird die Division mit `double`-Werten durchgeführt, wodurch keine Daten verloren gehen:

```
7.0 / 4.0
```

Dementsprechend erhält die Variable `ergebnis` mit der Zuweisung den Wert 1.75.

**Hinweis**

Dies entspricht natürlich ebenfalls nicht dem Ergebnis von  $7 / 2 * 2$ , da durch die Klammerung – berechnet wurde  $7 / (2 * 2)$  – die Ausführungsreihenfolge verändert wurde. Was Sie tun müssen, um diese Operationen in der gewöhnlichen Reihenfolge (von links nach rechts) ohne Datenverluste durchzuführen, erfahren Sie unten in Abschnitt 15.4 »Explizite Typumwandlungen«.

**15.3.4 char und short**

Wir hatten weiter oben bereits angedeutet, dass eine char-Variable auch an arithmetischen Operationen beteiligt sein kann. Hierzu ein Beispiel:

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

int main(void)
{
    char zeich = 'A';
    cout << 35.7 + zeich;
    return 0;
}
```

Ausgabe: 100.7

Die Ausgabe entspricht der Summe aus der Ordnungszahl des Zeichens 'A' im ASCII-Code und dem Wert 35.7. Der char-Wert 'A' wird vor der Addition nach double umgewandelt, dem Datentyp des linken Operanden.

Wenn man in der Ausgabeanweisung statt der Variablen zeich das Literal 'A' verwendet, erhält man das gleiche Ergebnis:

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

int main(void)
{
    cout << 35.7 + 'A';
    return 0;
}
```

Ausgabe: 100.7

Eine Ausnahme bzw. Ergänzung zu der Regel, dass in Ausdrücken die Konvertierung in den größten Datentyp erfolgt, bezieht sich auf Datentypen, die kleiner sind als int, was auf die Datentypen char und short zutrifft. Diese werden auf jeden Fall in den Datentyp int konvertiert:

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

int main(void)
{
    char c = 'A', zeich = 'B';
    cout << c + zeich;
    return 0;
}
```

Ausgabe: 131

Die Ausgabe entspricht der Summe aus den Ordnungszahlen der Zeichen 'A' (Ordnungszahl 65) und 'B' (Ordnungszahl 66) des ASCII-Codes. Beide char-Werte werden vor der Addition nach int umgewandelt, obwohl es sich hier um Operanden gleichen Datentyps handelt.

## 15.4 Explizite Typumwandlungen

Explizite Typumwandlungen werden vom Programmierer bewusst ausgelöst. Dabei kann man diese sowohl dort anbringen, wo der Compiler selbst keine Typumwandlung durchführen kann, als auch an Stellen, an denen der Compiler eine automatische – implizite – Typumwandlung vornehmen würde. Sie teilen auf diese Weise dem Compiler – und im letzteren Fall auch Programmierern, die Ihren Quellcode lesen (was in jedem Fall einem guten Programmierstil entspricht) – mit, dass die Umwandlung beabsichtigt ist und es sich nicht um ein Versehen handelt.

Zur expliziten Typumwandlung steht Ihnen zum einen der herkömmliche cast-Operator aus der Programmiersprache C zur Verfügung. Dieser ist einfach zu handhaben – Sie schreiben den gewünschten Datentyp in runde Klammern vor den zu konvertierenden Wert:

*(Datentyp) Ausdruck*

Wie angedeutet darf *Ausdruck* auch ein komplexer Ausdruck sein, der dann allerdings entsprechend geklammert werden muss. Vielfach empfiehlt sich die Klammerung allein wegen der Deutlichkeit:

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

int main(void)
{
    cout << (double) (7) / 2 * 2.0;
    return 0;
}
```

Ausgabe: 7

Hier ist die Ausgabe 7, da der Wert 7 zunächst nach `double` konvertiert wird – `cast`-Operatoren besitzen eine höhere Priorität als arithmetische Operatoren.

```
7.0 / 2
```

Der rechte Operand der Division (2) wird nunmehr nach der bekannten Regel ebenfalls nach `double` umgewandelt, dem Datentyp des linken Operanden (7.0).

```
7.0 / 2.0
```

Daher gehen bei der Division keine Nachkommastellen verloren. Andernfalls erhielte man die Ausgabe 6, da eine Integerdivision beider Operanden als 3 gewertet wird (siehe Beispiel weiter oben).

Die Klammern um das Literal 7 sind, wie gesagt, optional, hier aber wegen der Deutlichkeit zu empfehlen. Lässt man sie weg, ändert das am Ergebnis nichts:

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

int main(void)
{
    cout << (double) 7 / 2 * 2.0;
    return 0;
}
```

Ausgabe: 7

Vorsicht ist jedoch bei der expliziten Konvertierung von komplexen Ausdrücken geboten. Hier wird allein der Rückgabewert in den angegebenen Datentyp konvertiert, nicht etwa jeder einzelne Operand für sich:

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

int main(void)
{
    cout << (double) (7 / 2) * 2.0;
    return 0;
}
```

Ausgabe: 6

Im obigen Beispiel wird demnach zunächst die Integerdivision durchgeführt:

```
7 / 2
```

Erst anschließend wird der sich daraus ergebende Wert (der 3 beträgt, da die Nachkommastellen wegfallen) nach `double` konvertiert (zu 3.0). Dies wäre – wie Sie vermutlich erkannt haben – auch ohne Casting geschehen. Daher ist die Ausgabe hier 6 und nicht 7.

Neben dem `cast`-Operator des alten Stils gibt es mehrere neuere `cast`-Operatoren. Insgesamt sind es vier, von denen der `static_cast` dem herkömmlichen `cast`-Operator am nächsten kommt. Seine Anwendung ist ebenso problemlos wie die des oben gezeigten `cast`-Operators:

```
static_cast<Datentyp> (Ausdruck)
```

Der gewünschte Datentyp ist in spitzen Klammern anzugeben und *Ausdruck* muss stets geklammert werden, auch wenn es sich nur um einen einfachen Wert handelt:

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

int main(void)
{
    cout << static_cast<double> (7) / 2 * 2.0;
    return 0;
}
```

Ausgabe: 7

### Hinweis

Unter dem Gesichtspunkt, dass der `static_cast`-Operator eben zu den Neuerungen des ANSI/ISO-Standards gehört, sollte diesem der Vorzug gegeben werden. Allerdings werden wohl viele Programmierer weiterhin den alten `cast`-Operator verwenden, zumal dieser kürzer ist und damit schneller eingegeben werden kann.

# 16

## Verzweigungen

Wenn nichts anderes bestimmt ist, werden die einzelnen Anweisungen beim Programmablauf – so wie sie im Code stehen – von oben nach unten abgearbeitet. Oft ist es aber notwendig, in diesen Ablauf einzugreifen. Dies ist der Fall, wenn Anweisungen nur unter bestimmten Bedingungen, alternativ oder mehrere Male hintereinander auszuführen sind, so z.B. wenn das weitere Geschehen von einer Auswahl des Benutzers, einem bestimmten Eingabewert oder einer anderen Entscheidung abhängig gemacht werden soll.

Zur Steuerung (»Kontrolle«) des Programmflusses stehen dem Programmierer eine Reihe von Kontrollstrukturen zur Verfügung. Dazu gehören sowohl Verzweigungs- als auch Wiederholungsanweisungen. Erstere bilden das Thema dieses Kapitels, Wiederholungsanweisungen – Schleifen genannt – werden Sie in Kapitel 17 kennen lernen.

### 16.1 Logische Ausdrücke

Bevor wir uns mit den Kontrollstrukturen an sich beschäftigen, soll zunächst auf die Formulierung von Bedingungen eingegangen werden. Diesen kommt in allen Kontrollstrukturen – von der `switch`-Anweisung abgesehen – eine entscheidende Bedeutung zu.

Bedingungen im programmiersprachlichen Sinne sind logische Ausdrücke, die in Verbindung mit besagten Kontrollstrukturen auftreten. Was sind nun aber logische Ausdrücke?

Wie Sie bereits erfahren haben, wird jeder Teil einer Anweisung, der für einen Wert steht, als Ausdruck bezeichnet. Und wie Sie ebenfalls wissen, besitzt jeder Ausdruck einen Rückgabewert. Ist dieser Rückgabewert vom Datentyp `bool`, so handelt es sich um einen logischen Ausdruck. Dieser ist also ein spezieller Fall von dem, was Sie bisher gemeinhin als Ausdruck kennen gelernt haben.

#### 16.1.1 Vergleichsoperatoren

Für die Formulierung von logischen Ausdrücken stellt C++ verschiedene Vergleichsoperatoren (weitere Bezeichnung: relationale Operatoren) zur Verfügung, die Sie Tabelle 16.1 entnehmen können.

Ausdrücke, die mit Vergleichsoperatoren gebildet werden, ergeben nach ihrer Auswertung immer einen booleschen Wert, also entweder `true` (für wahr) oder `false` (für falsch).

Operator	Bedeutung
<	kleiner als
<=	kleiner als oder gleich
>	größer als
>=	größer als oder gleich
==	gleich
!=	ungleich

**Tabelle 16.1:** Vergleichsoperatoren

Vergleichsoperatoren gehören zu den binären Operatoren. Das heißt, sie erfordern einen linken und einen rechten Operanden. Diese werden in Form einer logischen Aussage einander gegenübergestellt. Über die Art des Vergleichs entscheidet dabei das verwendete Symbol.

```
1 < 2
```

Der logische Ausdruck »1 kleiner 2« ist wahr. Dementsprechend ergibt die Auswertung dieses Ausdrucks den Wert `true`, der, wie Sie wissen, intern mit 1 dargestellt wird. Auch `cout` verwendet für die Ausgabe boolescher Werte 0 bzw. 1:

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

int main(void)
{
    bool myBool = 1 < 2;
    cout << myBool;
    return 0;
}
```

Ausgabe: 1

### Hinweis

Logische Ausdrücke werden wegen ihres Datentyps auch »boolesche Ausdrücke« genannt.

Wie in allen Ausdrücken dürfen natürlich auch als Operanden eines logischen Vergleichs wiederum beliebige Ausdrücke stehen, also z.B. auch Variablenbezeichner. Um einen Ausdruck zu formulieren, der `true` ergibt, wenn eine Integervariable mit dem Bezeichner `alter` den Wert 30 hat, schreiben Sie also

```
alter == 30
```



## Achtung

Achten Sie darauf, beim logischen Vergleich auf Gleichheit unbedingt zwei Gleichheitszeichen zu verwenden (==). Andernfalls handelt es sich um eine Zuweisung.

## 16.1.2 Logische Operatoren

Folgender Ausdruck wird als `true` bewertet, falls die Variable `alter` einen Wert enthält, der größer als 20 ist, andernfalls ergibt die Auswertung `false`:

```
alter > 20
```

Wie verhält es sich aber, wenn Sie einen Ausdruck formulieren müssen, der nur dann `true` ergibt, falls der in der Variablen `alter` gespeicherte Wert sich im Bereich zwischen 20 und 80 bewegt (jeweils exklusive).

Dann muss der Wert von `alter` größer sein als 20

```
alter > 20
```

und er muss kleiner sein als 80

```
alter < 80
```

Das heißt, beide Bedingungen müssen gleichzeitig erfüllt sein. Um zwei logische Ausdrücke zu einem einzigen Gesamtausdruck zusammenzufassen, gibt es die so genannten »logischen Operatoren« (siehe Tabelle 16.2).

Operator	Bedeutung
&&	logisches Und
	logisches Oder
!	logische Negation

**Tabelle 16.2:** Logische Operatoren

Eine Verknüpfung mit `&&` (logisches Und) ergibt nur dann einen wahren Gesamtausdruck, wenn beide Teilausdrücke wahr sind. Also lautet der passende zusammengesetzte Ausdruck im obigen Fall

```
alter > 20 && alter < 80
```

Das heißt, der ganze Ausdruck ist nur dann `true`, wenn sowohl der Teilausdruck

```
alter > 20
```

als auch der Teilausdruck

```
alter < 80
```

als true gewertet wird:

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

int main(void)
{
    int alter = 39;
    bool myBool = alter > 20 && alter < 80;
    cout << myBool;
    return 0;
}
```

Ausgabe: 1

Im folgenden Beispiel gibt der Benutzer des Programms den zu überprüfenden Wert selbst ein:

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

int main(void)
{
    int alter;
    cin >> alter;
    bool myBool = alter > 20 && alter < 80;
    cout << myBool;
    return 0;
}
```

Für Eingabewerte von 21 bis 79 erhalten Sie die Ausgabe 1, andernfalls 0 (für false).

Beim logischen Oder (||) genügt es, wenn nur einer der verknüpften Teilausdrücke wahr ist. Dann ergibt die Auswertung des logischen Gesamtausdrucks den Wert true. Das logische Oder ist einschließend. Das heißt, der Ergebniswert des Gesamtausdrucks ist ebenfalls true, wenn nicht nur einer, sondern beide Teilausdrücke wahr sind.

### Tipp

Das Zeichen | wird mit **AltGr**+<img alt="pipe key icon" data-bbox="388 775 405 790"/> eingefügt.

```
2 < 1 || 12 > 5
```

Obiger Ausdruck ergibt demnach den Wert `true`, da der rechte Teilausdruck (`»12 größer 5«`) wahr ist. Ebenso `true` ergibt

```
3 > 2 || 2 > 1
```

Beide Teilausdrücke – `»3 größer 2«` und `»2 größer 1«` – sind wahr, also ist der Gesamtausdruck ebenfalls wahr.

Hier eine kleine »Wahrheitstabelle«, in der A und B für logische Teilausdrücke stehen (Tabelle 16.3).

A	B	A && B	A    B
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

**Tabelle 16.3:** »Wahrheitstabelle«

Einziger Operand der logischen Negation (!) – es handelt sich also um einen unären Operator – ist ein beliebiger logischer Ausdruck, dessen Wahrheitswert umgedreht wird. Daher sind z.B. die Ausdrücke

```
!false
```

und

```
true
```

äquivalent, beide geben den Wert `true` zurück.

Das Gleiche gilt für

```
alter != 30 // alter ungleich 30
```

und

```
!(alter == 30) // nicht alter gleich 30
```

### Achtung

Im letzten Ausdruck ist die Klammerung notwendig, da sich der Operator `!` nicht auf die Variable `alter`, sondern auf den gesamten logischen Ausdruck `alter == 30` beziehen soll.

Selbstverständlich lassen sich mit den logischen Operatoren `&&` und `||` nicht nur zwei, sondern auch mehr Teilausdrücke beliebig zusammenfassen, da zwei verknüpfte Teilausdrücke als *ein* logischer Ausdruck angesehen werden, der wiederum mit einem weiteren logischen Ausdruck verknüpft werden kann:

```
alter > 20 && alter < 80 && alter != 40
```

Obiger Ausdruck ergibt nach wie vor genau dann `true`, wenn das Alter zwischen 21 und 79 liegt (jeweils inklusive), nun aber mit einer Ausnahme – es darf nicht 40 sein (letzter Teilausdruck »alter ungleich 40«).

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

int main(void)
{
    int alter = 40;
    bool myBool = alter > 20 && alter < 80 && alter != 40;
    cout << myBool;
    return 0;
}
```

Ausgabe: 0

Der genannte logische Ausdruck ist zwar etwas komplexer, dennoch dürfte Ihnen die Interpretation kaum Schwierigkeiten bereitet haben. Sobald nur ein Teilausdruck falsch ist, trifft dies auch für den logischen Gesamtausdruck zu. Für das Ergebnis spielt es keine Rolle, welche von den beiden `&&`-Verknüpfungsoperationen zuerst ausgeführt wird.

Anders verhält es sich aber mit dem logischen Ausdruck

```
alter > 100 || alter >= 20 && alter <= 30
```

Es soll davon ausgegangen werden, dass `alter` den Wert 101 aufweist. Dann ist der erste Teilausdruck (»alter größer 100«) wahr, der zweite (»alter größer gleich 20«) ebenfalls wahr und der dritte (»alter kleiner gleich 30«) falsch.

```
true || true && false
```

Angenommen, der Ausdruck wird von links nach rechts ausgewertet:

```
true || true
```

würde als `true` bewertet und

```
true && false
```

ergibt anschließend den Wert `false` für den obigen Gesamtausdruck.

Zu einem anderen Ergebnis gelangt man, falls der rechte Teilausdruck

```
true && false
```

als Erstes ausgewertet wird. Dieser ergibt false und eine Verknüpfung

```
true || false
```

ergibt für den Gesamtausdruck den Wahrheitswert true.

Bezogen auf den Ausdruck

```
alter > 100 || alter >= 20 && alter <= 30
```

erhebt sich also die Frage, ob die Reihenfolge der Auswertung

```
(alter > 100 || alter >= 20) && alter <= 30
```

oder

```
alter > 100 || (alter >= 20 && alter <= 30)
```

entspricht. Zeit, sich über die Prioritätsrangfolge von logischen Operatoren – und in diesem Zusammenhang natürlich auch von Vergleichsoperatoren – zu unterhalten.

### Referenz

Zur Priorität von Operatoren siehe auch Kapitel 11, Abschnitt 11.5 »Priorität von Operatoren«.

## 16.1.3 Priorität von logischen und Vergleichsoperatoren

Bezüglich der Rangfolge von Vergleichs- und logischen Operatoren gilt:

- Vergleichsoperatoren besitzen eine höhere Priorität als logische Operatoren.
- Sowohl Vergleichsoperatoren als auch logische Operatoren sind linksassoziativ.
- Der Operator && (logisches Und) bindet stärker als der Operator || (logisches Oder).

Danach entspricht die Reihenfolge der Auswertung im Ausdruck

```
alter > 100 || alter >= 20 && alter <= 30
```

der Klammerung

```
alter > 100 || (alter >= 20 && alter <= 30)
```

Für den Wert 101 von alter ergibt sich somit ein »wahrer« Gesamtausdruck.

**Tipp**

Denken Sie daran, dass Sie in jedem Ausdruck durch Setzen von Klammern auf die Reihenfolge der Ausführung Einfluss nehmen können. Dies verhält sich mit logischen Ausdrücken nicht anders.

In Ergänzung zu Tabelle 11.3 sehen Sie in Tabelle 16.4 eine Zusammenstellung der bis jetzt besprochenen Operatoren nach Prioritätsstufe geordnet, wobei der am weitesten oben stehende Operator die höchste Priorität (16) besitzt. Operatoren mit übereinstimmender Prioritätskennzahl sind gleichrangig (eine vollständige Tabelle aller C++-Operatoren finden Sie im Anhang).

Priorität	Operator	Bedeutung	Assoziativität
16	<code>static_cast &lt;Typ&gt;</code>	zur Übersetzungszeit geprüfte Konvertierung	keine
15	<code>++</code>	Inkrement	keine
15	<code>--</code>	Dekrement	keine
15	<code>!</code>	logische Negation	keine
15	<code>-</code>	unäres Minuszeichen	keine
15	<code>+</code>	unäres Pluszeichen	keine
15	<code>sizeof</code>	Größe in Byte	keine
15	<code>(Typ)</code>	Cast (Typkonvertierung)	von rechts nach links
13	<code>*</code>	Multiplikation	von links nach rechts
13	<code>/</code>	Division	von links nach rechts
13	<code>%</code>	Modulo	von links nach rechts
12	<code>+</code>	Addition	von links nach rechts
12	<code>-</code>	Subtraktion	von links nach rechts
10	<code>&lt;</code>	kleiner als	von links nach rechts
10	<code>&lt;=</code>	kleiner gleich	von links nach rechts
10	<code>&gt;</code>	größer als	von links nach rechts
10	<code>&gt;=</code>	größer gleich	von links nach rechts
9	<code>==</code>	Gleichheit	von links nach rechts
9	<code>!=</code>	Ungleichheit	von links nach rechts
5	<code>&amp;&amp;</code>	logisches Und	von links nach rechts
4	<code>  </code>	logisches Oder	von links nach rechts
3	<code>? :</code>	Bedingungsoperator	von rechts nach links
2	<code>=</code>	Zuweisung	von rechts nach links
2	<code>*=</code>	Multiplikation und Zuweisung	von rechts nach links

Tabelle 16.4: Prioritätsreihenfolge der C++-Operatoren

Priorität	Operator	Bedeutung	Assoziativität
2	/=	Division und Zuweisung	von rechts nach links
2	%=	Modulo und Zuweisung	von rechts nach links
2	+=	Addition und Zuweisung	von rechts nach links
2	-=	Subtraktion und Zuweisung	von rechts nach links

Tabelle 16.4: Prioritätsreihenfolge der C++-Operatoren (Forts.)

### Referenz

Der Operator für die bedingte Zuweisung (`? :`) wird weiter unten in diesem Kapitel in Abschnitt 16.4 »Bedingungsoperator« behandelt.

## 16.2 Die if-Anweisung

Eine einfach zu handhabende und zugleich sehr effektive Möglichkeit, Programmverzweigungen einzurichten, bietet die `if`-Anweisung. Sie gestattet die Ausführung einer oder mehrerer Anweisungen in Abhängigkeit von einer Bedingung. Bei der Bedingung handelt es sich um einen logischen Ausdruck, der nach dem Schlüsselwort `if` in Klammern anzugeben ist. Die Syntax der `if`-Anweisung in ihrer einfachsten Form lautet:

```
if (Bedingung)
    Anweisung
```

bzw.

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

Falls es sich nur um eine einzelne Anweisung handelt, können Sie zwischen beiden Formen wählen. Mehrere Anweisungen müssen zu einem Block zusammengefasst werden (`{...}`).

Die Funktionsweise des `if`-Konstrukts ist schnell erklärt. Bei Eintritt in dieses während des Programmlaufs wird zunächst die Bedingung geprüft. Es gilt:

- Ist die Bedingung wahr (ergibt die Auswertung des logischen Ausdrucks den Wert `true`), dann wird (werden) die zugehörige(n) Anweisung(en) ausgeführt. Danach wird das Programm mit der nächsten Anweisung fortgesetzt, die im Code unterhalb des `if`-Konstrukts steht.
- Ist die Bedingung falsch (ergibt die Auswertung des logischen Ausdrucks den Wert `false`), so werden alle zum `if`-Konstrukt gehörigen Anweisungen ignoriert und es

wird unmittelbar mit der Anweisung fortgefahren, die im Code unterhalb des if-Konstrukts steht.

Die Ausführung der zum if-Konstrukt gehörigen Anweisung(en) ist also abhängig von der if-Bedingung. Ist die Bedingung falsch, verhält es sich daher so, als wäre das if-Konstrukt gar nicht vorhanden:

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

int main(void)
{
    int punkte;
    cout << "Wie viele Punkte hast Du? ";
    cin >> punkte;
    if (punkte > 10)
    {
        cout << "So viele!" << endl;
        cout << "Gratuliere!" << endl;
    }
    cout << "Hier geht's \"normal\" weiter" << endl;
    return 0;
}
```

Ausgabe:

```
Wie viele Punkte hast Du? 12
So viele!
Gratuliere!
Hier geht's "normal" weiter
```

bzw.

```
Wie viele Punkte hast Du? 7
Hier geht's "normal" weiter
```

Im obigen Beispiel wird der Benutzer nur dann beglückwünscht, wenn er mehr als 10 Punkte hat, andernfalls erfolgt keine Reaktion.

### Tipp

Wie bereits erwähnt, empfiehlt es sich, die Anweisungen innerhalb eines Blocks ({...}) um eine konstante Anzahl von Stellen einzurücken. Auf diese Weise wird die Struktur des Quellcodes sehr viel deutlicher. Beachten Sie, dass der if-Block einen inneren Block zum Rumpf der Funktion `main()` darstellt.



**Hinweis**

Viele Programmierer bevorzugen es, die erste geschweifte Klammer an das Ende der Zeile zu setzen, die ein bestimmtes Konstrukt einleitet:

```
...
if (punkte > 10) {
    cout << "So viele!" << endl;
    cout << "Gratuliere!" << endl;
}
...
```

Gelegentlich werden auch wir uns dieser Schreibweise bedienen.

Falls es sich um genau eine bedingungsabhängige Anweisung handelt, besteht keine syntaktische Notwendigkeit, diese in {} einzuschließen:

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

int main(void)
{
    int punkte;
    cout << "Wie viele Punkte hast Du? ";
    cin >> punkte;
    if (punkte > 10)
        cout << "So viele!" << endl
            << "Gratuliere!" << endl;
    cout << "Hier geht's \"normal\" weiter" << endl;
    return 0;
}
```

Ausgabe wie zuvor.

**Hinweis**

Lassen Sie sich nicht irritieren. Obwohl auf zwei Zeilen verteilt, handelt es sich natürlich um nur eine bedingungsabhängige Anweisung:

```
cout << "So viele!" << endl
    << "Gratuliere!" << endl;
```

## Tipp

Auch wenn nur eine – nicht geklammerte – Anweisung zum `if`-Konstrukt vorhanden ist, sollten Sie diese einrücken. Man erkennt dann besser, dass die Anweisung zur Kontrollstruktur gehört:

```
if (Bedingung)
    Anweisung // bedingungsabhängig
Anweisung // nicht bedingungsabhängig
```

Ferner sei angemerkt, dass das Setzen von Klammern in jedem Fall davor schützt, dies nachträglich zu vergessen, falls bei eventuellen Änderungen weitere bedingungsabhängige Anweisungen hinzukommen.

Der Compiler rechnet automatisch die unmittelbar folgende Anweisung zum `if`-Konstrukt, sofern nicht – wie weiter oben der Fall – mehrere Anweisungen zu einem Block zusammengefasst sind. Dies führt zu unerwünschten Ergebnissen, falls der Programmierer versehentlich hinter die `if`-Bedingung ein Semikolon setzt:

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

Ein Semikolon allein wird vom Compiler ebenfalls als vollständige Anweisung angesehen (siehe dazu Kapitel 7, Abschnitt 7.2 »Anweisungsende«). Deshalb rechnet er nun dieses zur Kontrollstruktur. Alle folgenden Anweisungen werden daher in jedem Fall ausgeführt, unabhängig von der `if`-Bedingung – was vom Programmierer vermutlich nicht so beabsichtigt war:

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

int main(void)
{
    int punkte;
    cout << "Wie viele Punkte hast Du? ";
    cin >> punkte;
    if (punkte > 10); // ???
    {
        cout << "So viele!" << endl;
        cout << "Gratuliere!" << endl;
    }
    cout << "Hier geht's \"normal\" weiter" << endl;
    return 0;
}
```

Ausgabe:

```
Wie viele Punkte hast Du? 0
So viele!
Gratuliere!
Hier geht's "normal" weiter
```

## 16.2.1 Verschachteln von Kontrollstrukturen

Man spricht bei Kontrollstrukturen von Anweisungen (z.B. if-Anweisung und switch-Anweisung), nachdem sie im Code überall dort eingesetzt werden dürfen, wo auch eine einfache Anweisung stehen darf. So gesehen werden Kontrollstrukturen vom Compiler tatsächlich wie einfache Anweisungen behandelt. Nun enthält jede Kontrollstruktur selbst wiederum eine oder mehrere Anweisungen. Daraus folgt, dass Kontrollstrukturen untereinander beliebig verschachtelt werden können:

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

Wie gesagt, die Möglichkeit der Verschachtelung gilt für alle Kontrollstrukturen und ist auch nicht auf ein und dieselbe Kontrollstruktur beschränkt.

### Hinweis

Allgemein gilt zwischen einzelnen Anweisungen und Anweisungsblöcken folgende Beziehung: An jeder Stelle im Code, an der eine einfache Anweisung verwendet werden kann, darf auch ein Anweisungsblock stehen. Daraus ergibt sich, dass Blöcke grundsätzlich verschachtelt werden dürfen.

Dazu ein kleines Beispiel unter Anwendung des Modulo-Operators (zu diesem Operator siehe Kapitel 11, Abschnitt 11.2.1 »Der Modulo-Operator«).

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

int main(void)
{
    int zahl;
    cout << "Gib eine Zahl ein: ";
```

```

cin >> zahl;
if (zahl > 0)
{
    cout << "Die Zahl ist positiv";
    if (zahl % 2 == 0)
    {
        cout << " und gerade";
    }
    cout << '.' << endl;
}
cout << "Tschau" << endl;
return 0;
}

```

Ausgabe:

```

Gib eine Zahl ein: -32
Tschau

```

bzw.

```

Gib eine Zahl ein: 99
Die Zahl ist positiv.
Tschau

```

bzw.

```

Gib eine Zahl ein: 100
Die Zahl ist positiv und gerade.
Tschau

```

Eine Zahl ist gerade, wenn sie durch 2 ohne Rest teilbar ist. Um im Programm festzustellen, ob die zuvor vom Benutzer eingegebene Zahl gerade ist, ist es daher ausreichend, den Ausdruck `zahl % 2 == 0` zu vergleichen:

```

zahl % 2 == 0

```

Ist diese Bedingung wahr, so ist der in `zahl` gespeicherte Wert gerade, und die Ausgabe wird mit der Anweisung

```

cout << "und gerade";

```

um den Text

```

und gerade

```

ergänzt. Das Programm führt diesen Vergleich ausschließlich für positive Zahlen durch, da der Programmlauf bei Eingabe von negativen Zahlen die innere if-Anweisung nicht erreicht.

## 16.2.2 Konvertierung in logischen Ausdrücken

Wie weiter oben festgestellt, liefern logische Ausdrücke immer einen Wahrheitswert zurück: `true` oder `false`. Zudem gilt, dass es sich bei Bedingungen stets um logische Ausdrücke handeln muss. Dennoch ist es in C++ erlaubt, Bedingungen zu formulieren, die einen numerischen Wert ergeben. Dies ist darin begründet, dass C++ in Bedingungen gegebenenfalls eine implizite Konvertierung in den booleschen Datentyp durchführt, wobei der Wert 0 zu `false`, alle anderen numerischen Werte zu `true` konvertiert werden.

Statt

```
if (zahl != 0)
```

kann man unter Ausnutzung des beschriebenen Konvertierungsverhaltens auch kürzer schreiben:

```
if (zahl)
```

Ebenso lässt sich die Bedingung

```
if (zahl == 0)
```

auch als

```
if (!zahl)
```

formulieren.

### Hinweis

Ob dies nun einem guten Programmierstil entgegensteht, sei dahingestellt. Vor allem unter C-Programmierern war und ist es weit verbreitet, von der kürzeren Notation Gebrauch zu machen. Dem Programmierneuling sei auf jeden Fall empfohlen, Bedingungen auszuformulieren.

Vorsicht ist in diesem Zusammenhang beim Vergleich von Variablenwerten auf Gleichheit geboten:

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

```
int main(void)
{
    int z;
    cin >> z;
    if (z == 8)
        cout << "Die Acht!";
    return 0;
}
```

Bei Eingabe des Wertes 8 soll das Programm den Text

```
Die Acht!
```

ausgeben. Für alle anderen Eingabewerte soll nichts geschehen, was in Bezug auf obiges Listing auch der Fall ist.

Falls der Programmierer jedoch in der Bedingung versehentlich ein Gleichheitszeichen vergisst, handelt es sich eben nicht um einen logischen Vergleich, sondern um eine Zuweisung:

```
if (z = 8)
```

Das Schlimme daran ist, dass dieses Versehen nicht etwa eine Fehlermeldung zur Folge hat. Vielmehr wird die Zuweisung durchgeführt, die Variable `z` erhält den Wert 8 und anschließend erfolgt die Konvertierung in den booleschen Datentyp gemäß der oben genannten Regel. Und da 8 eben eine Zahl ungleich 0 ist, ist diese Bedingung immer wahr:

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

int main(void)
{
    int z;
    cin >> z;
    if (z = 8)
        cout << "Die Acht!";
    return 0;
}
```

Ausgabe:

```
11
Die Acht!
```

**Achtung**

Die Verwechslung von = und == ist einer der häufigsten und teilweise tückischsten Fehler, da er vom Compiler in den meisten Fällen nicht als syntaktischer Fehler erkannt wird (wie es oben der Fall ist).

Nur wenn links vom Gleichheitszeichen ein Literal oder eine const-Konstante steht, wird der Fehler vom Compiler moniert, da einer solchen natürlich kein Wert zugewiesen werden kann.

**Tipp**

Manche Programmierer bevorzugen daher die umgekehrte Schreibweise, das Literal in diesem Fall links vom »==«-Operator:

```
if (8 == z) // entspricht z == 8
    cout << "Die Acht!";
```

Die Wirkung bei diesem Vergleich ist dieselbe. Schreibt man jedoch versehentlich ein Gleichheitszeichen, moniert der Compiler einen syntaktischen Fehler. Da sich das Programm gar nicht erst kompilieren lässt, findet man den Fehler erheblich leichter:

```
if (8 = z) // SYNTAKTISCHER FEHLER!
    cout << "Die Acht!";
```

### 16.2.3 if else

Bei Bedarf kann man das if-Konstrukt um einen else-Zweig erweitern. Dieser besteht aus dem Schlüsselwort else und einem zugehörigen Anweisungsteil, der nur dann ausgeführt wird, wenn die if-Bedingung den Wert false ergibt. Die Syntax lautet:

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

Dabei kann es sich jeweils um eine beliebige Anzahl von Anweisungen handeln. Ist nur eine Anweisung zum else-Zweig zugehörig, ist das Setzen von Klammern optional. Das Gleiche gilt natürlich für den if-Zweig.

Ist die if-Bedingung wahr, werden die Anweisungen des if-Zweiges ausgeführt, andernfalls die Anweisungen des else-Zweiges. Es kommt also – alternativ – in jedem Fall ein Anweisungsteil zum Zuge:

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

int main(void)
{
    int zahl;
    cout << "Gib eine Zahl ein: ";
    cin >> zahl;
    if (zahl % 2 == 0)
        cout << "Die Zahl ist gerade.";
    else
        cout << "Die Zahl ist ungerade.";
    cout << endl;
    return 0;
}
```

Ausgabe:

```
Gib eine Zahl ein: 12
Die Zahl ist gerade.
```

bzw.

```
Gib eine Zahl ein: 9
Die Zahl ist ungerade.
```

Beachten Sie:

- Das Konstrukt `if else` wird wie die einfache `if`-Anweisung vom Compiler als Einheit, gewissermaßen wie eine Anweisung, behandelt (siehe weiter oben).
- Auch im `else`-Zweig eines `if else`-Konstrukts ist eine Schachtelung von weiteren Kontrollstrukturen möglich.
- Ein `else` darf niemals alleine (ohne zugehöriges `if`) im Code stehen.
- Durch Einrücken der Anweisungen innerhalb eines Blocks wird deutlich, welcher `else`-Teil zu welchem `if` gehört.

### Hinweis

Wenn Sie einen Quellcode lesen müssen, bei dem die Strukturen nicht durch Einrücken erkennbar sind, verfahren Sie nach folgender Regel:

Ein `else` gehört jeweils zum letzten `if` ohne `else`.

Gehen Sie im Code von oben nach unten, bis Sie auf das erste `else` treffen. Dieses rechnen Sie zum letzten `if`. Dann gehen Sie weiter, bis Sie auf das nächste `else` treffen und rechnen dieses zum letzten `if`, das noch ohne `else` ist usw.



## 16.2.4 Stringvergleiche

Die Vergleichsoperatoren `==` und `!=` lassen sich auch auf Strings anwenden. Damit steht Ihnen eine einfache Möglichkeit zum Vergleich von Zeichenketten bzw. Stringvariablen zur Verfügung:

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

int main(void)
{
    string vorname;
    cout << "Sag mir Deinen Vornamen: ";
    cin >> vorname;
    if (vorname == "Walter")
        cout << "Du schon wieder!" << endl;
    else
        cout << "Willkommen!" << endl;
    // ...
    return 0;
}
```

Ausgabe:

```
Sag mir Deinen Vornamen: Walter
Du schon wieder!
```

bzw.

```
Sag mir Deinen Vornamen: Hans
Willkommen!
```

## 16.2.5 else if

Oft ist es notwendig, auf mehrere Bedingungen zu testen. Dies kann geschehen, indem man mehrere `if`-Anweisungen im Code hintereinander setzt. Dabei kann es durchaus erwünscht sein, dass für bestimmte Fälle mehrere Anweisungsteile zur Ausführung gelangen:

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

int main(void)
{
    int zahl;
    cout << "Gib mir eine Zahl: ";
```

```

cin >> zahl;
if (zahl > 0)
{
    cout << "Die Zahl ist positiv." << endl;
}
if (zahl % 2 == 0)
{
    cout << "Die Zahl ist gerade." << endl;
}
return 0;
}

```

Ausgabe:

```

Gib mir eine Zahl: 8
Die Zahl ist positiv.
Die Zahl ist gerade.

```

In anderen Fällen erfordert es die Programmlogik, dass sich die einzelnen Bedingungen gegenseitig ausschließen. Dann ist bei der Formulierung der if-Bedingungen besondere Aufmerksamkeit geboten:

```

#include <iostream>
using namespace std;

int main(void)
{
    int punkte;
    cout << "Wie viele Punkte hast Du? ";
    cin >> punkte;
    if (punkte < 5)
    {
        cout << "So wenig! - Tut mir Leid!";
    }
    if (punkte >= 5 && punkte <= 10)
    {
        cout << "Ist doch ganz gut!";
    }
    if (punkte > 10)
    {
        cout << "So viele! - Gratuliere!";
    }
    cout << endl;
    return 0;
}

```

Ausgabe:

```
Wie viele Punkte hast Du? 7
Ist doch ganz gut!
```

So ist für die Bedingung der zweiten if-Anweisung eine logische Und-Verknüpfung notwendig.

```
punkte >= 5 && punkte <= 10
```

Die Anweisung

```
cout << "Ist doch ganz gut!";
```

soll ja nur für eine Punktzahl, die im Bereich zwischen 5 und 10 (inklusive) liegt, ausgeführt werden. Lautet die Bedingung dagegen `punkte >= 5`, dann kommt für eine Punktzahl, die über 10 liegt, sowohl diese Anweisung als auch die Anweisung des letzten if-Konstrukts zur Ausführung:

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

int main(void)
{
    int punkte;
    cout << "Wie viele Punkte hast Du? ";
    cin >> punkte;
    if (punkte < 5)
    {
        cout << "So wenig! - Tut mir Leid!";
    }
    if (punkte >= 5) // ???
    {
        cout << "Ist doch ganz gut!";
    }
    if (punkte > 10)
    {
        cout << "So viele! - Gratuliere!";
    }
    cout << endl;
    return 0;
}
```

Ausgabe:

```
Wie viele Punkte hast Du? 11
Ist doch ganz gut!So viele! - Gratuliere!
```

Falls sich mehrere Bedingungen gegenseitig ausschließen sollen, lassen sich die Verzweigungen auch sehr elegant mit einer `else if`-Konstruktion erzeugen. Die entsprechende Syntax lautet:

```
if (Bedingung_1)
{
    Anweisung(en)
}
else if (Bedingung_2)
{
    Anweisung(en)
}
...
else if (Bedingung_n)
{
    Anweisung(en)
}
```

Sobald entweder der Anweisungsteil des `if`-Zweiges oder einer der `else if`-Zweige zur Ausführung gelangt – weil die entsprechende Bedingung wahr ist –, landet die Programmausführung außerhalb des ganzen Konstrukts. Es ist also sichergestellt, dass nicht mehr als ein Anweisungsteil abgearbeitet wird.

### Hinweis

Im Grunde genommen handelt es sich bei dieser Konstruktion um nichts anderes als um eine Verschachtelung von `if else`-Zweigen:

```
if (Bedingung)
{
    Anweisung(en)
}
else
    if (Bedingung 2)
    {
        Anweisung(en)
    }
    else
        if (Bedingung n)
        {
            Anweisung(en)
        }
    ...
```

Denken Sie daran, dass ein `if else`-Konstrukt gewissermaßen als eine einzige Anweisung anzusehen ist. Da hier jedem `else` nur eine einzige (`if else`-)Anweisung folgt, ist das Setzen von Klammern (`{}`) für die jeweiligen `else`-Zweige nicht erforderlich.

Hier die entsprechende Konstruktion auf unser obiges Beispiel angewendet. Für die zweite Bedingung ist der logische Ausdruck `punkte >= 5` nunmehr ausreichend:

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

int main(void)
{
    int punkte;
    cout << "Wie viele Punkte hast Du? ";
    cin >> punkte;
    if (punkte > 10)
    {
        cout << "So viele! - Gratuliere!";
    }
    else if (punkte >= 5)
    {
        cout << "Ist doch ganz gut! ";
    }
    else if (punkte < 5)
    {
        cout << "So wenig! - Tut mir Leid! ";
    }
    cout << endl;
    return 0;
}
```

Ausgabe:

```
Wie viele Punkte hast Du? 15
So viele! - Gratuliere!
```

### Hinweis

Beachten Sie, dass es bei `else if`-Anweisungen auf die Reihenfolge der einzelnen Bedingungen ankommen kann. Oben wird als Erstes auf `punkte > 10` und als Zweites auf `punkte >= 5` überprüft. Würde dagegen die ursprüngliche Reihenfolge verwendet, nämlich dass der Vergleich `punkte >= 5` vor dem von `punkte > 10` erfolgen würde, träfe `punkte >= 5` auch auf einen Punktestand von über 10 zu und die dann dritte Bedingung `punkte > 10` würde nicht mehr abgefragt. Man erhielte dann grundsätzlich ab einem Punktestand von 5 (inklusive) die Ausgabe `Ist doch ganz gut!`, aber niemals den Bewertungstext `So viele! - Gratuliere!`. Es ist also bei solchen Abfragen zu beachten, dass sie von oben nach unten abgearbeitet werden; man darf daher keine Fälle »aussortieren«, die man später noch abfragen möchte.

Selbstverständlich hat es keine negativen Auswirkungen, wenn Sie für die entsprechende Bedingung dennoch

```
if (punkte >= 5 && punkte <= 10)
```

formulieren.

Da es sich bei dieser Konstruktion eben eigentlich um eine Verschachtelung von `if else`-Zweigen handelt (siehe Hinweis weiter oben), lässt sich bei Bedarf auch noch ein abschließender `else`-Zweig hinzufügen:

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

int main(void)
{
    int note;
    cout << "Welche Note hast Du bekommen? ";
    cin >> note;
    if (note == 1)
    {
        cout << "Sehr gut!";
    }
    else if (note == 2)
    {
        cout << "Gut!";
    }
    else if (note == 3)
    {
        cout << "In Ordnung";
    }
    else if (note == 4)
    {
        cout << "So lala";
    }
    else if (note == 5)
    {
        cout << "Nicht gut!";
    }
    else if (note == 6)
    {
        cout << "Schlecht!";
    }
    else
    {
        cout << "Diese Note gibt es nicht!";
    }
}
```

```
    cout << endl;  
    return 0;  
}
```

Ausgabe:

```
Welche Note hast Du bekommen? 2  
Gut!
```

Der abschließende `else`-Zweig wird immer dann ausgeführt, wenn dies für keinen der übrigen Zweige zutrifft:

```
Welche Note hast Du bekommen? 9  
Diese Note gibt es nicht!
```

## 16.3 Die switch-Anweisung

Speziell zur Implementation von Mehrfachverzweigungen bietet C++ als weitere Möglichkeit die `switch`-Anweisung. Zwar können Sie jede Aufgabenstellung auch unter Anwendung von `if`-Anweisungen bzw. `if else`-Konstrukten umsetzen. Für den Benutzer Ihres Programms macht das keinen Unterschied. Dennoch besitzt die `switch`-Anweisung ihre Existenzberechtigung, da diese in gewisser Weise geradlinig und gut überschaubar ist.

### Hinweis

Viele Kontrollstrukturen sind letzten Endes untereinander austauschbar – was insbesondere für Schleifen gilt (zu Schleifen siehe Kapitel 17 »Wiederholungsanweisungen«). Lassen Sie sich von dieser Tatsache nicht verwirren. Wie an anderer Stelle bereits erwähnt, gibt es in der Programmierung in den seltensten Fällen nur eine Lösung für ein Problem.

Die `switch`-Anweisung eignet sich besonders dann, wenn mehrere Anweisungsteile alternativ ausgeführt werden sollen. Von daher entspricht sie weitgehend der im letzten Abschnitt vorgestellten `else if`-Konstruktion. Es erfolgt jedoch keine Bedingungsprüfung, sondern ein direkter Wertevergleich, was den Anwendungsbereich der `switch`-Anweisung etwas einschränkt.

Die Syntax lautet

```
switch (Ausdruck)  
{  
    case Konstante_1:  
        Anweisung(en)  
        break;
```

```

    case Konstante_2:
        Anweisung(en)
        break;
    ...
    case Konstante_n:
        Anweisung(en)
        break;
    default:
        Anweisung(en)
}

```

Der dem einleitenden Schlüsselwort `switch` folgende Ausdruck muss in runde Klammern gesetzt werden. Grundsätzlich darf es sich dabei um einen beliebigen Ausdruck handeln, mit der Einschränkung, dass dieser einen ganzzahligen Datentyp besitzen muss. Meist wird dort der Bezeichner einer Variablen vom Datentyp `int` oder `char` stehen.

Die einzelnen Programmzweige werden jeweils mit dem Schlüsselwort `case` und der Angabe eines konstanten Wertes, gefolgt von einem Doppelpunkt, eingeleitet.

### Hinweis

Demnach darf in den `case`-Marken kein Variablenname angegeben werden, wohl aber der Bezeichner einer `const`-Konstanten.

Die `case`-Konstante muss vom selben Datentyp sein wie der oben nach dem Schlüsselwort `switch` angegebene Ausdruck. Daraus folgt, dass auch für die `case`-Konstanten ausschließlich ganzzahlige Datentypen in Frage kommen. Gleitkommawerte sowie Strings sind nicht erlaubt.

Nach dem Doppelpunkt schließt sich der Anweisungsteil des entsprechenden `case`-Zweiges an. Eine Klammerung ist hier nicht erforderlich, da die `case`-Marken bzw. die `default`-Marke gleichfalls als Begrenzer zwischen den einzelnen Anweisungsteilen fungieren (für den Anweisungsteil des letzten Zweiges gilt natürlich die schließende geschweifte Klammer des `switch`-Konstrukts als unterer Begrenzer).

Bei Eintritt in die `switch`-Anweisung während des Programmlaufs wird zunächst der Ausdruck ausgewertet. Der Rückgabewert wird der Reihe nach mit den einzelnen `case`-Konstanten verglichen. Sobald eine Übereinstimmung auftritt, wird der zugehörige Programmzweig ausgeführt.

Der `default`-Zweig ist optional. Ist er vorhanden, so werden die Anweisungen dieses Zweiges ausgeführt, falls keine der `case`-Konstanten mit dem Rückgabewert des `switch`-Ausdrucks übereinstimmt. Ansonsten wird in diesem Fall die `switch`-Anweisung ohne Aktion verlassen, und die Programmausführung setzt mit der nächsten Anweisung nach dem Konstrukt fort.



Hier das letzte Beispiel, jetzt mit der `switch`-Anweisung umgesetzt:

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

int main(void)
{
    int note;
    cout << "Welche Note hast Du bekommen? ";
    cin >> note;
    switch (note)
    {
        case 1:
            cout << "Sehr gut!";
            break;
        case 2:
            cout << "Gut!";
            break;
        case 3:
            cout << "In Ordnung";
            break;
        case 4:
            cout << "So lala";
            break;
        case 5:
            cout << "Nicht gut!";
            break;
        case 6:
            cout << "Schlecht!";
            break;
        default:
            cout << "Diese Note gibt es nicht!";
    }
    cout << endl;
    return 0;
}
```

Ausgabe:

```
Welche Note hast Du bekommen? 1
Sehr gut!
```

Vermutlich fragen Sie sich, was es mit den `break`-Anweisungen (`break;`) auf sich hat. Dazu Folgendes: Die Stelle vom Schlüsselwort `case` bis zum Doppelpunkt (`case Konstante:`) definiert eine so genannte Sprungmarke. Dabei handelt es sich um eine Position im Quellcode, die benannt ist. Diese Marke wird angesprungen, wenn der Wert der Konstanten mit dem Ergebniswert des `switch`-Ausdrucks übereinstimmt. Sobald ein Sprung-

ziel erreicht ist, werden alle folgenden Anweisungen bis zum Ende des switch-Konstrukts abgearbeitet, weitere case-Marken praktisch ignoriert:

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

int main(void)
{
    int note;
    cout << "Welche Note hast Du bekommen? ";
    cin >> note;
    switch (note)
    {
        case 1:
            cout << "Sehr gut!";
        case 2:
            cout << "Gut!";
        case 3:
            cout << "In Ordnung";
        case 4:
            cout << "So lala";
        case 5:
            cout << "Nicht gut!";
        case 6:
            cout << "Schlecht!";
        default:
            cout << "Diese Note gibt es nicht!";
    }
    cout << endl;
    return 0;
}
```

Ausgabe:

```
Welche Note hast Du bekommen? 3
In OrdnungSo lalaNicht gut!Schlecht!Diese Note gibt es nicht!
```

Die Anweisung

```
break;
```

bewirkt nun, dass ein switch-Konstrukt sofort verlassen wird, was im obigen Fall von der Logik her jeweils am Ende eines case-Zweiges erwünscht ist.

Sie dürfen die `break`-Anweisung übrigens auch an anderen Stellen des `switch`-Konstrukts einsetzen, etwa in inneren `if else`-Zweigen. Sie muss nicht zwangsläufig die am weitesten unten stehende Anweisung innerhalb eines Zweiges sein.

```
...
int note, letzteNote;
...
switch (note)
{
...
    case 3:
        if (letzteNote > 3)
        {
            cout << "Immer besser!";
            break;
        }
        else
        {
            cout << "In Ordnung";
            break;
        }
    case 4:
...
}
```

### Achtung

Beachten Sie, dass die Verwendung der `break`-Anweisung ausschließlich im Zusammenhang mit der `switch`-Anweisung sowie mit Schleifen erlaubt ist (zu Schleifen siehe Kapitel 17 »Wiederholungsanweisungen«). Im obigen Codestück steht die `break`-Anweisung zwar innerhalb eines `if else`-Konstrukts, sie bezieht sich aber auf die äußere `switch`-Anweisung.

Wie am obigen Beispiel zu erkennen, darf die `switch`-Anweisung – wie jede andere Kontrollstruktur – beliebig verschachtelt werden. Das heißt auch, dass jeder `case`-Zweig – und natürlich auch der `default`-Zweig – wiederum weitere `switch`-Anweisungen enthalten darf.

### Hinweis

Über die Verschachtelungstiefe brauchen Sie sich keine Gedanken zu machen. Falls Ihr Compiler diesbezüglich eine Grenze setzt, liegt diese nicht unter 256.

Des Weiteren müssen die einzelnen case-Konstanten innerhalb der switch-Anweisung keine besondere Reihenfolge haben. Es ist also nicht erforderlich, diese aufsteigend anzugeben:

```
...
switch (note)
{
    case 5:
        cout << "Nicht gut!";
        break;
    case 2:
        cout << "Gut!";
        break;
    case 4:
        cout << "So lala";
        break;
...

```

Ferner sei darauf hingewiesen, dass in den einzelnen case-Zweigen – wie grundsätzlich in jedem Block – beliebig viele Anweisungen stehen dürfen.

Beachten Sie aber, dass die Verwendung der switch-Anweisung folgenden Beschränkungen unterworfen ist:

- Sie ist nur auf ganzzahlige Datentypen anwendbar. Der Vergleich von Gleitkommawerten oder Zeichenketten ist nicht erlaubt.
- Es sind keine Bereichsprüfungen möglich (z.B. `note > 3`).

Für die genannten Fälle bleibt Ihnen nichts anderes übrig, als auf die `if`- bzw. `if else`-Konstrukte zurückzugreifen.

Häufig wird die switch-Anweisung eingesetzt, um ein Auswahlménú zu realisieren. Der Anwender erhält zunächst ein Ménú und entscheidet dann durch Eingabe einer Zahl oder eines Zeichens, welche Aktion ausgeführt werden soll.

Als Beispiel hierzu ein kleines Rechenprogramm. Der Benutzer soll zwei Zahlen eingeben und dann entscheiden, welche Rechenoperation mit diesen Zahlen stattfindet:

### CD-ROM

Die endgültige Version des Programms finden Sie als *k16p* im Ordner *Beispiele/K16* auf der Buch-CD.

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

```
int main(void)
{
    double zahl1, zahl2;
    char auswahl;
    cout << "Geben Sie 2 Zahlen ein: ";
    cin >> zahl1 >> zahl2;
    cout << "\n(A)ddition" << endl;
    cout << "(S)ubtraktion" << endl;
    cout << "(M)ultiplikation" << endl;
    cout << "(D)ivision\n\n";
    cout << "Auswahl: ";
    cin >> auswahl;
    // ...
    cout << endl;
    return 0;
}
```

Ausgabe:

```
Geben Sie 2 Zahlen ein: 17.9 29.11
```

```
(A)ddition
(S)ubtraktion
(M)ultiplikation
(D)ivision
```

```
Auswahl: A
```

Die Anfangsbuchstaben der zur Verfügung stehenden Rechenoperationen sind in Klammern gesetzt, um dem Benutzer deutlich zu machen, wie er zwischen diesen wählen kann. So soll er sich z.B. durch Drücken der Taste **A** für die Addition entscheiden können. Ober- und unterhalb des Menüs wurde eine zusätzliche Leerzeile eingefügt (`\n`). Die Variable `auswahl` dient dazu, die Benutzerauswahl festzuhalten. Als Datentyp wird `char` eingesetzt, da es sich bei der entsprechenden Eingabe um ein Zeichen handelt.

Nun muss sich das Programm in Abhängigkeit von der Benutzerauswahl verzweigen. Zur Implementation der verschiedenen Programmzweige eignet sich die `switch`-Anweisung mit `auswahl` als Prüfvariable:

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

int main(void)
{
    double zahl1, zahl2;
    char auswahl;
    cout << "Geben Sie 2 Zahlen ein: ";
```

```

cin >> zahl1 >> zahl2;
cout << "\n(A)ddition" << endl;
cout << "(S)ubtraktion" << endl;
cout << "(M)ultiplikation" << endl;
cout << "(D)ivision\n\n";
cout << "Auswahl: ";
cin >> auswahl;
cout << endl;
switch (auswahl)
{
    case 'A':
        cout << zahl1 << " + " << zahl2
            << " = " << zahl1 + zahl2;
        break;
    case 'S':
        cout << zahl1 << " - " << zahl2
            << " = " << zahl1 - zahl2;
        break;
    case 'M':
        cout << zahl1 << " * " << zahl2
            << " = " << zahl1 * zahl2;
        break;
    case 'D':
        cout << zahl1 << " / " << zahl2
            << " = " << zahl1 / zahl2;
    }
    cout << endl;
    return 0;
}

```

Beachten Sie, dass für den letzten case-Zweig kein break-Befehl erforderlich ist, da das switch-Konstrukt hier ohnehin endet. Falls Sie jedoch für eine ungültige Auswahl einen default-Zweig einrichten, sollten Sie das nachträgliche Setzen der break-Anweisung in diesem case-Zweig nicht vergessen:

```

...
case 'D':
    cout << zahl1 << " / " << zahl2
        << " = " << zahl1 / zahl2;
    break;
default:
    cout << "Ung\u00fcltige Auswahl (A, S, M oder D)";
} // Ende von switch
...

```

Die Ausgabe des obigen Programms für die Addition der Eingabezahlen 37.31 und 73.69 ist Abbildung 16.1 zu entnehmen.

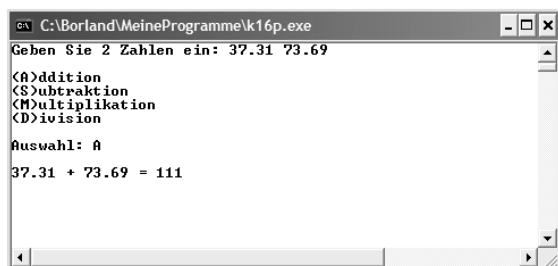


Abbildung 16.1: Rechenprogramm

Achten Sie bitte nach dem Start des Programms darauf, bei der Auswahl für die Rechenoperation den entsprechenden Großbuchstaben (z.B. A) einzugeben, da nur auf diesen überprüft wird.

Genau hier liegt ein Verbesserungspotenzial. Dabei bietet es sich an, dem Benutzer die Auswahl auch mit den entsprechenden Kleinbuchstaben zu gestatten. Es sollen also für zwei Werte (Groß- und Kleinbuchstabe) die gleichen Aktionen durchgeführt werden, z.B. Addition bei Eingabe von A und a. Solche Fälle – dass bei zwei oder mehr Werten dasselbe geschehen soll – kommen in der Praxis häufig vor.

Wir hatten oben dargelegt, dass, nachdem ein Sprungziel erreicht ist, mit der Programmausführung so lange sequenziell weitergemacht wird, bis das nächste break; oder das Ende der switch-Anweisung erreicht ist – ungeachtet, ob dabei die Anweisungen mehrerer case-Zweige ausgeführt werden. Dieses Verhalten lässt sich ausnutzen, um für zwei oder mehr case-Marken denselben Anweisungsteil zur Ausführung zu bringen. Man setzt die entsprechenden case-Marken einfach untereinander, wobei nur die letzte case-Marke einen Anweisungsteil besitzt.

Wenn Sie im obigen Rechenprogramm erreichen möchten, dass die Addition sowohl für die Eingabe A als auch für die Eingabe a durchgeführt wird, dann schreiben Sie:

```
...
case 'a':
case 'A':
    cout << zahl1 << " + " << zahl2
        << " = " << zahl1 + zahl2;
    break;
...
```

Hier das entsprechend erweiterte Listing:

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

int main(void)
{
    double zahl1, zahl2;
    char auswahl;
```

```

cout << "Geben Sie 2 Zahlen ein: ";
cin >> zahl1 >> zahl2;
cout << "\n(A)ddition" << endl;
cout << "(S)ubtraktion" << endl;
cout << "(M)ultiplikation" << endl;
cout << "(D)ivision\n\n";
cout << "Auswahl: ";
cin >> auswahl;
cout << endl;
switch (auswahl)
{
    case 'a':
    case 'A':
        cout << zahl1 << " + " << zahl2
            << " = " << zahl1 + zahl2;
        break;
    case 's':
    case 'S':
        cout << zahl1 << " - " << zahl2
            << " = " << zahl1 - zahl2;
        break;
    case 'm':
    case 'M':
        cout << zahl1 << " * " << zahl2
            << " = " << zahl1 * zahl2;
        break;
    case 'd':
    case 'D':
        cout << zahl1 << " / " << zahl2
            << " = " << zahl1 / zahl2;
    }
    cout << endl;
    return 0;
}

```

Ausgabe:

Geben Sie 2 Zahlen ein: 17 39

(A)ddition  
(S)ubtraktion  
(M)ultiplikation  
(D)ivision

Auswahl: m

17 \* 39 = 663



Beachten Sie, dass es sich trotz mehrerer case-Marken um jeweils nur einen Programmzweig handelt – case-Marken dienen als Sprungziel, ansonsten werden sie ignoriert.

Enthält die Variable `auswahl` also den Wert `'m'` (weil der Benutzer bei der Auswahl ein kleines »m« eingegeben hat), so wird die zugehörige case-Marke (`case 'm':`) angesprungen und der nächstliegende Anweisungsteil ausgeführt. Dieser beginnt hier unmittelbar nach der Marke `case 'M':`. Die Programmausführung »rutscht« gewissermaßen vom Sprungziel bis zum nächsten `break`; bzw. bis zum Ende der `switch`-Anweisung durch. Etwaige weitere case-Marken, auf die getroffen wird, werden dabei nicht beachtet.

## 16.4 Bedingungsoperator

Der Bedingungsoperator (`?:`) arbeitet ähnlich wie die `if else`-Anweisung, hat aber gegenüber dieser einige Vorteile. So lassen sich Abfragen kürzer schreiben und es ist eine Einbettung in bestimmte Konstruktionen möglich, z.B. in Funktionsaufrufe, ohne dass zusätzliche (Hilfs-)Variablen eingeführt werden müssen.

Beim Bedingungsoperator handelt es sich um den einzigen tertiären Operator der Programmiersprache C++. Das heißt, er besitzt drei Operanden:

```
Ausdruck_1 ? Ausdruck_2 : Ausdruck_3
```

Der am weitesten links stehende Operand (*Ausdruck1*) ist ein logischer Ausdruck. Dieser wird ausgewertet, woraufhin entschieden wird, welchen Rückgabewert der gesamte, mit dem Bedingungsoperator gebildete Ausdruck besitzt. Ist *Ausdruck1* wahr, so entspricht der Wert des Gesamtausdrucks dem Rückgabewert von *Ausdruck2*, andernfalls dem von *Ausdruck3*. Mit anderen Worten: Ist *Ausdruck1* `true`, wird *Ausdruck2* ausgewertet, ansonsten *Ausdruck3*.

Man kann den ganzen Ausdruck demzufolge lesen als

*wenn zutreffend ? dann : sonst*

Dazu ein Beispiel:

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

int main(void)
{
    cout << (1 < 2 ? "ja" : "nein");
    return 0;
}
```

Ausgabe: ja

Die Bedingung

```
1 < 2
```

ist true, daher besitzt der gesamte Ausdruck

```
1 < 2 ? "ja" : "nein"
```

den Wert "ja", was dem Wert des zweiten Ausdrucks (*Ausdruck2*) entspricht.

### Hinweis

Der Ausdruck

```
1 < 2 ? "ja" : "nein"
```

muss in der Anweisung

```
cout << (1 < 2 ? "ja" : "nein");
```

geklammert werden, da der Ausgabeoperator (<<) eine höhere Priorität als der Bedingungsoperator (?:) besitzt.

Häufig wird der Bedingungsoperator in Verbindung mit einer Zuweisung verwendet:

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

int main(void)
{
    double z1, z2, groesser;
    cin >> z1 >> z2;
    groesser = z1 > z2 ? z1 : z2;
    cout << "Wert von groesser: " << groesser;
    return 0;
}
```

Ausgabe:

```
15
20
Wert von groesser: 20
```

Mit der Zuweisung

```
groesser = z1 > z2 ? z1 : z2;
```

erhält die Variable *groesser* die größtmögliche der eingegebenen Zahlen. Falls der in *z1* gespeicherte Wert größer ist als der in *z2*, bekommt die Variable *groesser* den Wert der Variablen *z1* zugewiesen, ansonsten den Wert der Variablen *z2*.

**Hinweis**

Die Kombination von Bedingungsoperator und Zuweisung wird auch *bedingte Zuweisung* genannt.

Betrachten wir nun das Beispiel mithilfe eines `if else`-Konstrukts umgesetzt. Hier zeigen sich die Gemeinsamkeiten – beim Bedingungsoperator handelt es sich letztendlich – wie vorhin angedeutet – um eine Kurzform der `if else`-Anweisung. Der Bedingungsoperator ist aber dieser gegenüber eingeschränkt, da er nur einen Wert zurückgibt, aber nicht die Ausführung von weiteren Aktionen – realisiert über Anweisungen – erlaubt.

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

int main(void)
{
    double z1, z2, groesser;
    cin >> z1 >> z2;
    if(z1 > z2)
        groesser = z1;
    else
        groesser = z2;
    cout << "Wert von groesser: " << groesser;
    return 0;
}
```

Ausgabe wie zuvor.

## 16.5 Zufallszahlen auslosen

Zum Abschluss dieses Kapitels möchte ich Ihnen eine Funktion (eigentlich sind es zwei, mit der Funktion `time()` sogar drei) vorstellen, die es ermöglicht, in einem Programm Zufallszahlen zu generieren. Die Funktion heißt `rand()`. Sie liefert eine Zufallszahl im Bereich 0 bis `RAND_MAX`, wobei `RAND_MAX` eine vordefinierte symbolische Konstante ist, deren Wert vom verwendeten Compiler abhängt. Der C++-Standard (gemeint ist der ANSI/ISO-Standard) schreibt diesbezüglich keinen bestimmten Wert vor. Bei den meisten Compilern – so bei dem von Borland wie auch beim C++-Compiler des Visual Studio – entspricht dieser Wert 32767, sodass ein Aufruf von `rand()` eine Zufallszahl zwischen 0 und 32767 liefert.

**Hinweis**

Wenn man eine Funktion verwendet, sagt man, »die Funktion wird aufgerufen«. Dementsprechend spricht man von einem Funktionsaufruf.

Falls eine Funktion, die einen Rückgabewert besitzt, im Code verwendet – aufgerufen – wird, so steht der Funktionsaufruf als solcher zugleich für den Rückgabewert der aufgerufenen Funktion. Das heißt nichts anderes, als dass der Ausdruck

```
rand()
```

im Code so zu behandeln ist wie etwa ein Literal, das im Wert dem Rückgabewert der aufgerufenen Funktion entspricht. Demnach kann man diesen Ausdruck einfach `cout` übergeben, um die geloste Zufallszahl anzeigen zu lassen.

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

int main(void)
{
    cout << rand();
    return 0;
}
```

Ausgabe ist eine Zahl zwischen 0 und 32767 – wobei wir hier voraussetzen, dass `RAND_MAX` dem Wert 32767 entspricht.

### Hinweis

Möglicherweise müssen Sie die Headerdatei *cstdlib* – bei älteren Compilern *stdlib.h* – einbinden, um die Funktion `rand()` verwenden zu können:

```
#include <cstdlib>

bzw.

#include <stdlib.h>
```

Beachten Sie, dass es sich bei dem Funktionsaufruf `rand()` definitionsgemäß um einen Ausdruck handelt, da dieser ja für einen Wert – eben den Rückgabewert der bezeichneten Funktion – steht. Man könnte also bezüglich des obigen Listings anstelle von »die Funktion `rand()` besitzt den Rückgabewert ...« genauso gut »der Ausdruck `rand()` besitzt den Rückgabewert ...« sagen (zu Ausdrücken siehe Kapitel 11, Abschnitt 11.1 »Was ist ein Ausdruck?«).

### Hinweis

Dies gilt natürlich für alle Funktionen, insofern diese einen Rückgabewert besitzen.

Um den Rückgabewert der Funktion `rand()` – die geloste Zahl – dauerhaft festzuhalten, können Sie diesen auch in einer Variablen speichern. Dem oben Gesagten zufolge muss in diesem Fall der Funktionsaufruf – der Ausdruck `rand()` – auf der rechten Seite der Zuweisung stehen:

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

int main(void)
{
    int glueck;
    glueck = rand();
    cout << glueck;
    return 0;
}
```

Ausgabe wie zuvor.

Beachten Sie, dass jede erneute Verwendung von `rand()` eine weitere Zufallszahl erzeugt:

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

int main(void)
{
    cout << rand() << endl;
    cout << rand() << endl;
    cout << rand() << endl;
    cout << rand() << endl;
    cout << rand() << endl;
    return 0;
}
```

Ausgabe:

```
41
18467
6334
26500
19169
```

Ausgegeben werden fünf Zufallszahlen im Bereich von 0 bis 32767. Falls Sie das Programm wiederholt ausführen, werden Sie feststellen, dass immer die gleiche Folge von Zufallszahlen ausgelost wird. Dies liegt daran, dass der Zufallsgenerator mit einem Algorithmus arbeitet, der auf einem Startwert basiert, wobei sich aus einem bestimmten Startwert auch stets die gleiche Folge von Zufallszahlen ergibt. Der Zufallsgenerator benutzt permanent den Startwert 1, falls ihm nicht durch einen Aufruf der Funktion `srand()` ein anderer Ausgangswert übermittelt wird. Der Funktion `srand()` ist beim Aufruf der gewünschte Startwert zu übergeben. Der Aufruf

```
srand(5)
```

initialisiert den Zufallsgenerator folglich mit dem Wert 5. Allerdings führt dies auch nicht zum gewünschten Ergebnis, da sich nun zwar eine andere, jedoch sich ebenfalls stets wiederholende Folge von Zufallszahlen ergibt.

Das Problem lässt sich lösen, indem man `srand()` den Rückgabewert der Funktion `time()` übergibt und zwar in der Form

```
srand(time(NULL))
```

wobei der Ausdruck

```
time(NULL)
```

die seit dem 1. Januar 1970 vergangenen Sekunden als Integerwert zurückliefert. Somit ist sichergestellt, dass sich dieser Wert von Programmstart zu Programmstart ändert.

Denken Sie daran, dass ein Funktionsaufruf gleichzeitig für den Rückgabewert der aufgerufenen Funktion steht, `time(NULL)` also für den besagten Integerwert. Daher ist es möglich, einer anderen Funktion – hier `srand()` – einen Wert in Form eines Funktionsaufrufs zu übergeben.

Über den Parameter `NULL` brauchen Sie sich hier noch keine Gedanken zu machen – es handelt sich um einen so genannten Nullzeiger (zum Thema Zeiger siehe Kapitel 28). Im Übrigen gilt das auch für Funktionen. Diese sind Gegenstand des 20. Kapitels. Dort werden Themen wie Funktionsaufrufe, Rückgabewerte von Funktionen, Parameter von Funktionen usw. ausführlich behandelt. Schaden kann es aber nichts, an dieser Stelle schon einmal etwas davon gehört zu haben. Wenn Sie in Ihren Programmen Zufallszahlen benötigen, initialisieren Sie den Zufallsgenerator einfach mit der Anweisung

```
srand(time(NULL));
```

Um die Funktion `rand()` sinnvoll verwenden zu können, muss diese Initialisierung vor dem ersten Aufruf von `rand()` erfolgen, und zwar nur ein einziges Mal. Für die Funktion `time()` ist es erforderlich, die Headerdatei `ctime` (bei älteren Compilern `time.h`) einzubinden.

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

int main(void)
{
    // ...
    srand(time(NULL));
    cout << rand() << endl;
    //...
    cout << rand() << endl;
    // ...
    return 0;
}
```

Hier ein kleines Programm, das den Benutzer eine Zufallszahl raten lässt. Die mittels der Funktion `rand()` ausgeloste Zufallszahl wird in der Variablen `glueck` und der Rateversuch des Benutzers in der Variablen `rate` festgehalten. Anschließend werden beide Zahlen miteinander verglichen (`rate == glueck`).

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

int main(void)
{
    int glueck, rate;
    srand(time(NULL));
    glueck = rand();
    cout << "Raten Sie mal! ";
    cin >> rate;
    if (rate == glueck)
    {
        cout << "Richtig geraten!";
    }
    else
    {
        cout << "Daneben! - Die Zahl war "
              << glueck;
    }
    cout << endl;
    return 0;
}
```

Ausgabe:

```
Raten Sie mal! 7
Daneben! - Die Zahl war 3882
```

Allerdings ist bei einem Chancenverhältnis von 1:32768 das Programm gegenüber dem Benutzer allzu sehr im Vorteil. Von daher ist es wünschenswert, die Auslösung der Zufallszahl auf einen kleineren Bereich einzugrenzen. Hierzu bietet sich der Modulo-Operator an, wobei zu beachten ist, dass ein Ausdruck

```
A % B
```

der ja dem Restwert einer Integerdivision  $A / B$  entspricht, nie größer sein kann als  $B - 1$ . Oder genauer: Die möglichen Werte, die dieser Ausdruck annehmen kann, liegen im Bereich von 0 bis  $B - 1$ . Um also eine Zufallszahl im Bereich von 0 bis 10 zu erhalten, kann `rand()` wie folgt verwendet werden:

```
rand() % 11
```

Dieser Ausdruck liefert stets Werte im Bereich von 0 bis 9 (einschließlich). Möchte man eine Zufallszahl im Bereich von 1 bis 10, lautet der entsprechende Ausdruck:

```
(rand() % 10) + 1
```

Die äußeren Klammern sind von der Priorität her nicht notwendig. Sie dienen hier allein der Verdeutlichung.

### Hinweis

Um eine Lottozahl auszulosen, die ja im Bereich von 1 bis 49 liegt, verwenden Sie demnach den Ausdruck

```
(rand() % 49) + 1
```

Hier das entsprechend modifizierte Listing – die Zufallszahl liegt nun im Bereich von 1 bis 10:

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

int main(void)
{
    int glueck, rate;
    srand(time(NULL));
    glueck = (rand() % 10) + 1;
    cout << "Raten Sie mal! ";
    cin >> rate;
    if (rate == glueck)
    {
        cout << "Richtig geraten!";
    }
    else
    {
        cout << "Daneben! - Die Zahl war "
            << glueck;
    }
    cout << endl;
    return 0;
}
```

Ausgabe:

```
Raten Sie mal! 5
Daneben! – Die Zahl war 4
```



## CD-ROM

Das Programm finden Sie als *k16s* im Ordner *Beispiele/K16* auf der Buch-CD.



Abbildung 16.2: Ratespiel »1 aus 10«



# 17

## Wiederholungs- anweisungen

Vielfach ist es notwendig, bestimmte Anweisungen wiederholt auszuführen. Dann müssen Sie diese nicht etwa erneut in den Quellcode schreiben. Zudem ist die Anzahl der erforderlichen Wiederholungen zum Zeitpunkt der Programmentwicklung oft gar nicht vorhersehbar, da sie der Logik entsprechend von einer Entscheidung oder einem bestimmten Verhalten des Benutzers abhängt. Ein Beispiel ist, dass dem Benutzer angeboten wird, eine bestimmte Folge von Aktionen erneut auszuführen oder an dieser Stelle abzubrechen.

Zur Einrichtung von Wiederholungsabläufen stellt C++ spezielle Anweisungen zur Verfügung. Diese zählen wie die im letzten Kapitel besprochenen Verzweigungsanweisungen zu den Kontrollstrukturen, da sie ebenfalls eine Steuerung des Programmflusses ermöglichen. Wegen ihres kreisförmigen Ablaufs werden sie als »Schleifen« bezeichnet.

### 17.1 Die while-Schleife

Die while-Schleife erlaubt es, in Abhängigkeit von einer Bedingung zu entscheiden, ob bzw. wie viele Male eine Anweisung bzw. ein Anweisungsblock ausgeführt werden soll. Ihre Syntax lautet:

```
while (Bedingung)  
{  
    Anweisung(en)  
}
```

Falls genau eine Anweisung verwendet werden soll, muss diese nicht als Block gekennzeichnet werden. Folglich können die geschweiften Klammern weggelassen werden. Die Syntax lautet entsprechend:

```
while (Bedingung)  
    Anweisung
```

Beginnend mit dem Schlüsselwort `while` schließt sich eine Bedingung an, gefolgt vom Anweisungsteil. Beachten Sie die syntaktische – und auch funktionale – Ähnlichkeit mit der `if`-Anweisung.

Bei Eintritt in die while-Schleife wird zunächst die Schleifenbedingung geprüft. Hierbei handelt es sich wie bei der `if`-Anweisung um einen logischen Ausdruck, der wahr oder falsch sein kann. Ist er wahr, wird der zugehörige Anweisungsteil ausgeführt. Nach jedem Schleifendurchgang wird die Bedingung erneut geprüft. Der Anweisungsteil der

while-Schleife wird also so lange wiederholt, wie die Auswertung der Schleifenbedingung den Wert `true` ergibt. Andernfalls wird die Schleife verlassen und die Ausführung mit der nächsten Anweisung außerhalb der `while`-Schleife fortgesetzt.

### Hinweis

Der Anweisungsteil von Schleifen wird auch »Schleifenrumpf« genannt. Analog spricht man mitunter vom Kopf einer Schleife. Letzteres allerdings vorwiegend im Zusammenhang mit der `for`-Schleife, nie im Zusammenhang mit der `do while`-Schleife, da diese praktisch keinen »Kopf« besitzt (zu den Schleifenarten siehe Abschnitt 17.2 »Die `do while`-Schleife« und Abschnitt 17.3 »Die `for`-Schleife«).

Beispiel:

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

int main(void)
{
    int zahl = 9;
    while (zahl > 0)
    {
        cout << zahl << "-";
        zahl--;
    }
    cout << "ZERO";
    return 0;
}
```

Ausgabe: 9-8-7-6-5-4-3-2-1-ZERO

Die Variable `zahl` erhält bei ihrer Deklaration den Startwert 9. Die Auswertung der Schleifenbedingung ergibt `true` ( $9 > 0$ ), und die Anweisungen des Schleifenrumpfes werden ausgeführt: Es wird der aktuelle Wert von `zahl`, der ja bei Schleifeneintritt 9 ist, zusammen mit dem Zeichen »-« ausgegeben (9-), anschließend die Variable dekrementiert, das heißt der Wert von `zahl` um den Betrag von 1 verringert. Die anschließende Prüfung der Schleifenbedingung ergibt wiederum `true` ( $8 > 0$ ), also wird der Anweisungsteil der `while`-Schleife zum zweiten Mal ausgeführt: Die Ausgabe ist nun 8-, danach wird `zahl` dekrementiert. Nun wird die Schleifenbedingung erneut geprüft ( $7 > 0$ ) usw. Dies setzt sich so lange fort, bis die Variable `zahl` den Wert 0 besitzt. Für diesen Wert von `zahl` ergibt die Auswertung der Schleifenbedingung `false` ( $0 > 0$ ), und die `while`-Schleife wird verlassen.

Die `while`-Schleife ist eine *abweisende* Schleife. Das heißt, es kann vorkommen, dass der zugehörige Anweisungsteil kein einziges Mal ausgeführt wird. Dieser Fall tritt ein, wenn die erste Auswertung der Schleifenbedingung bereits den Wert `false` ergibt – was für obiges Beispiel dann zutrifft, wenn die Variable `zahl` mit einem Initialisierungswert versehen wird, der kleiner ist als 1.

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

int main(void)
{
    int zahl = 0;
    while (zahl > 0)
    {
        cout << zahl << "-";
        zahl--;
    }
    cout << "ZERO";
    return 0;
}
```

Ausgabe: ZERO

Dieses Listing dient natürlich nur der Anschauung. Vom praktischen Standpunkt ist die while-Schleife hier sinnlos, da ja von vornherein feststeht, dass diese nie durchlaufen wird. Anders verhält es sich, wenn die Anzahl der Schleifendurchgänge vom Verhalten des Benutzers bei der Programmausführung abhängt. Dann wird diese in der Regel von Programmlauf zu Programmlauf variieren, wobei es durchaus vorkommen kann, dass eine while-Schleife in bestimmten Fällen überhaupt nicht zur Ausführung gelangt:

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

int main(void)
{
    int zahl;
    cin >> zahl;
    while (zahl > 0)
    {
        cout << zahl << "-";
        zahl--;
    }
    cout << "ZERO";
    return 0;
}
```

Gibt der Benutzer hier eine Zahl ein, die kleiner ist als 1, ergibt die Bedingungsprüfung der while-Schleife schon bei der ersten Auswertung den Wert `false`, und der zugehörige Anweisungsteil wird kein einziges Mal ausgeführt. In allen anderen Fällen – wenn die eingegebene Zahl 1 oder größer als 1 ist – kommt der Schleifenrumpf zumindest ein Mal zur Ausführung, wobei sich die Wiederholungszahl wiederum nach der Größe der Eingabezahl richtet.

### 17.1.1 Endlosschleifen

Einen meist unerwünschten Sonderfall stellt die so genannte »Endlosschleife« dar. So werden Schleifen genannt, die nie abbrechen, da die Auswertung der Schleifenbedingung stets den Wert `true` ergibt. Die Endlosschleife beruht in der Regel auf einem Versehen des Programmierers.

Unterlässt man es z.B. in den obigen Listings, die Variable `zahl` im Schleifenrumpf zu dekrementieren (`zahl--`), entsteht eine Endlosschleife, da sich in diesem Fall das Ergebnis der Schleifenbedingung (`zahl > 0`) nach der ersten Auswertung nicht mehr ändern kann. Das heißt, die Programmausführung bleibt in der Schleife »hängen«.

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

int main(void)
{
    int zahl = 9;
    while (zahl > 0)
    {
        cout << zahl << "-";
    }
    cout << "ZERO";
    return 0;
}
```

Ausgabe: 9-9-9-9-9-9-9-9-9-9-9-...

#### Tipp

Falls Sie mit einem Kommandozeilen-Compiler arbeiten und Sie es mit einer Endlosschleife zu tun haben, versuchen Sie, mit einer der Tastenkombinationen `[Strg]+[C]` oder `[Strg]+[Pause]` das Programm zu beenden. Wenn das nichts hilft, schließen Sie das Konsolenfenster mit dem Schließen-Button rechts oben in der Titelleiste. In einer IDE wie dem Visual Studio von Microsoft brauchen Sie gewöhnlich nur das Ausgabefenster zu schließen, ohne die Programmieroberfläche verlassen zu müssen.

Wie gesagt, in den seltensten Fällen sind Endlosschleifen vom Programmierer beabsichtigt. Achten Sie also stets darauf, den Schleifenrumpf so einzurichten, dass sich die Schleifenbedingung nach einer Wiederholung auch wirklich ändern kann. Diese Aussage gilt im Übrigen für alle Schleifenkonstrukte.

## 17.1.2 Fakultät berechnen

Es gilt:

- Die Fakultät  $n$  einer Zahl – geschrieben  $n!$  – ist das Produkt aller möglichen ganzzahligen Faktoren von  $n$  bis 1.
- Die Fakultät der Zahl 0 ist definitionsgemäß 1.

Beispiele:

$$5! = 5 * 4 * 3 * 2 * 1$$

$$4! = 4 * 3 * 2 * 1$$

$$3! = 3 * 2 * 1$$

$$2! = 2 * 1$$

$$1! = 1$$

$$0! = 1$$

Hier ein Programm, das die Fakultät einer vom Benutzer einzugebenden Zahl berechnet. Die eigentliche Berechnung erfolgt in einer `while`-Schleife:

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

int main(void)
{
    int fakul = 1, zahl, hilf;
    cin >> zahl;
    hilf = zahl;
    while (hilf > 0)
    {
        fakul *= hilf--;
    }
    cout << zahl << "! = " << fakul;
    return 0;
}
```

Ausgabe:

```
7
7! = 5040
```

In der zusammengesetzten Zuweisung

```
fakul *= hilf--;
```

wird `hilf` jeweils nach der Zuweisung dekrementiert (Postfixnotation). Der Operator `*` multipliziert den aktuellen Wert der Variablen `fakul` mit dem Wert der Variablen `hilf`.

Diese nimmt im Verlauf der Wiederholungen nacheinander alle Werte von `zahl` bis 0 an, wobei die Multiplikation für den Wert 0 nicht mehr ausgeführt wird, da die `while`-Schleife vorher abbricht.

Falls die eingegebene Zahl 0 ist, kommt die `while`-Schleife kein einziges Mal zur Ausführung. Das Programm arbeitet auch in diesem Fall korrekt, da der Initialisierungswert 1 von `fakul` dem Ergebnis von 0! (lies »Null Fakultät«) entspricht.

Beachten Sie, dass die in der Variablen `zahl` gespeicherte Eingabezahl der Hilfsvariablen `hilf` zugewiesen wird, damit `zahl` im Zuge der folgenden Berechnung nicht verändert wird. Was die Berechnung an sich betrifft, ließe sich in der `while`-Schleife genauso gut die Variable `zahl` verwenden:

```
...
while (zahl > 0)
{
    fakul *= zahl--;
}
...
```

Allerdings ist es auf diese Weise nicht mehr möglich, `zahl` in der Ausgabe zu verwenden, da diese nun bei Schleifenaustritt in jedem Fall den Wert 0 besitzt. So hätte dann z.B. bei Eingabe der Zahl 5 die Anweisung

```
cout << zahl << "! = " << fakul;
```

nicht die korrekte Ausgabe

```
5! = 120
```

sondern die Ausgabe

```
0! = 120
```

zur Folge.

Es sei angemerkt, dass das obige Programm nur für Eingabezahlen bis einschließlich 12 korrekte Ergebnisse liefert, vorausgesetzt der Datentyp `int` ist 4 Byte groß.

### Hinweis

Andernfalls empfiehlt es sich, die Variable als `long (int)` zu deklarieren:

```
long fakul = 1, zahl, hilf;
```



Man könnte nun etwa mit einer if-Bedingung verhindern, dass Berechnungen mit Werten, die größer als 12 sind, überhaupt durchgeführt werden:

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

int main(void)
{
    int fakul = 1, zahl, hilf;
    cin >> zahl;
    if (zahl <= 12)
    {
        hilf = zahl;
        while (hilf > 0)
        {
            fakul *= hilf--;
        }
        cout << zahl << "! = " << fakul;
    }
    else
    {
        cout << "Bei " << zahl;
        cout << "! muss ich passen";
    }
    return 0;
}
```

Ausgabe:

```
12
12! = 479001600
```

bzw.

```
13
Bei 13! muss ich passen
```

Allerdings lässt sich die Rechenkapazität auch erhöhen, indem man mit double-Werten arbeitet:

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

int main(void)
{
    double fakul = 1, zahl, hilf;
```

```

cin >> zahl;
hilf = zahl;
while (hilf > 0)
{
    fakul *= hilf--;
}
cout << fixed << setprecision(0);
cout << zahl << "! = " << fakul;
return 0;
}

```

Ausgabe:

```

16
16! = 20922789888000

```

Natürlich sind auch dem Grenzen gesetzt, da die Berechnungsergebnisse allzu schnell ansteigen, wie an dem Ergebnis von 16! zu erkennen ist.

### Hinweis

Die Manipulatoren `fixed` und `setprecision(0)` sind notwendig, um die Ausgabe von `zahl` und `fakul` in der gewohnten Form – also nicht in Exponentialschreibweise – und ohne Nachkommastellen anzuzeigen (siehe dazu Kapitel 13 »Ausgabe mit Manipulatoren formatieren«).

Hier noch ein letztes Beispiel zur `while`-Schleife: das Ratespiel des letzten Kapitels in modifizierter Form. Der Benutzer darf bzw. er muss so lange raten, bis er die Zufallszahl getroffen hat. Da diese im Bereich von 1 bis 10 liegt, ist die Gefahr einer »praktischen« Endlosschleife wohl sehr gering:

```

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

int main(void)
{
    int glueck, rate;
    srand(time(NULL));
    glueck = (rand() % 10) + 1;
    cout << "Rate mal! ";
    cin >> rate;
    while (rate != glueck)
    {
        cout << "Versuch's nochmal: ";
    }
}

```

```
    cin >> rate;
}
    cout << "Richtig geraten!" << endl;
    return 0;
}
```

Ausgabe:

```
Rate mal! 3
Versuch's nochmal: 8
Versuch's nochmal: 4
Versuch's nochmal: 9
Versuch's nochmal: 7
Versuch's nochmal: 1
Versuch's nochmal: 10
Versuch's nochmal: 6
Richtig geraten!
```

## 17.2 Die do while-Schleife

Die do while-Schleife funktioniert im Wesentlichen wie die while-Schleife, mit dem Unterschied, dass die Prüfung der do while-Schleife erst nach dem Schleifendurchgang erfolgt. Das hat zur Folge, dass die Schleife auf jeden Fall zumindest ein Mal durchlaufen wird. Man spricht daher in diesem Zusammenhang auch von einer *annehmenden* Schleife. Die Syntax der do while-Schleife lautet:

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

bzw.

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

Falls es sich um nur eine Anweisung handelt, sind beide Varianten möglich. Mehrere Anweisungen müssen zu einem Block zusammengefasst werden. Beachten Sie, dass die do while-Schleife mit einem Semikolon abgeschlossen wird.

Ein Einsatz der do while-Schleife ergibt vor allem dann Sinn, wenn die Anweisungen des Schleifenrumpfes zumindest ein Mal ausgeführt werden müssen. Im folgenden Beispiel wird die do while-Schleife verwendet, um eine Eingabe gegebenenfalls so lange zu wiederholen, bis der Benutzer eine positive Zahl angibt:

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

int main(void)
{
    int zahl;
    do
    {
        cout << "Gib mir eine positive Zahl: ";
        cin >> zahl;
    }while (zahl < 0);
    cout << "Danke!" << endl;
    return 0;
}
```

Die erste Eingabe wird auf jeden Fall entgegengenommen (`cin >> zahl;`), danach in der Schleifenbedingung geprüft, ob die eingegebene Zahl negativ ist (`zahl < 0`). Ist das der Fall, ist die Schleifenbedingung wahr und die Schleife wird erneut durchlaufen, das heißt, die Eingabe wird wiederholt. Dies setzt sich so lange fort, bis der Benutzer eine positive Zahl eingibt.

### Hinweis

Beachten Sie, dass es sich so verhält, als stünden die Anweisungen

```
cout << "Gib mir eine positive Zahl: ";
```

```
cin >> zahl;
```

alleine – ohne Schleife – im Code, falls schon die erste Eingabe, wie vom Programm gefordert, positiv ist. Die `do while`-Schleife hat in diesem Fall keinerlei äußere Wirkung.

Da der Benutzer unseres Ratespiels auf jeden Fall ein Mal tippen soll, bietet sich auch hier eine Umsetzung mit der `do while`-Schleife an:

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

int main(void)
{
    int glueck, rate;
    srand(time(NULL));
    glueck = (rand() % 10) + 1;
```

```

do
{
    cout << "Dein Tipp: ";
    cin >> rate;
} while (rate != glueck);
    cout << "Richtig!" << endl;
return 0;
}

```

Ausgabe:

```

Dein Tipp: 3
Dein Tipp: 5
Dein Tipp: 9
Richtig!

```

## 17.3 Die for-Schleife

Was den Einsatz von Schleifen betrifft, lassen sich grob zwei Fälle unterscheiden:

- Die Anzahl der Schleifendurchläufe ist zum Zeitpunkt der Programmentwicklung nicht bekannt, da sie von bestimmten äußeren Bedingungen abhängt.
- Die Wiederholungszahl steht bei Programmentwicklung bereits fest.

Während für die Lösung von Problemen der ersten Kategorie überwiegend die `while`- und die `do while`-Schleife zum Einsatz kommen, eignet sich die `for`-Schleife besonders dann, wenn die Gesamtzahl der Wiederholungen von vornherein feststeht. Dementsprechend ist die `for`-Schleife als typische Zählschleife vorgesehen. Ihre Syntax lautet

```

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

```

wobei die geschweiften Klammern wiederum optional sind, wenn es sich um eine einzige Anweisung handelt.

Etwas ungewöhnlich gegenüber den beiden anderen Schleifenarten präsentiert sich der Kopf der `for`-Schleife:

```

for(Initialisierung; Bedingung; Aktualisierung)

```

Die Schleifenbedingung entspricht der `while`-Schleife. Sie wird vor jedem Schleifendurchlauf geprüft. Der Initialisierungsteil kommt nur ein einziges Mal – bei Eintritt in die `for`-Schleife, noch vor der ersten Bedingungsprüfung –, der Aktualisierungsteil nach jedem Schleifendurchgang zur Ausführung.

## Hinweis

Beachten Sie, dass die Bedingung mit einem Semikolon abgeschlossen wird, der Aktualisierungsteil aber nicht. Letzteres ist insofern bemerkenswert, als dort in der Regel Anweisungen – gewöhnlich eine – stehen. Die Semikola fungieren hier ausnahmsweise als Trennzeichen zwischen den genannten Teilen, nicht zwischen Anweisungen.

Die Bezeichnungen »Initialisierungsteil« und »Aktualisierungsteil« beziehen sich bereits auf eine sinnvolle Verwendung der for-Schleife. Tatsächlich darf an den entsprechenden Stellen jeweils eine beliebige Anzahl von Anweisungen stehen, wobei C++ auch über die Art der Anweisungen nichts vorschreibt. Mit »beliebige Anzahl« ist gemeint, dass entweder genau eine Anweisung oder mehrere, durch Kommata voneinander getrennte Anweisungen oder auch gar keine Anweisung verwendet werden kann. Die beiden folgenden Beispiele stellen daher syntaktisch fehlerfreie for-Schleifen dar:

```
int x;
for (cout << "Hi", cout << "lfe"; true; cin >> x)
    cout << "???";
```

oder

```
for (;true;)
    cout << "Endlosschleife";
```

Das ist aber auch schon das Einzige, denn diese beiden for-Schleifen sind nicht nur zweckentfremdet gestaltet, sondern bilden – aufgrund der Schleifenbedingungen `true` – auch Endlosschleifen.

Hierzu muss gesagt werden: C++ gibt dem Programmierer viele Freiheiten, was auch dazu führen kann, dass die gebotenen Möglichkeiten nicht sinnvoll eingesetzt werden. Gedacht ist die for-Schleife, wie gesagt, als eine Art Zählschleife. Initialisierungs- und Aktualisierungsteil werden in Verbindung mit der Schleifenbedingung aufeinander abgestimmt, um die Anzahl der Wiederholungen zu steuern. Dazu wird eine zusätzliche Variable – Schleifenvariable oder Laufvariable genannt – herangezogen. In der Regel wird diese im Initialisierungsteil auf einen Startwert gesetzt, im Aktualisierungsteil wird der Wert der Variablen verändert und in der Schleifenbedingung wird diese Variable auf ihren aktuellen Wert hin geprüft. Meist wird sich also im Initialisierungsteil und im Aktualisierungsteil jeweils genau eine Anweisung befinden:

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

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

```
for (i = 1; i < 4; i++)  
{  
    cout << "Wiederholung ";  
}  
return 0;  
}
```

Ausgabe:

```
Wiederholung Wiederholung Wiederholung
```

Die Anweisung im Initialisierungsteil wird nur ein einziges Mal ausgeführt, und zwar vor dem ersten Schleifendurchgang:

```
i = 1;
```

Damit wird die Schleifenvariable *i* auf den Anfangswert 1 gesetzt.

### Hinweis

Wie alle anderen Variablen muss auch die Schleifenvariable der *for*-Schleife vor Gebrauch vereinbart werden. Gebräuchliche Bezeichner sind *i*, *j* und *k*.

Im Aktualisierungsteil wird *i* mit der Anweisung

```
i++;
```

um 1 hochgezählt. Dies geschieht nach jedem Schleifendurchlauf, also nach jeder Ausführung des Anweisungsteils, der hier allein aus einer Anweisung besteht. Vor jedem Schleifendurchgang erfolgt die Bedingungsprüfung

```
i < 4
```

sodass die Schleife abbricht, sobald *i* den Wert 4 besitzt. Im Ergebnis kommt die Anweisung

```
cout << "Wiederholung ";
```

drei Mal zur Ausführung, für die Werte 1, 2, 3 von *i*.

Beachten Sie, dass die Schleifenbedingung auch schon vor dem ersten Schleifendurchlauf ausgewertet wird. So kommt der Anweisungsteil in der folgenden *for*-Schleife gar nicht zur Ausführung, da bereits die erste Bedingungsprüfung den Wert *false* ergibt:

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

int main(void)
{
    int i;
    for (i = 1; i < 1; i++)
    {
        cout << "Wiederholung ";
    }
    cout << "ENDE";
    return 0;
}
```

Ausgabe: ENDE

### Hinweis

Es ist auch möglich, die Schleifenvariable im Kopf der for-Schleife zu deklarieren:

```
for (int i = 1; i < 4; i++)
{
    cout << "Wiederholung ";
}
```

Oft wird die Schleifenvariable im Anweisungsteil der for-Schleife zu Berechnungen herangezogen oder für die Ausgabe verwendet:

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

int main(void)
{
    int i;
    for (i = 1; i <= 9; i++)
    {
        cout << "Quadrat von "<< i << ": "
              << i * i << endl;
    }
    return 0;
}
```



Ausgabe:

```
Quadrat von 1: 1
Quadrat von 2: 4
Quadrat von 3: 9
...
Quadrat von 8: 64
Quadrat von 9: 81
```

### Achtung

Vermeiden Sie es jedoch, die Schleifenvariable im Schleifenrumpf zu manipulieren. Der Wert der Schleifenvariablen sollte ausschließlich im Aktualisierungsteil des Schleifenkopfes verändert werden. Führen Sie also im Schleifenrumpf keine Zuweisungen an die Laufvariable durch. Dies bringt nicht nur eine erhöhte Anfälligkeit für logische Fehler mit sich, es verschlechtert auch die Lesbarkeit Ihres Quellcodes erheblich.

```
for (i = 0; i < 10; i++)
{
    ...
    i *= 2;  // PROBLEMATISCH
    ...
}
```

Selbstverständlich können Sie die Schleifenvariable mit jedem beliebigen Startwert versehen. Gegebenenfalls müssen natürlich Bedingung und Aktualisierung entsprechend angepasst werden, um ein bestimmtes Resultat zu erzielen wie die drei Schleifendurchgänge aus dem letzten Beispiel:

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

int main(void)
{
    int i;
    for (i = -10; i > -16; i -= 2)
    {
        cout << "Wiederholung ";
    }
    return 0;
}
```

Ausgabe:

```
Wiederholung Wiederholung Wiederholung
```

Die Laufvariable `i` wird hier mit dem Wert `-10` initialisiert und nach jedem Schleifendurchlauf um den Wert `2` verringert – die Aktualisierung muss also nicht notwendig in Einer-Schritten erfolgen. Die Schleifenbedingung wurde entsprechend angepasst, sodass die Schleife wie im Einführungsbeispiel dreimal durchlaufen wird.

Hier eine Implementation des Programms zur Berechnung der Fakultät mit der `for`-Schleife:

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

int main(void)
{
    int fakul = 1, zahl, i;
    cin >> zahl;
    for(i = zahl; i > 0; i--)
    {
        fakul *= i;
    }
    cout << zahl << "! = " << fakul;
    return 0;
}
```

Ausgabe:

```
8
8! = 40320
```

Falls die Zahl `0` eingegeben wird, wird die `for`-Schleife überhaupt nicht durchlaufen. Da `fakul` bei der Deklaration mit dem Wert `1` initialisiert wird, gibt das Programm in diesem Fall das korrekte Ergebnis `– 1` aus. Beachten Sie, dass hier die Laufvariable `i` die Funktion der Hilfsvariablen `h1lf` bezüglich des Programms mit der `while`-Schleife übernimmt. Somit ist sichergestellt, dass die Variable `zahl` in der `for`-Schleife nicht verändert wird und daher für die Ausgabe verwendet werden kann (`cout << zahl ...`).

Folgendes Programm gibt die Zeichen des ASCII-Codes auf den Bildschirm aus:

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

int main(void)
{
    for (int i = 0; i < 256; i++)
    {
```

```

    if (i % 25 == 0)
        cout << endl << endl;
        cout << ' ' << static_cast<char> (i) << ' ';
    }
    cout << endl;
    return 0;
}

```

Die Schleifenvariable *i* nimmt nacheinander alle Ordnungsnummern des ASCII-Codes von 0 bis 255 an. Für die Ausgabe wird der Zahlenwert in das entsprechende Zeichen umgewandelt:

```
static_cast<char> (i)
```

Die in die for-Schleife geschachtelte if-Bedingung bewirkt, dass jeweils nach Ausgabe von 25 Zeichen zwei Zeilenvorschübe erfolgen:

```

if (i % 25 == 0)
    cout << endl << endl;

```

Die Ausgabe ist Abbildung 17.1 zu entnehmen. Bei den leeren Stellen in den ersten zwei Zeilen handelt es sich um nicht druckbare Zeichen.

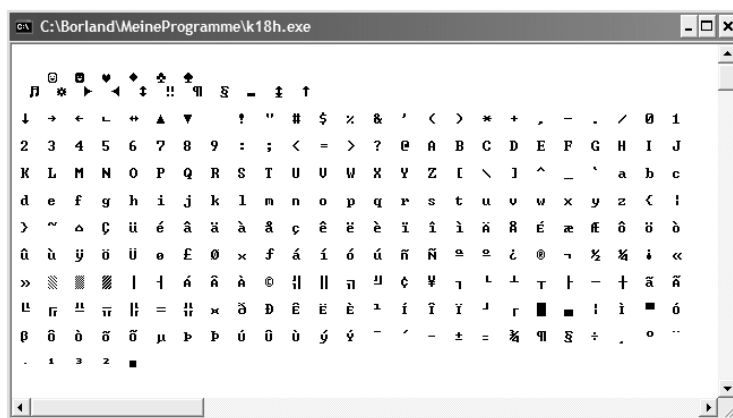


Abbildung 17.1: Zeichen des ASCII-Codes

## CD-ROM

Das Programm finden Sie als *k17n* im Ordner *Beispiele/K17* auf der Buch-CD

### 17.3.1 Sequenzoperator

Wir hatten oben erwähnt, dass im Initialisierungs- und im Aktualisierungsteil einer for-Schleife auch mehrere Anweisungen stehen dürfen und dass diese in diesem Fall durch Kommata zu trennen sind:

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

int main(void)
{
    int i, k;
    for (i = 0, k = 0; i < 5; i++, k = 2 * i)
        cout << i << ' ' << k << endl;
    return 0;
}
```

Ausgabe:

```
0 0
1 2
2 4
3 6
4 8
```

Bei den Kommata im Kopf der for-Schleife handelt es sich um den Sequenzoperator, mitunter auch Kommaoperator genannt. Er besitzt von allen Operatoren die niedrigste Priorität (Prioritätsstufe 1, siehe im Anhang »Prioritätsreihenfolge der C++-Operatoren«).

Grundsätzlich kann der Sequenzoperator in jeder Anweisung verwendet werden. Das heißt, Sie dürfen in einer Anweisung mehrere Ausdrücke unterbringen, wenn Sie diese mit Kommata voneinander trennen. Wobei dies oft Ausdrücke sein werden, die in der gleichen Form auch als einzelne Anweisungen im Code stehen dürften:

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

int main(void)
{
    int z1, z2, z3;
    z1 = 19, z2 = 3, z3 = 2;
    cout << z1 + z2 + z3;
    return 0;
}
```

Ausgabe: 24

**Achtung**

Vorsicht jedoch bei Deklarationsanweisungen. In diesen darf grundsätzlich nur ein Datentypbezeichner auftreten. Hier steht das Komma nicht für einen Sequenzoperator, sondern fungiert als einfaches Trennzeichen.

```
double z1, char z2; // FEHLER
```

Der Sequenzoperator ist binär und linksassoziativ, das heißt, von seinen beiden Operanden – den Ausdrücken links und rechts vom Komma – wird der linke zuerst ausgewertet. Der Rückgabewert mehrerer, durch Kommata getrennter Ausdrücke entspricht dem am weitesten rechts stehenden:

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

int main(void)
{
    int z1, z2, z3;
    cout << (z1 = 1, z2 = ++z1, z3 = ++z2);
    return 0;
}
```

Ausgabe: 3

Der Sequenzoperator wird in der Regel nur dort eingesetzt, wo aufgrund syntaktischer Vorgaben ansonsten nicht mehr als ein Ausdruck bzw. eine Einzelanweisung erlaubt wäre, wie etwa im Kopf einer for-Schleife.

## 17.4 Die Anweisungen break und continue

Die break-Anweisung haben Sie bereits im Zusammenhang mit der switch-Anweisung kennen gelernt. Ebenso wie diese beendet break auch jede Schleife.

In der folgenden Kennwortabfrage wird ein vom Benutzer einzugebendes Passwort auf Übereinstimmung mit der Zeichenfolge "xyz" geprüft. Der Benutzer hat drei Versuche, um das richtige Kennwort einzugeben. Für den Fall, dass er dies schon beim ersten oder zweiten Versuch tut, darf die Anweisung

```
cin >> kennwort;
```

natürlich nicht mehr ausgeführt werden. Daher wird die for-Schleife mit der break-Anweisung vorzeitig verlassen, wenn die Bedingung

```
(kennwort == "xyz")
```

wahr ist:

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

int main(void)
{
    string kennwort;
    for(int i = 0; i < 3; i++)
    {
        cin >> kennwort;
        if (kennwort == "xyz")
            break;
    }
    if (kennwort != "xyz")
        return 0; // beendet das Programm
    cout << "Willkommen im Programm!";
    // ...
    return 0;
}
```

Beachten Sie, dass der Fall zu berücksichtigen ist, dass erst beim dritten Versuch das richtige Kennwort angegeben wird. Das heißt, bei Austritt aus der for-Schleife lässt sich keine Aussage darüber treffen, ob diese vorzeitig verlassen wurde – weil das eingegebene Kennwort richtig war – oder nach drei erfolglosen Versuchen. Daher muss die Bedingung nach der for-Schleife erneut geprüft werden:

```
if (kennwort != "xyz")
```

Ist diese Bedingung wahr, so bedeutet dies, dass der Benutzer dreimal ein falsches Kennwort eingegeben hat. In diesem Fall wird das Programm mit

```
return 0;
```

beendet.

### Hinweis

Genauso wie `break` eine Schleife beendet, führt `return` zur Beendigung einer Funktion. Da es sich hier um die Funktion `main()` handelt, ist dies gleichbedeutend mit dem Ende der Programmausführung (mehr zur `return`-Anweisung erfahren Sie in Kapitel 20, Abschnitt 20.4 »Rückgabewerte von Funktionen«).

Bei verschachtelten `switch`- bzw. Schleifenkonstruktionen ist zu beachten, dass mit der `break`-Anweisung nur die innerste dieser Kontrollstrukturen verlassen wird:

```
while (Bedingung)
{
    ...
    for(Initialisierung; Bedingung; Aktualisierung)
    {
        ...
        if (Bedingung)
            break; // for-Schleife wird beendet
        ...
    }
    ...
}
```

Die break-Anweisung in der for-Schleife beendet diese, nicht aber die while-Schleife. Die Programmausführung setzt mit der ersten Anweisung nach der for-Schleife fort.

Die Tatsache, dass die break-Anweisung wie hier innerhalb eines if-Konstrukts steht, wirkt sich zwar dahingehend aus, dass sie nur dann ausgeführt wird, wenn die entsprechende if-Bedingung wahr ist, hat aber sonst keine weitere Bedeutung. Wie gesagt, kann sich ein break-Befehl allein auf eine switch-Anweisung oder auf ein Schleifenkonstrukt beziehen, nicht aber auf eine if-Anweisung. Mit anderen Worten: Eine break-Anweisung muss auf jeden Fall im Kontext einer switch-, while-, do while- oder for-Anweisung verwendet werden. Unter dieser Voraussetzung darf es selbstverständlich auch innerhalb einer if-Anweisung platziert werden.

Anders als break beendet die Anweisung

```
continue;
```

nur den aktuellen Schleifendurchlauf:

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

int main(void)
{
    for (int i = 1; i <= 5; i++)
    {
        if (i == 3)
            continue;
        cout << i << ' ';
    }
    return 0;
}
```

Ausgabe: 1 2 4 5

Wenn `i` den Wert 3 erreicht, wird die `if`-Bedingung wahr und der aktuelle Schleifendurchgang abgebrochen, das heißt, die Anweisung

```
cout << i << ' ';
```

kommt für diesen Wert von `i` nicht zur Ausführung. Die `for`-Schleife wird jedoch nicht verlassen, sondern die Programmausführung wird mit dem nächsten Schleifendurchgang fortgesetzt.

## 17.5 Gültigkeitsbereich von Variablen

Sie haben nun die Kontrollanweisungen von C++ kennen gelernt. Mit diesen sind in Ihren Programmen auch sehr komplexe Strukturen denkbar. Dass Variablen grundsätzlich an jeder Stelle im Code deklariert werden dürfen, ist Ihnen ebenso bekannt wie die Tatsache, dass dies vor ihrer ersten Verwendung geschehen muss. Was im Hinblick auf verschachtelte Blockstrukturen des Weiteren zu beachten ist, bezieht sich auf den Gültigkeitsbereich einer Variablen.

Es gilt:

- Eine Variable ist gültig in dem Block, in dem sie deklariert ist, sowie in allen inneren Blöcken.
- Eine Variable kann nur innerhalb ihres Gültigkeitsbereichs angesprochen werden.

Außerhalb ihres Gültigkeitsbereichs kann auf eine Variable nicht zugegriffen werden. Es verhält sich dann so, als sei die Variable gar nicht existent.

Solange Programme nur aus einem Block – dem von `main()` – bestehen, gibt es in dieser Hinsicht nichts zu berücksichtigen. Eine einmal vereinbarte Variable besitzt in diesem Block und damit im gesamten Code Gültigkeit. Mit anderen Worten: Nach ihrer Deklaration kann an jeder Stelle im Code auf sie zugegriffen werden:

```
...
int main(void)
{
    double meineVariable;
    ...
    cin >> meineVariable; // OK
    ...
    cout << meineVariable; // OK
    ...
    return 0;
}
```

Ist eine Variable jedoch in einem inneren Block deklariert, so kann diese nur in diesem Block – sowie in eventuell weiteren darin geschachtelten Blöcken – angesprochen werden:



```
...
int main(void)
{
    double meineVariable;
    ...
    while (Bedingung)
    {
        int inside = 1;
        meineVariable = inside; // OK
    }
    ...
    cout << meineVariable; // OK
    cout << inside; // FEHLER
    ...
    return 0;
}
```

### Die Zuweisung

```
meineVariable = inside; // OK
```

ist korrekt, da im Block der `while`-Schleife beide Variablen gültig sind. Die Variable `inside` ist in diesem Block vereinbart. Der Gültigkeitsbereich der Variablen `meineVariable` erstreckt sich ebenfalls auf den Block der `while`-Schleife, da dieser ja in den Block von `main()` geschachtelt, also ein innerer Block zu dem von `main()` ist.

Der Zugriff auf die Variable `inside` in der Anweisung

```
cout << inside; // FEHLER
```

dagegen ist fehlerhaft, da diese Variable hier nicht bekannt ist. Die Anweisung befindet sich ja in einem äußeren Block zu demjenigen, in dem `inside` deklariert ist.

Wie oben beschrieben, darf die Schleifenvariable auch im Kopf der `for`-Schleife deklariert werden. Der C++-Standard schreibt vor, dass eine so vereinbarte Variable auch nur innerhalb der `for`-Schleife gültig ist. Der Borland C++-Compiler und inzwischen auch der des Visual Studio halten sich daran.

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

int main(void)
{
    for (int i = 1; i <= 3; i++)
    {
        cout << i << ' ';
    }
}
```

```
cout << i; // FEHLER
return 0;
}
```

Falls Ihr Compiler einen Syntaxfehler in der Zeile `cout << i;` ausgibt, ist bei diesem diese Feinheit des Standards in puncto Gültigkeitsbereich richtig umgesetzt.

### Hinweis

Insofern Ihr Compiler hier vom Standard abweicht – was z.B. auf den C++-Compiler im Visual Studio 6 zutraf – geht – bezogen auf obiges Listing – die Laufvariable `i` mit dem Wert 4 aus der Schleife, da sie nach dem letzten Schleifendurchgang noch ein Mal hochgezählt wird. Die Aktualisierung findet also auch nach dem letzten Schleifendurchgang statt. Die Ausgabe lautet dann 1 2 3 4.

Wenn eine Variable im Block der `for`-Schleife vereinbart ist, hat diese in jedem Fall nur innerhalb des Schleifenblocks Gültigkeit:

```
...
int main(void)
{
    int i;
    for (i = 1; i <= 3; i++)
    {
        int innen;
        ...
    }
    ...
    cout << innen; // FEHLER
    ...
    return 0;
}
```

### Hinweis

Variablen, deren Gültigkeitsbereich sich auf einen Block erstreckt, bezeichnet man als lokal zu diesem Block, wobei jeweils der äußerste Block gemeint ist. Beispielsweise ist hier `innen` lokal zum Block der `for`-Schleife, die Variable `i` lokal zum Block von `main()`.

## 17.5.1 »Lebensdauer« von Variablen

Neben ihrem Gültigkeitsbereich kennzeichnet eine Variable deren »Lebensdauer«. Diese endet, wenn der vom Programm reservierte Speicherbereich wieder freigegeben wird. Der Programmierer sagt lapidar, die Variable wird gelöscht. Dies geschieht bei gewöhnlichen Variablen, wenn der Block, in dem sie deklariert sind, endgültig verlassen wird.

### Hinweis

Das Attribut »endgültig« ist insofern bedeutsam, als es vorkommen kann, dass von Block A nach Block B gesprungen wird und anschließend wieder zurück. In diesem Fall werden die zu A lokalen Variablen nicht gelöscht, wenn die Anweisungen in B ausgeführt werden, sondern erst, wenn – nach dem Rücksprung – alle Anweisungen in Block A abgearbeitet sind (mehr darüber erfahren Sie in Kapitel 20 »Funktionen«).

### Hinweis

Die lokalen Variablen, die Sie bisher verwendet haben, besitzen die Speicherklasse `auto`. Im Gegensatz zu diesen sind Variablen der Speicherklasse `static` während der gesamten Programmlaufzeit existent. Beide unterscheiden sich unter anderem dadurch, dass sie in verschiedenen Bereichen des Arbeitsspeichers verwaltet werden: Lokale Variablen werden dabei auf dem so genannten Stack (Stapelspeicher) angelegt, Variablen, die die Speicherklasse `static` besitzen, dagegen in einem globalen Namensbereich. Ein Beispiel für solche Variablen werden Sie gleich kennen lernen (Abschnitt 17.5.2 »static-Variablen«).

```
...
int main(void)
{
    int a;
    ...
    {
        int b;
    }
    ...
    return 0;
}
```

Nach dem oben Gesagten endet hier die Lebensdauer der Variablen `b` mit dem Verlassen des – willkürlich gesetzten – inneren Blocks, das heißt, der für diese Variable reservierte Speicherbereich wird freigegeben. Die Variable `a` wird erst gelöscht, wenn alle Anweisungen in `main()` abgearbeitet sind, also das Programmende erreicht ist.

Im obigen Codesegment ist es ausgeschlossen, dass der Block, in dem `b` deklariert ist, ein zweites Mal durchlaufen wird. Anders verhält es sich z.B. bei Schleifenblöcken. Diese können während eines Programmlaufs mehrere Male wiederholt werden. Hier ist zu berücksichtigen, dass nach einem Schleifendurchgang der Schleifenblock – endgültig – verlassen und mit jeder Wiederholung aufs Neue betreten wird. Das bedeutet für Variablen, die lokal zum Schleifenblock sind, dass diese nach jedem Schleifendurchgang gelöscht und wieder neu angelegt werden, wenn dieser Block mit dem nächsten Schleifendurchgang erneut betreten wird.

**Hinweis**

Beachten Sie, dass in diesem Sinne die Bedingungsprüfung außerhalb des Schleifenblocks erfolgt. Folglich darf eine im Schleifenblock deklarierte Variable dort nicht verwendet werden.

```
...
int main(void)
{
    ...
    while (zahl > 0) // FEHLER
    {
        int zahl;
        cin >> zahl;
    }
    ...
    return 0;
}
```

Demgemäß arbeitet folgendes Programm, das dazu gedacht sein soll, die eingegebenen Zahlen aufzuaddieren und jeweils die Zwischensumme anzuzeigen, logisch nicht korrekt:

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

int main(void)
{
    double zahl;
    for (int i = 0; i < 4; i++)
    {
        double sum = 0;
        cin >> zahl;
        sum += zahl;
        cout << "Summe: " << sum << endl;
    }
    return 0;
}
```

Ausgabe:

```
30
Summe: 30
10
Summe: 10
33
Summe: 33
27
Summe: 27
```

Als Endsumme wird nicht etwa 100, sondern 27 angezeigt, abgesehen davon, dass die Ausgabe der Zwischensummen ebenfalls fehlerhaft ist.

Die Ursache liegt einfach darin, dass die Variable `sum` nach jedem Schleifendurchgang gelöscht wird. Der zuvor gespeicherte Wert geht damit verloren. Mit jeder Wiederholung wird eine neue Variable `sum` angelegt und mit dem Wert 0 initialisiert.

```
double sum = 0;
```

Somit besitzt diese Variable nach der Zuweisung

```
sum += zahl;
```

genau den Wert von `zahl`, was dem zuletzt eingegebenen Zahlenwert entspricht.

### Hinweis

Es ist notwendig, die Variable `sum` mit 0 zu initialisieren, da ansonsten mit der Anweisung

```
sum += zahl;
```

auf einen undefinierten Wert zugegriffen wird (siehe Kapitel 10, Abschnitt 10.3 »Zuweisung«).

Das Problem lässt sich nun auf zweierlei Weise beheben. Die naheliegendste Lösung ist die, dass man die Variable `sum` im Block von `main()` deklariert. Dann behält sie nicht nur ihre Gültigkeit, sondern auch ihre Lebensdauer während der Ausführung der `while`-Schleife. Der Block von `main()` wird ja erst nach Programmende verlassen:

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

int main(void)
{
    double zahl, sum = 0;
    for (int i = 0; i < 4; i++)
    {
        cin >> zahl;
        sum += zahl;
        cout << "Summe: " << sum << endl;
    }
    return 0;
}
```

Ausgabe:

```
30
Summe: 30
10
Summe: 40
33
Summe: 73
27
Summe: 100
```

### Hinweis

Auch bei dieser Implementation ist die Initialisierung der Variablen `sum` notwendig, da diese ansonsten bei der ersten Ausführung der Anweisung `sum += zahl;` keinen definierten Wert besitzt.

## 17.5.2 static-Variablen

Eine zweite Möglichkeit zur Lösung des genannten Problems besteht darin, die lokale Variable `sum` als `static` zu vereinbaren. Dies geschieht, indem man das Schlüsselwort `static` in der Deklarationsanweisung vor den Datentypbezeichner schreibt:

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

int main(void)
{
    double zahl;
    for (int i = 0; i < 4; i++)
    {
        static double sum = 0;
        cin >> zahl;
        sum += zahl;
        cout << "Summe: " << sum << endl;
    }
    return 0;
}
```

Ausgabe:

```
30
Summe: 30
10
Summe: 40
33
Summe: 73
27
Summe: 100
```

Der Gültigkeitsbereich einer so deklarierten Variablen beschränkt sich zwar nach wie vor auf den Block, in dem sie deklariert ist, die Lebensdauer einer `static`-Variablen erstreckt sich jedoch auf das ganze Programm. Das heißt, eine `static`-Variable wird erst gelöscht, wenn das Programm beendet wird. Sie behält somit ihren Wert bei – es sei denn, dieser wird durch Neuzuweisungen verändert, was ja im Programm beabsichtigt ist.

Demzufolge wird bei jeder Ausführung der Anweisung

```
sum += zahl;
```

zu dem aktuellen Wert von `sum` der Wert von `zahl` hinzuaddiert.

### Hinweis

Im Gegensatz zu Variablen der Speicherklasse `auto` werden `static`-Variablen bei ihrer Vereinbarung automatisch mit dem Anfangswert 0 versehen. Von daher ist eine Initialisierung der Variablen `sum` in diesem Beispiel nicht notwendig. Sie dient hier nur der Verdeutlichung.

### Hinweis

Das Schlüsselwort `static` bewirkt unter anderem, dass die Deklarationsanweisung

```
static double sum = 0;
```

tatsächlich nur ein einziges Mal ausgeführt wird, beim ersten Schleifendurchgang und dann nie wieder.

## Hinweis

Im Normalfall decken sich Lebensdauer und Gültigkeitsbereich von Variablen, was auf static-Variablen, wie gesagt, nicht zutrifft. So ist die Lebensdauer der Variablen `sum` zwar global, ihr Gültigkeitsbereich aber lokal zum Block der `for`-Schleife. Außerhalb dieser kann `sum` daher nicht angesprochen werden.

```
...
int main(void)
{
    double zahl;
    for (int i = 0; i < 4; i++)
    {
        static double sum = 0;
        ...
    }
    ...
    cout << sum; // FEHLER
    ...
    return 0;
}
```

### 17.5.3 Namensgleichheit

In Kapitel 10 »Variablen« haben Sie erfahren, dass Bezeichner eindeutig sein müssen. Daher ist folgendes Codestück fehlerhaft, da mit der Vereinbarung von zwei Variablen gleichen Namens die Eindeutigkeit verloren ginge. Dass es sich dabei um Variablen unterschiedlichen Datentyps handelt, ändert an diesem Sachverhalt nichts:

```
...
int main(void)
{
    char eineVar;
    int eineVar; // FEHLER
    ...
    return 0;
}
```

Die genannte Regel bezieht sich allerdings nur auf ein und denselben Gültigkeitsbereich. Das heißt, Sie dürfen im Code Datenobjekte mit gleich lautenden Bezeichnern vereinbaren, sofern sich diese in verschiedenen Gültigkeitsbereichen befinden.

So kann jede C++-Funktion ihre eigenen lokalen Variablen besitzen, wobei es keine Rolle spielt, ob die gleichen Bezeichner für lokale Variablen anderer Funktionen gewählt sind (zu Funktionen siehe Kapitel 20). Dies ist auch unproblematisch, da in diesem Fall die beiden Funktionsblöcke – also die beiden Gültigkeitsbereiche – völlig voneinander getrennt liegen:



```
...
void func(void)
{
    int zahl;
    ...
}
int main(void)
{
    int zahl; // OK
    ...
    return 0;
}
```

Allerdings kann es vorkommen, dass sich zwei Gültigkeitsbereiche überschneiden. So sind z.B. in Abänderung des vorletzten Codestücks folgende Deklarationen zulässig, da jeder Block ja einen eigenen Gültigkeitsbereich beschreibt:

```
...
int main(void)
{
    char eineVar;
    {
        int eineVar; // OK
    }
    ...
    return 0;
}
```

Hier beanspruchen beide Variablen mit dem Bezeichner `eineVar` für den inneren Block Gültigkeit. Die eine, da sie in diesem Block deklariert ist, die andere, weil sich deren Gültigkeitsbereich grundsätzlich auf alle in den Funktionsblock von `main()` geschachtelten Blöcke erstreckt.

Für solche Fälle – wenn der eine Gültigkeitsbereich den anderen enthält – muss es irgendwie geregelt sein, welche Variable gültig ist – das heißt, welche Variable angesprochen wird –, wenn der entsprechende Bezeichner im Code des sich überschneidenden Teils verwendet wird:

```
...
int main(void)
{
    char eineVar = 7;
    {
        int eineVar = 11;
        cout << eineVar;
    }
}
```

```
...
    return 0;
}
```

### Hinweis

In Bezug auf derartige Namenskonflikte spricht der C++-Programmierer von Namensgleichheit.

Es gilt:

- Bei Namensgleichheit hat die im inneren Gültigkeitsbereich deklarierte Variable Vorrang.
- Die im äußeren Gültigkeitsbereich deklarierte Variable verliert für den inneren Gültigkeitsbereich vorübergehend ihre Gültigkeit.

Dass bedeutet hier, dass die zu `main()` lokale Variable `eineVar` im ganzen Funktionsblock gültig ist – bis auf den inneren Block. Was den Gültigkeitsbereich der im inneren Block vereinbarten Variablen `eineVar` angeht, ändert sich nichts. Demzufolge wird mit der Anweisung

```
cout << eineVar;
```

diese Variable angesprochen, daher die Ausgabe 11:

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

int main(void)
{
    char eineVar = 7;
    {
        int eineVar = 11;
        cout << eineVar;
    }
    return 0;
}
```

Ausgabe: 11

### Hinweis

Es sei darauf hingewiesen, dass es zu einem guten Programmierstil gehört, solche Namenskonflikte zu vermeiden. Für deren Auftreten besteht in der Regel keine Notwendigkeit.

Bis jetzt ging es in unseren Beispielprogrammen meist um die Eingabe zweier oder auch nur eines Wertes seitens des Benutzers. Oft ist es in Programmen jedoch notwendig, eine Vielzahl von Daten zu verarbeiten. Dann erweist sich der ausschließliche Gebrauch von Variablen sehr schnell als unbequem, und ab einer bestimmten Anzahl von Variablen verliert man im Code vermutlich gänzlich den Überblick, sofern sich diese in ein und demselben Gültigkeitsbereich befinden. Wobei in diesem Fall noch dafür zu sorgen ist, dass sich die einzelnen Variablenbezeichner voneinander unterscheiden.

Bestimmte Aufgabenstellungen lassen sich mit gewöhnlichen Variablen ohnehin nicht sinnvoll realisieren, z.B. wenn sehr viele Datenelemente verarbeitet werden sollen. Sie müssten dann für jedes Datenelement eine Variable deklarieren und dieser einen eindeutigen Namen geben, bei Tausenden zu speichernden Datenelementen ein hoffnungsloses Unterfangen.

Doch auch hierfür gibt es eine Lösung: Zur Verwaltung von größeren Datenmengen stellt C++ eine spezielle Datenstruktur zur Verfügung, das so genannte »Array«, das häufig auch als »Feld« oder »Variablenfeld« bezeichnet wird. Arrays erlauben es, mehrere Werte gleichen Datentyps unter einem einzigen Bezeichner anzusprechen.

## 18.1 Deklaration von Arrays

Eigentlich ist ein Array bereits ein komplexer Datentyp und – wenn man es ganz theoretisch betrachtet – eine Datenstruktur.

### Hinweis

Zu den komplexen Datentypen zählen auch die Strukturen und die Klassen. Schon jetzt der Hinweis: Alle komplexen Datentypen sind mehr oder weniger aus einfachen Datentypen zusammengesetzt.

Vom praktischen Standpunkt aus gesehen, ist ein Array eine Ansammlung von Variablen gleichen Datentyps, die nebeneinander im Speicher liegen. Allerdings werden diese »Elemente« genannt, was aber am Sachverhalt nichts ändert. Man spricht also nicht etwa von den Variablen eines Arrays, sondern von den Elementen eines Arrays.

Die Deklaration eines Arrays entspricht der einfacher Variablen, mit dem Unterschied, dass nach dem Bezeichner für das Array die Anzahl der Elemente in eckigen Klammern anzugeben ist.

Demzufolge vereinbart man mit der Anweisung

```
int meinArr[5];
```

ein Array aus fünf Elementen, die vom Datentyp `int` sind. Es können also fünf `int`-Werte gespeichert werden, die über den Bezeichner `meinArr` angesprochen werden. Beachten Sie aber, dass die einzelnen Elemente des soeben vereinbarten Arrays noch keinen definierten Wert besitzen.

Zur Angabe der Elementzahl kann jede Integerkonstante herangezogen werden, nicht aber eine Variable.

```
...
int main(void)
{
    int a;
    cin >> a;
    int meinArr[a]; // FEHLER
    ...
    return 0;
}
```

Erlaubt sind aber symbolische Konstanten:

```
...
int main(void)
{
    const int ANZAHL = 5;
    int meinArr[ANZAHL]; // OK
    ...
    return 0;
}
```

## 18.2 Mit Arrays arbeiten

Der Zugriff auf die einzelnen Elemente eines Arrays erfolgt, indem man hinter den Bezeichner des Arrays – ebenfalls in eckige Klammern – eine Indexangabe setzt, wobei der Index bei 0 beginnt. Das erste Element des Arrays `meinArr` wird also mit

```
meinArr[0]
```

das letzte Element mit

```
meinArr[4]
```

angesprochen.

### Achtung

Beachten Sie, dass bei der Deklaration eines Arrays die Elementzahl, nicht die Indexobergrenze anzugeben ist.

```
int meinArr[5];
```

Da der Index stets bei 0 beginnt, ist der letzte gültige Index eines Arrays, das aus  $n$  Elementen besteht,  $n-1$ . Hier im Beispiel ist es 4.

Um beispielsweise dem dritten Element des Arrays den Wert 99 zuzuweisen, schreiben Sie:

```
meinArr[2] = 99;
```

und um eine Benutzereingabe im ersten Element zu speichern

```
cin >> meinArr[0];
```

Wenn Sie nun die Summe der im ersten und im dritten Element enthaltenen Werte ausgeben möchten, notieren Sie einfach

```
cout << meinArr[0] + meinArr[2];
```

Zur Verdeutlichung hier das entsprechende Listing:

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

int main(void)
{
    int meinArr[5];
    meinArr[2] = 99;
    cout << "Bitte eine Zahl: ";
    cin >> meinArr[0];
    cout << meinArr[0] + meinArr[2];
    return 0;
}
```

Ausgabe:

```
Bitte eine Zahl: 1
100
```

Beachten Sie, dass die Elemente `meinArr[1]`, `meinArr[3]` und `meinArr[4]` hier immer noch keinen wohldefinierten Wert besitzen.

Ein Lesezugriff auf diese wäre demnach problematisch:

```
cout << meinArr[1]; // ???
```

### Tipp

Dem Leser sei nach wie vor empfohlen, sich das Bild von – hier fünf – Variablen vor Augen zu halten:

```
meinArr[0]
meinArr[1]
meinArr[2]
meinArr[3]
meinArr[4]
```

In diesem Sinne behält alles Gültigkeit, was Sie bis jetzt über Zuweisungen, Ausgaben, Eingaben usw. gehört haben.

Anders als bei Angabe der Elementzahl im Zuge der Deklaration eines Arrays (siehe oben), ist beim Zugriff auf ein Arrayelement die Verwendung von Variablenbezeichnern erlaubt:

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

int main(void)
{
    int meinArr[5], nummer = 3;
    meinArr[nummer] = 979;
    cout << meinArr[3];
    return 0;
}
```

Ausgabe: 979

## 18.3 Arrays in for-Schleifen durchlaufen

Arrays lassen sich bequem in for-Schleifen durchlaufen. Zum Zugriff auf die einzelnen Elemente des Arrays wird dabei die Schleifenvariable verwendet:

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

int main(void)
{
    int deinArr[1000];
```

```
for (int i = 0; i <= 999; i++)
{
    deinArr[i] = 1;
}
return 0;
}
```

Nach Austritt aus der for-Schleife besitzt jedes Arrayelement den Wert 1.

Beachten Sie, dass der letzte erlaubte Elementzugriff `deinArr[999]` ist. Die Schleifenbedingung muss daher

```
i <= 999
```

oder

```
i < 1000
```

lauten, nicht aber

```
i <= 1000 // unerlaubter Elementzugriff
```

Da in diesem Fall die for-Schleife für den Wert 1000 von `i` durchlaufen wird, würde das den Zugriff auf ein 1001. Element des Arrays bedeuten.

```
deinArr[1000] = 1;
```

### Achtung

Das Gefährliche an solchen Zugriffen liegt darin, dass sie vom Compiler nicht als Syntaxfehler erkannt werden. In der Regel erfolgt auch keine Warnmeldung. Dafür stellt sich beizeiten ein Laufzeitfehler ein (Programmabsturz).

In folgendem Beispiel wird das Array `meinArr` mit Benutzereingaben gefüllt und anschließend wieder ausgegeben:

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

int main(void)
{
    int meinArr[5];
    int i = 0;
    for (i = 0; i < 5; i++)
    {
        cin >> meinArr[i];
    }
}
```

```

}
cout << "Du hast die Zahlen";
for (i = 0; i < 5; i++)
{
    cout << ' ' << meinArr[i];
    if (i < 3)
        cout << ',';
    if (i == 3)
        cout << " und";
}
cout << " eingegeben." << endl;
return 0;
}

```

Ausgabe:

```

2007
-59
1999
44
-23
Du hast die Zahlen 2007, -59, 1999, 44 und -23 eingegeben.

```

Folgendes Programm berechnet den Durchschnittswert der Eingaben:

```

#include <iostream>
using namespace std;

int main(void)
{
    double noten[5], sum = 0;
    cout << "Noten: " << endl;
    int i;
    for (i = 0; i < 5; i++)
    {
        cin >> noten[i];
    }
    for (i = 0; i < 5; i++)
    {
        sum += noten[i];
    }
    cout << "Durchschnittsnote: " << sum / 5 << endl;
    return 0;
}

```



Ausgabe:

```
Noten:
3
5
3
3
2
Durchschnittsnote: 3.2
```

## 18.4 Initialisierung von Arrays

Es ist auch möglich, Arrays bei der Deklaration zu initialisieren, indem man die einzelnen Werte durch Kommata getrennt in geschweifte Klammern setzt:

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

int main(void)
{
    char buchstaben[7] = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
    for (int i = 0; i < 7; i++)
    {
        cout << buchstaben[i];
    }
    return 0;
}
```

Ausgabe: ABCDEFG

In diesem Fall ist es sogar erlaubt, die Elementzahl in den eckigen Klammern wegzulassen. Diese ergibt sich dann aus der Anzahl der Initialisierungswerte. So wird beispielsweise mit der Deklarationsanweisung

```
double zahlen[] = {2.34, 45.0, -99.7, 75.3, 421.99};
```

ein Array aus fünf Elementen mit dem Bezeichner `zahlen` vom Datentyp `double` vereinbart:

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

int main(void)
{
    double zahlen[] = {2.34, 45.0, -99.7, 75.3, 421.99};
```

```

    for (int i = 0; i < 5; i++)
    {
        cout << zahlen[i] << ' ';
    }
    return 0;
}

```

Ausgabe: 2.34 45 -99.7 75.3 421.99

Falls – vorausgesetzt, Sie geben die Elementzahl in den eckigen Klammern an – die Anzahl der in der Initialisierungsliste notierten Werte kleiner ist als die Elementzahl des Arrays, wird der Rest bei numerischen Werten mit 0 initialisiert, bei char-Werten mit dem Nullbyte ('\0'), das im ASCII-Code die Ordnungsnummer 0 besitzt:

```

#include <iostream>
using namespace std;

int main(void)
{
    int nummern[5] = {1, 2, 3};
    for (int i = 0; i < 5; i++)
    {
        cout << nummern[i] << ' ';
    }
    return 0;
}

```

Ausgabe: 1 2 3 0 0

Auf diese Weise ist es möglich, auch sehr große Arrays in einer sehr kurzen Schreibweise zu initialisieren. So besitzen nach der Anweisung

```
int vieleZahlen[32000] = {0};
```

alle 32000 Elemente des Arrays `vieleZahlen` den Wert 0:

```

#include <iostream>
using namespace std;

int main(void)
{
    int vieleZahlen[32000] = {0};
    cout << vieleZahlen[31997];
    return 0;
}

```

Ausgabe: 0

**Hinweis**

Es sei jedoch darauf hingewiesen, dass der C++-Standard dieses Verhalten nicht vorschreibt. Von daher ist es besser, diese Initialisierung in einer `for`-Schleife durchzuführen, um ganz sicherzugehen:

```
for (int i = 0; i < 32000; i++)
    vieleZahlen[i] = 0;
```

Falls ein anderer Initialisierungswert gewünscht ist als 0 – beispielsweise 1 –, müssen Sie auf jeden Fall so verfahren.

### 18.4.1 Die Größe eines Arrays bestimmen mit dem `sizeof`-Operator

Mit »Größe eines Arrays« ist im Allgemeinen – wie auch hier – nicht die Größe in Byte, sondern die Anzahl der Elemente gemeint. Falls erforderlich, können Sie diese mit dem `sizeof`-Operator im Code feststellen. Dies kann vor allem im Zusammenhang mit dynamischen Arrays von Nutzen sein. Was zu tun ist, um Speicher dynamisch anzufordern, das heißt zur Laufzeit, erfahren Sie in Kapitel 28 »Zeiger«.

**Referenz**

Zum `sizeof`-Operator siehe auch Kapitel 12, Abschnitt 12.1.1 »`sizeof`-Operator« sowie Kapitel 15, Abschnitt 15.3.1 »Ausdrücke als Operanden des `sizeof`-Operators«.

Die Methode ist denkbar einfach:

Um die Gesamtgröße eines Arrays in Byte zu erhalten, ist dem `sizeof`-Operator der Bezeichner dieses Arrays zu übergeben, also z.B.

```
sizeof(arr)
```

Dass es sich mit `arr` um den Bezeichner eines Arrays handelt, setzen wir hier voraus. Nehmen wir des Weiteren an, der Datentyp des Arrays sei *datentyp*, wobei für *datentyp* natürlich der tatsächliche Datentyp einzusetzen ist. Dann ist ein einzelnes Element dieses Arrays

```
sizeof(datentyp)
```

groß. Um die Elementzahl des Arrays zu erhalten, müssen Sie die Gesamtgröße des Arrays durch die Elementgröße teilen:

```
sizeof(arr) / sizeof(datentyp)
```

Beispiel:

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

int main(void)
{
    double meineZahlen[] = {12.2, 999.7, -22.3,
                             312.765, 11.8, 543.77,
                             -96537.002, 55.8};
    cout << "Das Array meineZahlen hat "
          << sizeof(meineZahlen) / sizeof(double)
          << " Elemente";
    return 0;
}
```

Ausgabe:

```
Das Array meineZahlen hat 8 Elemente
```

## 18.5 Mehrdimensionale Arrays

Es ist auch möglich, mehrdimensionale Arrays zu erzeugen. In diesem Fall ist für jede Dimension die entsprechende Größe anzugeben. Beispielsweise vereinbart man mit der Anweisung

```
int dreid[10][9][10];
```

ein dreidimensionales Array von der Größe  $10 \times 9 \times 10$  zur Aufnahme von insgesamt 900 Integerwerten. Um ein bestimmtes Element des Arrays anzusprechen, ist es natürlich erforderlich, für jede Dimension einen Index anzugeben:

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

int main(void)
{
    int dreid[10][9][10];
    dreid[0][0][1] = 77;
    cout << dreid[0][0][1];
    return 0;
}
```

Ausgabe: 77

Vermutlich werden Sie Arrays mit drei oder mehr Dimensionen weniger häufig benötigen. Die Verwendung von zweidimensionalen Arrays kann jedoch mitunter sehr nützlich sein. So z.B., wenn Sie in Ihren Programmen eine Tabelle oder – für Spiele – ein Schach- bzw. Damebrett wiedergeben wollen.

```
int schachbrett[8][8];
```

So könnten Sie für ein Damespiel im Programm für weiße Steine den Wert 1, für schwarze den Wert 2 und für leere Felder den Wert 0 vergeben. Angenommen, die erste Dimension gibt die Zeilen wieder und die zweite die Spalten, wobei die Zählung jeweils bei 0 beginnt. Das heißt, die erste Zeile soll mit dem Index 0 bezeichnet werden. Ebenso wird das jeweils ganz links befindliche Feld innerhalb einer Zeile über den Index 0 adressiert.

Entsprechend setzen Sie z.B. mit der Anweisung

```
schachbrett[7][7] = 2;
```

einen schwarzen Stein in die rechte obere Ecke des Brettes.

Um ein entsprechendes zweidimensionales Array mit Null zu initialisieren, bietet sich eine verschachtelte for-Schleife an:

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

int main(void)
{
    int schachbrett[8][8];
    for (int i = 0; i < 8; i++)
    {
        for (int k = 0; k < 8; k++)
        {
            schachbrett[i][k] = 0;
        }
    };
    return 0;
}
```

Beachten Sie, dass für jeden Wert von *i* die innere for-Schleife einmal vollständig durchlaufen wird. Natürlich müssen in den beiden Schleifen zwei verschiedene Laufvariablen verwendet werden, im Beispiel *i* und *k*.

Allerdings ist es ebenso wie bei eindimensionalen Arrays erlaubt, mehrdimensionale Arrays durch eine in {} zu setzende Liste von Werten zu initialisieren:

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

int main(void)
{
    int zahlen[3][2][2] = {-11, 66, 0, -9044, 5, 6, 1,
                          -7, 9, 10, 101, -23};
    cout << zahlen[0][1][1];
    return 0;
}
```

Ausgabe: -9044

Um die Deklaration übersichtlicher zu gestalten, dürfen Sie sogar – allerdings nur in Bezug auf die erste Dimension – zusätzliche Klammern ({} ) verwenden. Die Elemente der ersten Dimension stehen dann jeweils in eigenen Klammernpaaren:

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

int main(void)
{
    int zahlen[3][2][2] = {
        {-11, 66, 0, -9044},
        {5, 6, 1, -7},
        {9, 10, 101, -23}
    };
    cout << zahlen[0][1][1];
    return 0;
}
```

Ausgabe: -9044

Auch bei mehrdimensionalen Arrays werden freie Stellen mit 0 (bei Integerarrays) bzw. mit dem Nullbyte '\0' (bei char-Arrays) aufgefüllt, falls nur ein Initialisierungswert angegeben ist. So wird mit der Deklarationsanweisung

```
int zahlen[3][2][2] = {1};
```

das Element `zahlen[0][0][0]` mit dem Wert 1 versehen, alle anderen mit dem Wert 0.

Falls die Initialisierungswerte der ersten Dimension explizit in weitere {} gefasst sind, gilt dies sogar speziell für eine bestimmte Zeile:

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

int main(void)
{
    int zahlen[3][2][2] = {
        {-11, 66},
        {-499, 6, 1, -7},
        {9, 10, 101, -23}
    };
    cout << zahlen[0][1][1] << endl;
    cout << zahlen[1][0][0];
    return 0;
}
```

Ausgabe:

```
0
-499
```

Beachten Sie, dass bei der Deklaration von mehrdimensionalen Arrays allenfalls die Größenangabe für die erste Dimension unterbleiben darf:

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

int main(void)
{
    int tabelle[][3] = {
        {17, 99, 48},
        {91, 55, 84},
        {48, 97, 19},
    };
    int i, k;
    for (i = 0; i < 3; i++)
        for (k = 0; k < 3; k++)
        {
            if (k % 3 == 0)
                cout << endl;
            cout << tabelle[i][k] << ' ';
        }
    return 0;
}
```

Ausgabe:

```
17 99 48
91 55 84
48 97 19
```

### Hinweis

Da ein Schleifenkonstrukt vom Compiler gewissermaßen wie eine einzige Anweisung behandelt wird (Kapitel 16, Abschnitt 16.2.1 »Verschachteln von Kontrollstrukturen«), ist es im obigen Listing für die äußere for-Schleife nicht zwingend notwendig, Klammern ({} ) zu setzen (zur for-Schleife siehe Kapitel 17, Abschnitt 17.3 »Die for-Schleife«).

Dementsprechend würde sich Ihr Compiler weigern, den Quellcode zu übersetzen, falls im Beispiel die Klammern für die zweite Dimension oder gar für beide Dimensionen leer bleiben:

```
...
int main(void)
{
    int tabelle[3][] = { // FEHLER
        {17, 99, 48},
        {91, 55, 84},
        {48, 97, 19},
    };
    ...
    return 0;
}
```



# 19

## Strings

Da Ihnen Arrays inzwischen bekannt und Strings eigentlich spezielle char-Arrays sind, ist es an der Zeit, sich mit diesem Thema zu beschäftigen. Sie erfahren in diesem Kapitel einiges über den Gebrauch von Strings im Allgemeinen sowie darüber, was es mit C-Strings auf sich hat bzw. worin sich diese von Variablen der Klasse `string` unterscheiden. Praktisch nebenbei lernen Sie eine Reihe nützlicher Funktionen kennen.

### Hinweis

Es sei an dieser Stelle daran erinnert, dass die Bezeichnung Strings ein Sammelbegriff sowohl für Zeichenketten als auch für `string`-Variablen ist.

## 19.1 Wie Zeichenketten dargestellt werden

Wie Stringliterals im Code aussehen, ist Ihnen natürlich inzwischen geläufig. Wie werden diese nun aber im Computer dargestellt? Dazu müssen Sie wissen, dass Strings immer Platz im Arbeitsspeicher beanspruchen, auch wenn diese als konstante Werte in Form von Literalen im Code auftreten. Das heißt, mit einer Anweisung wie

```
cout << "C++";
```

sorgt der Compiler dafür, dass im Arbeitsspeicher des Computers tatsächlich eine entsprechende Folge von Zeichen abgelegt wird – dies natürlich in binärer Form –, und teilt `cout` im Anschluss daran mit, wo sich die auszugebenden Zeichen befinden (dazu gleich mehr im nächsten Abschnitt).

Nun, wir haben an anderer Stelle bereits angedeutet, dass es sich bei Zeichenketten eigentlich um char-Arrays handelt. Das Besondere daran ist, dass die eigentliche Zeichenkette mit dem Nullzeichen (`\0`) – auch Nullbyte genannt – abgeschlossen wird. Dieses wird automatisch angefügt.

Demnach wird der String

```
"C++"
```

im Speicher wie folgt dargestellt:

'C'	'+'	'+'	'\0'
-----	-----	-----	------

Abbildung 19.1: Interne Darstellung der Zeichenkette "C++"

Das vom Compiler implizit erzeugte char-Array mit dem Endezeichen '\0' ist in diesem Fall namenlos. Natürlich lässt sich der gleiche String auch in einem vom Programmierer vereinbarten char-Array speichern:

```
...
int main(void)
{
    char meinString[4];
    meinString[0] = 'C';
    meinString[1] = '+';
    meinString[2] = '+';
    meinString[3] = '\0';
    ...
    return 0;
}
```

bzw.

```
...
int main(void)
{
    char meinString[4] = {'C', '+', '+', '\0'};
    ...
    return 0;
}
```

Wobei man im letzten Fall die Angabe der Elementzahl, wie gesagt, auch weglassen darf:

```
char meinString[] = {'C', '+', '+', '\0'};
```

Es geht sogar noch bequemer, indem man das Array einfach mit der Zeichenkette initialisiert:

```
...
int main(void)
{
    char meinString[4] = "C++";
    ...
    return 0;
}
```

Achten Sie in diesem Fall aber darauf – unter Berücksichtigung des Endezeichens '\0' –, genügend Platz für das Array bzw. die Zeichenkette zu reservieren. Allerdings kann auch hier die Angabe der Elementzahl unterbleiben, sodass z.B. mit der Anweisung

```
char comic[] = "Asterix";
```

ein char-Array mit acht Elementen erzeugt wird, wobei das letzte Element das Zeichen '\0' enthält.

Auf die einzelnen Elemente des Arrays kann danach in der gewohnten Weise mittels Indexangabe zugegriffen werden:

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

int main(void)
{
    char comic[] = "Asterix";
    cout << comic[0] << comic[1];
    cout << comic[2] << comic[3];
    return 0;
}
```

Ausgabe: Aste

## 19.2 Arrays und Adressen

Wie ist es nun aber, wenn Sie die im Array gespeicherte Zeichenkette nicht Zeichen für Zeichen, sondern auf einmal ausgeben wollen? Die besondere Art der internen Darstellung mit dem Endezeichen '\0' deutet schließlich irgendwie darauf hin, dass dies möglich ist.

In diesem Fall übergeben Sie dem Objekt cout bzw. dem Operator << einfach die Adresse des char-Arrays. Dazu müssen Sie zum einen wissen, dass Arrays grundsätzlich über ihre Adressen verwaltet werden, zum anderen, dass der Name eines Arrays für dessen Adresse steht. Wobei mit Adresse die erste Speicherzelle des ersten Elements gemeint ist. Um dies deutlich zu machen, könnte man auch von der Anfangsadresse eines Arrays sprechen.

```
int arr[5] = {12, 32, 44, 19, 7};
```

Dementsprechend bezeichnet

```
arr
```

die Anfangsadresse des mit der obigen Anweisung vereinbarten Arrays. Tatsächlich lässt sich diese bzw. der entsprechende Hexadezimalwert auch im Programm anzeigen, indem man cout diesen Ausdruck übergibt:

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

int main(void)
{
    int arr[5] = {12, 32, 44, 19, 7};
    cout << arr;
    return 0;
}
```

Ausgabe: 0012FF6C

Die Ausgabe wird natürlich von Fall zu Fall unterschiedlich ausfallen, je nachdem, an welcher Speicheradresse das Array während des aktuellen Programmlaufs gerade angelegt wurde.

Somit heißt ein Ausdruck wie z.B.

```
arr[2]
```

nichts anderes als

»gehe zur Adresse `arr`, dann  $2 * 4$  Bytes weiter und greife auf den dortigen Inhalt zu«.

Dabei bezeichnet man die Entfernung eines Elements in Byte von der Anfangsadresse des Arrays als den Offset dieses Elements. Der Offset des dritten Arrayelements (`arr[2]`) beträgt bei dem oben deklarierten Integerarray also  $2 * 4 = 8$  Byte, wobei wir hier für den Datentyp `int` eine Größe von 4 Byte voraussetzen.

### Hinweis

Ist der Datentyp `int` 2 Byte groß, dann ist der Offset des dritten Elements entsprechend  $2 * 2 = 4$  Byte. Entsprechendes gilt natürlich, falls das Array als `short` deklariert ist. Verallgemeinernd gilt für den Offset eines Arrayelements:

*Offset = Index \* Datentypgröße*

Oben genannte Möglichkeit, mithilfe des `cout`-Objekts die Adresse eines Arrays auszugeben, gilt natürlich auch für alle anderen Datentypen, mit Ausnahme von `char`-Arrays. Was diese angeht, verläuft die Kommunikation mit dem `cout`-Objekt in anderer Weise. Wenn Sie `cout` die Adresse eines `char`-Arrays übergeben, weisen Sie das `cout`-Objekt damit an, alle Zeichen, die sich ab dieser Adresse befinden, bis zum ersten Nullbyte auszugeben:

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

```
int main(void)
{
    char comic[] = "Asterix und Obelix";
    cout << comic;
    return 0;
}
```

Ausgabe: Asterix und Obelix

Dabei macht es keinen Unterschied, ob das Array tatsächlich größer ist als für die Aufnahme eines bestimmten Strings notwendig:

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

int main(void)
{
    char comic[77] = "Asterix und Obelix";
    cout << comic;
    return 0;
}
```

Ausgabe: Asterix und Obelix

Ebenfalls keine Rolle spielt es, ob sich im Array nach dem ersten Nullbyte noch weitere Zeichen befinden, sogar wenn es sich dabei um weitere Nullbytes handelt. Wenn also in der Zeichenkette

```
"C++"
```

beispielsweise das zweite Zeichen mit dem Wert `'\0'` überschrieben wird, dann lautet die Ausgabe

```
C
```

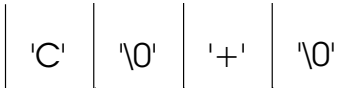
falls `cout` die Adresse des Arrays übergeben wird:

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

int main(void)
{
    char deinArray[] = "C++";
    cout << deinArray << endl;
    deinArray[1] = '\0';
    cout << deinArray << endl;
    return 0;
}
```

Ausgabe:

```
C++
C
```



deinArray

**Abbildung 19.2:** Zwei Nullbytes im char-Array

Natürlich ist der Elementzugriff mittels Index nach wie vor auf alle Zeichen des Arrays möglich:

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

int main(void)
{
    char deinArray[] = "C++";
    cout << deinArray << endl;
    deinArray[1] = '\0';
    cout << deinArray << endl;
    cout << deinArray[2];
    return 0;
}
```

Ausgabe:

```
C++
C
+
```

Prinzipiell ist es zwar ebenfalls möglich, cin die Adresse eines char-Arrays zu übergeben. Das Nullbyte wird auch in diesem Fall der Eingabe automatisch hinzugefügt:

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

int main(void)
{
    char deinArray[30];
    cin >> deinArray;
    cout << deinArray;
    return 0;
}
```

Ausgabe:

```
Asterix
Asterix
```

Allerdings können sich Laufzeitfehler einstellen, wenn die Eingabe größer ist, das heißt mehr Zeichen enthält, als das Array aufnehmen kann. Aus diesem Grunde sollten Sie es vorziehen, `cin` in dieser Form mit `string`-Variablen zu verwenden:

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

int main(void)
{
    string eingabe;
    cin >> eingabe;
    cout << eingabe;
    return 0;
}
```

Ausgabe:

```
Asterix und Obelix
Asterix
```

Wobei zu berücksichtigen ist, dass die Eingabe nur bis zum ersten Zwischenraumzeichen eingelesen wird, wie in der Ausgabe des obigen Beispiels zu sehen – die überzähligen Zeichen landen im Eingabepuffer. Dies gilt natürlich auch, falls `cin` die Adresse eines `char`-Arrays übergeben wird.

## 19.2.1 `cin.get()`

Um eine Eingabe zeichenweise und unter Berücksichtigung von Zwischenraumzeichen in ein `char`-Array einzulesen, eignet sich die Methode `get()` des `cin`-Objekts. Aufgerufen wird sie mit

```
cin.get()
```

wobei dieser Ausdruck für den Rückgabewert – das eingelesene Zeichen – steht. Folglich kann man

```
char c;
c = cin.get();
```

bzw.

```
char arr[11];
arr[0] = cin.get();
```


schreiben, um ein eingegebenes Zeichen einer char-Variablen bzw. dem Element eines char-Arrays zuzuweisen. Im folgenden Beispiel wird eine Benutzereingabe Zeichen für Zeichen in ein char-Array mit dem Bezeichner name eingelesen:

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

int main(void)
{
    char name[51];
    int i = 0;
    do
    {
        name[i] = cin.get();
    } while (name[i++] != '\n' && i < 50);
    name[i-1] = '\0';
    cout << name << endl;
    return 0;
}
```

Ausgabe:

```
Hans Meier
Hans Meier
```

Die Eingabe findet in der while-Schleife statt, wobei die Variable *i* zur Indizierung der einzelnen Arrayelemente herangezogen wird. Die Schleife bricht nach Eingabe von 50 Zeichen oder nach Drücken von  ab:

```
...
} while (name[i++] != '\n' && i < 51);
...
```



Beachten Sie, dass die Variable *i* erst nach Auswertung der Schleifenbedingung hochgezählt wird (Postfixnotation). Nach Austritt aus der Schleife muss an die eingegebenen Zeichen noch ein Nullbyte angehängt werden. Hierbei ist zu beachten, dass das zuletzt eingegebene Zeichen im Normalfall das Zeilenvorschubzeichen '\n' ist. Um dieses zu überschreiben, muss das entsprechende Arrayelement mit *i-1* indiziert werden:

```
name[i-1] = '\0';
```

Falls die while-Schleife abbricht, weil der Benutzer mehr als 50 Zeichen eingegeben hat, wird natürlich auf diese Weise das 51. Zeichen mit dem Nullbyte überschrieben.




Der Benutzer wird jedoch nicht daran gehindert, mehr als 51 Zeichen einzugeben. Diese erscheinen auch als Echo auf dem Bildschirm. Mit dem zweiten Teilausdruck der Schleifenbedingung ( $i < 51$ ) wird lediglich unterbunden, dass mehr Zeichen in das Array geschrieben werden, als für diesen Speicherplatz reserviert ist.

Wie gesagt, liest jedes `cin.get()` ein Zeichen ein. Die Eingabe mit `cin.get()` endet jedoch wie bei `cin >>` grundsätzlich erst mit dem Drücken von . Falls der Benutzer mehrere Zeichen eingegeben hat, verbleiben die übrigen Zeichen im Eingabepuffer und werden gegebenenfalls vom nächsten `cin.get()` dort abgeholt. In diesem Fall erhält der Benutzer keine Möglichkeit zur Eingabe. `cin.get()` liest also – wie das bei `cin >>` ja auch der Fall ist – entweder direkt vom Eingabepuffer oder – falls dieser leer ist – von der Tastatur (wobei allerdings jede Benutzereingabe nach Drücken von  zunächst einmal im Eingabepuffer landet).

Auf unser Beispiel bezogen heißt das:

Nach dem Drücken der -Taste seitens des Benutzers werden die zuvor eingegebenen Zeichen mit wiederholter Ausführung von

```
name[i] = cin.get();
```

nacheinander vom Eingabepuffer abgeholt und im Array `name` gespeichert. Der erste Teilausdruck der Schleifenbedingung (`name[i++] != '\n'`) sorgt dafür, dass der Benutzer nach einmaligem Drücken der -Taste keine weitere Möglichkeit zur Eingabe erhält. Überzählige Zeichen verbleiben im Eingabepuffer.

Das eben Dargelegte wird besonders deutlich, wenn Sie `cin.get()` im Code zweimal hintereinander verwenden:

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

int main(void)
{
    char zeichen;
    zeichen = cin.get();
    zeichen = cin.get();
    cout << zeichen;
    return 0;
}
```

Ausgabe:

```
xy
y
```

bzw.

```
x
```

Wenn Sie bei der ersten Eingabemöglichkeit die Zeichen »x« und »y« nacheinander eingeben, also in der Form

```
xy
```



dann nimmt sich das erste `cin.get()` das erste Zeichen – also 'x' – aus dem Eingabepuffer. Dieses wird mit der ersten Anweisung

```
zeichen = cin.get();
```

in der Variablen `zeichen` gespeichert. Da sich im Eingabepuffer nunmehr das Zeichen 'y' – sowie das Zeichen '\n' – befindet, erhält der Benutzer keine weitere Möglichkeit zur Eingabe. Mit der nächsten Anweisung

```
zeichen = cin.get();
```

wird nun das Zeichen 'y' aus dem Eingabepuffer geholt, der Variablen `zeichen` zugewiesen und damit der zuvor gespeicherte Wert 'x' überschrieben.

Falls vom Benutzer nur genau ein Zeichen – nehmen wir wiederum das Zeichen »x« an – eingegeben wird, erhält er dennoch keine zweite Eingabemöglichkeit. Der Grund dafür ist, dass nach Eingabe von »x« und dem die Eingabe abschließenden , wie oben angedeutet, zwei Zeichen im Eingabepuffer landen – nämlich 'x' und das mit der Taste  verbundene Zeilenvorschubzeichen '\n'. Folglich holt sich das erste `cin.get()` das 'x', das zweite das '\n'. Dieses wird dann in der Variablen `zeichen` gespeichert. Daher bewirkt in diesem Fall die Anweisung

```
cout << zeichen;
```

allein einen Zeilenvorschub.

### Hinweis

Während `cin >>` Zwischenraumzeichen ignoriert, berücksichtigt `cin.get()` diese beim Einlesen der Zeichen.

Übrigens können Sie eine Funktion oder eine Methode auch verwenden, ohne deren Rückgabewert weiterzuverarbeiten. So liest die Anweisung

```
cin.get();
```

ein Zeichen von der Tastatur bzw. dem Eingabepuffer, ohne etwas damit zu machen, also z.B. einer Variablen zuzuweisen. Dies lässt sich hier ausnutzen, um überschüssige Zeichen vom Eingabepuffer zu entfernen. Zuverlässig geschieht dies mit folgender Konstruktion:

```
while (cin.get() != '\n');
```

Die einzige Anweisung der `while`-Schleife ist die leere Anweisung (`;`). Dies ist aus syntaktischen Gründen notwendig, da die `while`-Schleife mindestens eine Anweisung benötigt. Die Methode `cin.get()`, um die es hier eigentlich geht, wird in der Schleifenbedingung notiert – was an und für sich nichts ungewöhnliches ist, da diese ja einen Rückgabewert besitzt. Dieser wird auf Ungleichheit mit dem Zeilenvorschubzeichen geprüft. Das bedeutet, die Schleife wird so lange ausgeführt, also Zeichen vom Eingabepuffer gelesen, bis dieser vollständig geleert ist – `'\n'` ist ja logischerweise das zuletzt in den Puffer gelangende Zeichen.

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

int main(void)
{
    char zeichen;
    zeichen = cin.get();
    while (cin.get() != '\n');
    zeichen = cin.get();
    cout << zeichen;
    return 0;
}
```

Ausgabe:

```
xyzabc123
V
V
```

Nun erhält der Benutzer auf jeden Fall eine zweite Eingabemöglichkeit.

## 19.2.2 `cout.put()`

Das Gegenstück zur `get()`-Methode des `cin`-Objekts ist die `put()`-Methode des `cout`-Objekts. Diese gibt ein einzelnes Zeichen aus, das der Methode beim Aufruf zu übergeben ist:

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

int main(void)
{
    char name[] = "Hans Meier";
    int i = 0;
    while (name[i] != ' ')
        cout.put(name[i++]);
    return 0;
}
```

Ausgabe:


Hans

In der `while`-Schleife werden die im Array `name` enthaltenen Zeichen bis zum ersten Leerzeichen (exklusive) ausgegeben. Das im Array enthaltene Nullbyte spielt hier natürlich keine Rolle.

### 19.2.3 Die Funktionen `getch()`, `getche()`, `kbhit()`

Da wir gerade dabei sind, hier noch einige weitere Funktionen, die zwar aus alten C-Zeiten stammen und nicht zum ANSI C++-Standard gehören, jedoch trotzdem sehr nützlich sind. Die Funktion `getch()` liest ein Zeichen direkt von der Tastatur, also ungepuffert. Rückgabewert der Funktion ist das eingelesene Zeichen.

Weitere Besonderheiten:

- Die Eingabe endet automatisch, ohne Bestätigung durch .
- Es erfolgt kein Wiederhall auf dem Bildschirm. Der Benutzer bekommt also von seiner Eingabe nichts zu sehen.

Sie können den Rückgabewert von `getch()` natürlich wie gewohnt einer Variablen zuweisen.

```
char zeich;
zeich = getch();
```

Vielfach wird `getch()` aber schlicht eingesetzt, um einen Haltepunkt zu programmieren:

```
#include <iostream>
#include <conio.h>
using namespace std;

int main(void)
{
    // ...
    cout << "Weiter mit beliebiger Taste";
    getch();
    cout << endl << "Weiter geht's";
    // ...
    return 0;
}
```

#### Achtung

Um die Funktionen `getch()`, `getche()` und `kbhit()` verwenden zu können, müssen Sie die Headerdatei `conio.h` mit der `#include`-Direktive einbinden.

Der einzige Unterschied der Funktion `getche()` gegenüber der Funktion `getch()` ist, dass `getche()` ein Echo des eingegebenen Zeichens auf dem Bildschirm wiedergibt.

Die Funktion `kbhit()` prüft, ob irgendeine Taste gedrückt wurde. Ist das der Fall, gibt die Funktion einen Integerwert ungleich 0 zurück, ansonsten 0:

```
#include <iostream>
#include <conio.h>
using namespace std;

int main(void)
{
    int k = 0, i;
    while (kbhit() == 0)
    {
        for (i=0; i< 100000000; i++) ;
        cout << ++k << ' ';
    }
    return 0;
}
```

Ausgabe: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 ...

Das Programm schreibt so lange fortlaufende Zahlen auf den Bildschirm, bis der Benutzer durch Drücken einer beliebigen Taste diesen Vorgang abbricht. Die innere `for`-Schleife

```
for (i=0; i< 100000000; i++) ;
```

erfüllt dabei die Funktion einer Warte- bzw. Verzögerungsschleife.

### Hinweis

Statt

```
kbhit() == 0
```

lässt sich in der Bedingung der `while`-Schleife auch kürzer

```
!kbhit()
```

formulieren, da numerische Werte in Ausdrücken implizit in den Datentyp `bool` konvertiert werden (Kapitel 16, Abschnitt 16.2.2 »Konvertierung in logischen Ausdrücken«).

## 19.3 Funktionen zur Stringverarbeitung

Für den C++-Programmierer gibt es grundsätzlich zwei Wege, mit Zeichenketten umzugehen: mit char-Arrays oder mit string-Variablen. Da es in der Programmiersprache C nur Erstere gibt, werden Zeichenketten, die in char-Arrays gespeichert sind, auch als »C-Strings« bezeichnet, um den Unterschied zu den string-Variablen deutlich zu machen.

### Hinweis

Gemäß dieser Definition handelt es sich bei Zeichenkettenliteralen ebenfalls um C-Strings, da deren Auftreten im Code regelmäßig mit Reservierung von Speicher verbunden ist. So wird z.B. mit Ausführung der Anweisung

```
cout << "XYZ";
```

die Zeichenkette "XYZ" gleichfalls in einem – wenn auch temporären, namenlosen und damit im weiteren Code nicht zugreifbaren – char-Array mit dem Nullbyte als Endezeichen gespeichert (Abschnitt 19.1 »Wie Zeichenketten dargestellt werden«).

C++ stellt zur Verarbeitung von C-Strings eine Reihe von Funktionen bereit. Die wichtigsten davon sollen im Folgenden besprochen werden.

### 19.3.1 strcmp()

Die Funktion `strcmp()` vergleicht zwei C-Strings miteinander. Rückgabewert ist 0, falls die Zeichenketten übereinstimmen. Andernfalls zeigt ein Rückgabewert größer als Null an, dass die erste der beiden zu übergebenden Zeichenketten lexikographisch größer ist als die zweite. Bei einem Rückgabewert kleiner als Null verhält es sich genau umgekehrt – die zweite Zeichenkette ist lexikographisch größer als die erste.

Eine Zeichenfolge ist lexikographisch größer als eine andere, wenn die Ordnungsnummer des ersten nicht übereinstimmenden Zeichens im ASCII-Code größer ist. Dabei ist zu berücksichtigen, dass den Zeichen '0' bis '9' bzw. 'A' bis 'Z' bzw. 'a' bis 'z' im ASCII-Code fortlaufende Nummern zugeordnet sind. Demnach ist z.B. das Zeichen 'B' »größer« als das Zeichen 'A' und die Zeichenkette "Wand" »größer« als "Hand". In gleicher Weise ist der String "Hanf" kleiner als "Hans", da eben das erste nicht übereinstimmende Zeichen – hier 'f' bzw. 's' – im ASCII-Code eine kleinere Ordnungsnummer besitzt.

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

int main(void)
{
    char meineZ[] = "Dagobert Duck";
    char deineZ[] = "Donald";
    int vgl = strcmp(meineZ, deineZ);
```

```

    if (vgl == 0)
        cout << "Beide Strings sind gleich";
    else if (vgl > 0)
        cout << "\"Dagobert Duck\" ist "
            << "lexikographisch gr\x94\xE1" << "er";
    else if (vgl < 0)
        cout << "\"Donald\" ist lexikographisch"
            << " gr\x94\xE1" << "er";
    return 0;
}

```

Ausgabe:

```
"Donald" ist lexikographisch größer
```

Es sei darauf hingewiesen, dass die Anzahl der Zeichen bei dieser Betrachtung nur dann ausschlaggebend ist, falls ansonsten alle Zeichen übereinstimmen. Demgemäß ist die Zeichenkette "Donald Duck" tatsächlich größer als "Donald". Ebenso wenig kommt es auf die Größe der entsprechenden Arrays an, entscheidend ist allein die in ihnen enthaltene Zeichenkette.

```

#include <iostream>
using namespace std;

int main(void)
{
    char meineZ[23] = "Daisy";
    char deineZ[6] = "Daisy";
    if (strcmp(meineZ, deineZ) == 0)
        cout << "Beide Strings sind gleich";
    else
        cout << "Die Strings sind nicht gleich";
    return 0;
}

```

Ausgabe:

```
Beide Strings sind gleich
```

### Hinweis

Im Funktionsnamen von `strcmp()` steht das »cmp« für das englische Wort »compare«, zu Deutsch »vergleichen«. Das »str« steht für »String(s)«.

### 19.3.2 strcpy()

Mit der Funktion `strcpy()` – das »cpy« steht für »copy« – können Sie einen C-String in einen anderen kopieren. Als erstes Argument beim Funktionsaufruf wird der Zielstring erwartet, als zweites der Quellstring:

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

int main(void)
{
    char erstes[] = "Asterix", zweites[] = "Idefix";
    strcpy(erstes, zweites);
    cout << erstes;
    return 0;
}
```

Ausgabe: Idefix

#### Achtung

Achten Sie darauf, dass das entsprechende Array (im Beispiel `erstes`) groß genug ist, um den zu kopierenden String aufzunehmen.

Zeichen, die nicht überschrieben werden, verbleiben im Array. Im Beispiel werden die darstellbaren Zeichen 'A', 's', 't', 'e', 'r', 'i', 'x' des Arrays `erstes` sämtlich mit den Zeichen 'I', 'd', 'e', 'f', 'i', 'x' sowie '\0' überschrieben. Ein weiteres '\0', das sich an der Position mit dem Index 7 befindet, bleibt davon unberührt. Das Array `zweites` wird im Zuge des Kopiervorgangs nicht verändert.

#### Tipp

Man kann sich die Reihenfolge der Argumente *Zielstring*, *Quellstring* übrigens leicht einprägen, da sie sich wie eine Zuweisung lesen. Bei einer solchen steht das Ziel ja auch als Erstes, und danach folgt der zuzuweisende Wert.

### 19.3.3 Die Konvertierungsfunktionen `atoi()`, `itoa()`, `atof()`

Zwar versagt bei der Konvertierung numerischer Werte in Strings bzw. von Strings in numerische Werte auch der `cast`-Operator:

```
int zahl;
zahl = static_cast<int> ("77"); // FEHLER
```

Dennoch sind solche Konvertierungen möglich. Dazu dienen die Funktionen `atoi()`, `itoa()` und `atof()`.



### Hinweis

atoi bedeutet »ascii to int«, itoa »int to ascii« und atof »ascii to float«.

Die Umwandlung von C-Strings nach Integer erfolgt mit der Funktion `atoi()`. Der zu konvertierende String ist ihr beim Aufruf zu übergeben:

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

int main(void)
{
    int zahl;
    zahl = atoi("77");
    cout << zahl + 23;
    return 0;
}
```

Ausgabe: 100

bzw.

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

int main(void)
{
    char meinStr[] = "77";
    int zahl;
    zahl = atoi(meinStr);
    cout << zahl + 23;
    return 0;
}
```

Ausgabe: 100

### Hinweis

Möglicherweise müssen Sie die Headerdatei *cstdlib* – bei älteren Compilern *stdlib.h* – einbinden, um die Funktionen `atoi()`, `itoa()` und `atof()` verwenden zu können:

```
#include <cstdlib>

bzw.

#include <stdlib.h>
```

Natürlich ist Voraussetzung, dass der umzuwandelnde String überhaupt als Zahl interpretiert werden kann. Dabei ist entscheidend, dass der String mit einer Ganzzahl beginnt. Weitere Zeichen werden gegebenenfalls ignoriert:

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

int main(void)
{
    char deinStr[] = "997abc";
    cout << atoi(deinStr);
    return 0;
}
```

Ausgabe: 997

Verwenden Sie die Funktion `atof()`, falls ein String in einen Gleitkommawert konvertiert werden soll:

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

int main(void)
{
    char ihrStr[] = "7.85";
    double z = atof(ihrStr);
    cout << z * 2;
    return 0;
}
```

Ausgabe: 15.7

Auch hier gilt wiederum, dass die ersten Zeichen als Gleitkommawert bzw. als ein Wert, der in einen solchen konvertierbar ist, angesehen werden können. Demnach verlaufen auch die Konvertierungen

```
atof("7.85 Euro") // ergibt 7.85
```

bzw.

```
atof("7ab44xy") // ergibt 7.0
```

erfolgreich.

In der umgekehrten Richtung wandelt die Funktion `itoa()` Integerwerte in Strings um:

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

```
int main(void)
{
    int num = 2077;
    char seinStr[20];
    itoa(num, seinStr, 10);
    cout << seinStr;
    return 0;
}
```

Ausgabe: 2077

Der Funktion `itoa()` sind beim Aufruf drei Parameter zu übergeben: der umzuwandelnde Integerwert, die Adresse eines `char`-Arrays, in dem der konvertierte Wert als Zeichenkette gespeichert werden soll, und die Basis für das gewünschte Zahlensystem – für das gewohnte Dezimalsystem also 10.

```
itoa(num, seinStr, 10);
```

Gibt man als dritten Parameter z.B. 16 an, dann wird die Zahl 2077 als hexadezimaler Wert interpretiert.

### Achtung

Beim Aufruf von Funktionen muss die Reihenfolge der zu übergebenden Werte stets beachtet werden. Speziell bei `itoa()` ist zu beachten, dass erst die Quelle (der Integerwert) und dann das Ziel (der String) angegeben werden muss.

## 19.4 string-Variablen

Wie an anderer Stelle schon erwähnt, sollten Sie es im Zweifel vorziehen, Zeichenketten mithilfe der `string`-Klasse zu verarbeiten, was schließlich dem C++-Stil entspricht. Die `string`-Klasse ist eine bedeutende Erweiterung gegenüber C und sie besitzt zahlreiche Vorteile, nicht nur was ihre Handhabung und ihr Zusammenspiel mit den Datenströmen `cout` und `cin` angeht.

Die `string`-Klasse baut letzten Endes ebenfalls auf Zeichenketten in Form eines `char`-Arrays mit einem abschließenden Nullbyte auf, wovon der Programmierer aber so gut wie nichts mitbekommt. Das heißt, es wird automatisch dafür gesorgt, dass ein Endezeichen (`'\0'`) angehängt wird, dass ausreichend Platz für die gerade aufzunehmende Zeichenkette reserviert wird usw. Entsprechend bequemer lässt sich in Abwandlung des obigen Beispiels zur Methode `cin.get()` die zeichenweise Eingabe mit einer `string`-Variablen umsetzen:

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

```
int main(void)
{
    string name;
    char c;
    do
    {
        c = cin.get();
        name += c;
    } while (c != '\n');
    cout << name << endl;
    return 0;
}
```

Ausgabe:

```
Hans Meier
Hans Meier
```

Beachten Sie, dass Sie sich hier weder um die Größe der Zeichenkette noch um das Stringendezeichen '\0' kümmern müssen. Beides erledigt die `string`-Klasse bzw. die Variable `name`, die eigentlich ein Objekt dieser Klasse ist.

Die `char`-Variable `c` ist hier für die Bedingungsprüfung notwendig. Ansonsten könnte man mit der Anweisung

```
name += cin.get();
```

das eingelesene Zeichen direkt dem String hinzufügen.

### Hinweis

Beachten Sie, dass mit Ausführung der zusammengesetzten Zuweisung

```
name += c;
```

dem in `name` enthaltenen String ein Zeichen an der vorletzten Position – zwischen Nullbyte und dem zuletzt gespeicherten Zeichen – hinzugefügt wird.

Im Übrigen ist es auch möglich, mittels Indexangabe auf einzelne Zeichen einer `string`-Variablen zuzugreifen:

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

int main(void)
{
    string name = "Hans Meier";
```

```
cout << name[0] << ". "  
      << name[5] << " '.";  
return 0;  
}
```

Ausgabe: H. M.

Berücksichtigen Sie aber, dass die Werte von `string`-Variablen dynamisch verwaltet werden. Das bedeutet, dass nach einer Neuweisung für die Variable bzw. für deren Zeichenkette unter Umständen weniger Platz im Speicher zur Verfügung steht als zuvor. So wird im Beispiel für die `string`-Variable `name` nach der Zuweisung der Zeichenkette "Hans Meier" ein `char`-Array zur Aufnahme von elf Zeichen angelegt. Dies geschieht, wie gesagt, automatisch, sozusagen vom Programmierer verborgen. Die Ursache liegt in der Funktionalität der `string`-Klasse begründet. Weist man nun der Variablen die Zeichenkette "Udo" zu, so reduziert sich der reservierte Speicherbereich entsprechend. Das heißt, der alte Speicherbereich wird freigegeben und ein neuer zur Aufnahme von nunmehr vier Zeichen angelegt:

```
#include <iostream>  
#include <string>  
using namespace std;  
  
int main(void)  
{  
    string name = "Hans Meier";  
    name[7] = 'y'; // OK  
    name = "Udo";  
    name[6] = 'a'; // FEHLER  
    return 0;  
}
```

Demzufolge ist der letzte Zugriff auf ein siebtes Element (`name[6]`) nicht mehr korrekt, da zum Zeitpunkt für den Wert der Variablen `name` nur vier Elemente – für vier Zeichen – zur Verfügung stehen (Laufzeitfehler).

### Achtung

Ein Überschreiben des vierten Elements, in dem sich ja nach Zuweisung von "Udo" das Nullbyte befindet, wäre jedoch ebenfalls fatal.

## 19.4.1 Die Methode `c_str()`

Grundsätzlich sind die in Abschnitt 19.3 »Funktionen zur Stringverarbeitung« genannten Funktionen nur auf C-Strings, nicht auf `string`-Variablen anwendbar. Dies ist zwar nicht problematisch, da die `string`-Klasse ihre eigene Funktionalität besitzt.

In verschiedenen Situationen kann es aber dennoch erforderlich bzw. vorteilhaft sein, auf C-String-Funktionen zurückzugreifen. Was die Funktionen `atoi()`, `atof()` und `itoa()` angeht, bleibt dem C++-Programmierer beispielsweise gar nichts anderes übrig, da es in der `string`-Klasse keine Pendant zu diesen gibt.

Um nun die genannten C-String-Funktionen bei Bedarf dennoch nutzen zu können, bietet die Klasse `string` die Methode `c_str()` an, mit der `string`-Variablen bzw. deren Werte in C-Strings konvertiert werden können:

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

int main(void)
{
    string preisangabe = "55.49 Euro";
    double preis = atof(preisangabe.c_str());
    cout << preis;
    return 0;
}
```

Ausgabe: 55.49

Im Ausdruck

```
atof(preisangabe.c_str())
```

wird zunächst der Wert der `string`-Variablen `preisangabe` in eine Zeichenkette umgewandelt, was die Methode `c_str()` bewirkt. Der C-String wird dann von der Funktion `atof()` nach `double` konvertiert.

### Hinweis

Im Gegensatz zu Funktionen, die gewissermaßen unabhängig sind, bezieht sich ein Methodenaufruf gewöhnlich auf ein bestimmtes Objekt einer Klasse. Eine Methode wird daher unter Angabe des Objektnamens in der Form

*Objekt.Methode()*

aufgerufen. Daher erfolgt der Aufruf der Methode `c_str()` für das Objekt `preisangabe` – jede `string`-Variable ist, wie gesagt, ein Objekt der Klasse `string` – in der entsprechenden Weise:

```
preisangabe.c_str()
```

Die Konvertierung in der entgegengesetzten Richtung verläuft automatisch, wofür wiederum die string-Klasse verantwortlich ist:

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

int main(void)
{
    char uderzo[] = "Asterix und Obelix";
    string comic = uderzo;
    cout << comic;
    return 0;
}
```

Ausgabe: Asterix und Obelix

Wie am Beispiel zu sehen, kann man einer string-Variablen einfach die Adresse eines C-Strings zuweisen.

### Hinweis

Dazu muss gesagt werden, dass ein Zeichenkettenliteral ebenfalls für die Adresse eines – namenlosen – Arrays steht. Wie bereits erwähnt, wird z.B. mit Ausführung der Anweisung

```
cout << "C++";
```

ein char-Array angelegt und cout dessen Adresse mitgeteilt. Der Befehl an cout lautet also nicht etwa

»gib die Zeichen 'C', '+', '+' aus«,

sondern

»gehe an die Speicherzelle *xy* und gib ab dieser Stelle alle Zeichen bis zum '\0' nacheinander aus«.





# 20 Funktionen

Jeder ausführbare Code in einem C++-Programm befindet sich innerhalb von Funktionen und Methoden. Letztere unterscheiden sich von Funktionen dadurch, dass sie zu einer Klasse gehören (dazu mehr in Kapitel 24 »Klassen und Objekte«). Die Aufteilung des Codes in Funktionen erlaubt es, bestimmte Teilaufgaben in separate Einheiten auszulagern. Dadurch wird der Code übersichtlicher, wartungsfreundlicher und weniger fehleranfällig.

Nach ihrer Verfügbarkeit lassen sich zwei Gruppen von Funktionen unterscheiden: vordefinierte und benutzerdefinierte. Vordefinierte Funktionen werden von Ihrem Compilersystem zur Verfügung gestellt, Sie können diese bei Bedarf einfach verwenden, wie wir das in unseren Beispielprogrammen bereits verschiedentlich getan haben.

## Hinweis

Vordefinierte Funktionen sind z.B. `itoa()`, `atoi()`, `atof()`, `getchar()`, um einige zu nennen. Im Unterschied dazu handelt es sich z.B. bei `cin.get()` und `cin.put()` um Methoden, wobei die eigentlichen Methoden `get()` bzw. `put()` heißen – `cin` ist das Objekt, auf das sie sich beziehen.

Benutzerdefinierte Funktionen werden dagegen vom Programmierer nach eigenem Gutdünken selbst definiert.

## Hinweis

Mit »Benutzer« ist hier also nicht etwa der zukünftige Benutzer eines Programms, sondern der Programmierer gemeint.

## 20.1 Funktionen definieren und aufrufen

Wie man Funktionen definiert, lässt sich am Beispiel von `main()` ansehen. Wie diese gliedert sich jede Funktion in einen Funktionskopf und einen Funktionsrumpf, womit der Funktionsblock, in dem die Anweisungen stehen, gemeint ist. Um eine erste Funktion – mit Blick auf `main()` sollten wir eigentlich sagen, eine weitere Funktion – selbst zu definieren, können wir uns, was den Funktionskopf angeht, fürs Erste ebenfalls an `main()` halten. Allerdings müssen wir der neuen Funktion aus bekannten Gründen einen anderen Namen geben; nennen wir sie einfach `fun()`. Außerdem definieren wir die Funktion ohne Rückgabewert. Deshalb verwenden wir das Schlüsselwort `void` vor dem Funktionsnamen und lassen im Funktionsrumpf die `return`-Anweisung weg (zu Rückgabe-

werten von Funktionen gleich mehr weiter unten in Abschnitt 20.4 »Rückgabewerte von Funktionen«). Dann sieht das Funktionsgerüst von `fun()` wie folgt aus:

```
void fun(void)
{
}
```

### Hinweis

Denken Sie daran, dass der ANSI/ISO-C++-Standard allein für die Funktion `main()` einen Rückgabewert vorschreibt. Jede andere Funktion dürfen Sie auch ohne Rückgabewert definieren.

Damit auch etwas geschieht, wenn die Funktion verwendet wird, muss der Funktionsblock natürlich Anweisungen enthalten. Zur Demonstration soll es ausreichen, wenn sich die Funktion nur in irgendeiner Weise nach außen hin bemerkbar macht. Daher begnügen wir uns zunächst mit einer simplen Ausgabeanweisung:

```
void fun(void)
{
    cout << "Hallo von fun()" << endl;
}
```

Somit kann die Definition der Funktion `fun()` als abgeschlossen betrachtet werden. Natürlich muss `fun()` bzw. deren Definition irgendwie in den Quellcode integriert werden. Dies darf auf keinen Fall innerhalb der Funktion `main()` geschehen, da es in C++ nicht erlaubt ist, Funktionen verschachtelt zu definieren:

```
...
int main(void)
{
    void fun(void) // FEHLER
    {
        cout << "Hallo von fun()" << endl;
    }
    ...
    return 0;
}
```

## Hinweis

Der Visual-Studio-Compiler erweist sich hier als sehr präzise. Die Fehlermeldung lautet:

»'fun': Lokale Funktionsdefinitionen sind unzulässig«

Borland begnügt sich mit dem Hinweis auf einen »Declaration syntax error«.

Platzieren wir die Definition von `fun()` einfach oberhalb von `main()`:

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

void fun(void)
{
    cout << "Hallo von fun()" << endl;
}

int main(void)
{
    cout << "In main()" << endl;
    return 0;
}
```

Ausgabe: In main()

Beim Ausführen des obigen Programms macht sich `fun()` in keiner Weise bemerkbar. Wie an der Ausgabe erkennbar, wird nur die Anweisung von `main()` abgearbeitet, nicht die von `fun()`.

Dies sollte Sie auch nicht zu sehr verwundern, da `fun()` in `main()` ja nirgendwo erwähnt ist. Es sei daran erinnert, dass die Funktion `main()` den Einstiegspunkt der Programmausführung bildet. Von daher steht eindeutig fest, dass die erste Anweisung im Rumpf von `main()` beim Programmlauf zuerst ausgeführt wird, dann die nächste usw., bis das Ende des Funktionsblocks erreicht ist. Die Reihenfolge des Programmablaufs ist nach den Ihnen bekannten Regeln vorgegeben. Dabei spielt es zunächst einmal keine Rolle, wie viele Funktionsdefinitionen sich im Code befinden. Solange die Programmausführung auf ihrem Weg nichts von einer anderen Funktion erfährt, bleibt diese unberücksichtigt.

## Hinweis

Ebenso wenig hat eine Funktionsdefinition Einfluss auf die Größe der ausführbaren Datei, solange die Funktion im Code nicht verwendet wird.

Wenn Sie also wollen, dass eine bestimmte Funktion zur Ausführung gelangt, dann müssen Sie diese an einer Stelle im Code verwenden – aufrufen –, die vom Programmablauf erreicht wird.

Mit anderen Worten: Die Schnittstelle zwischen einzelnen Funktionen bilden die Funktionsaufrufe. Sobald der Compiler auf einen Funktionsaufruf trifft, springt er zu der aufgerufenen Funktion und arbeitet die Anweisungen dieses Funktionsblocks ab. Im Anschluss daran erfolgt der Rücksprung in die aufrufende Funktion.

### Hinweis

Oft wird eine Funktion, aus der heraus eine andere aufgerufen wird, schlicht als »Aufrufer« bezeichnet. Dementsprechend sagt man von dieser Funktion (dem Aufrufer), sie ruft die andere Funktion auf. Mitunter spricht man auch von der aufrufenden Umgebung.

Was die Verwendung einer Funktion angeht, macht es keinen Unterschied, ob es sich um eine vordefinierte oder um eine benutzerdefinierte Funktion handelt. Voraussetzung ist allein, dass eine aufgerufene Funktion existent, das heißt definiert ist. Selbst definierte Funktionen werden also in der gleichen Weise wie vordefinierte aufgerufen – das heißt unter Angabe des Funktionsnamens gefolgt von runden Klammern. Wobei die Klammern bei unseren ersten selbst definierten Funktionen zunächst einmal leer bleiben, wie das z.B. beim Aufruf der vordefinierten Funktionen `getchar()` und `itoa()` ebenso der Fall ist. Somit lautet ein Aufruf der Funktion `fun()` einfach

```
fun()
```

Dieser Funktionsaufruf darf selbstverständlich nicht in einem komplexen Ausdruck stehen, da die Funktion `fun()` ja keinen Wert zurückliefert (zu Rückgabewerten von Funktionen siehe Abschnitt 20.4). Somit muss der Funktionsaufruf von `fun()` in einer eigenen Anweisung erfolgen:

```
fun();
```

Schließlich wollen wir ja auch nicht mehr, als dass die im Funktionsblock von `fun()` stehende Anweisung beim Programmablauf zur Ausführung gelangt:

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

void fun(void)
{
    cout << "Hallo von fun()" << endl;
}
```

```
int main(void)
{
    cout << "In main()" << endl;
    fun();
    cout << "Wieder in main()";
    return 0;
}
```

Ausgabe:

```
In main()
Hallo von fun()
Wieder in main()
```

Die Anweisung

```
cout << "Wieder in main()";
```

haben wir hinzugefügt, um den Rücksprung nach `main()` deutlich zu machen.

Grundsätzlich gilt, dass jede Funktion jede andere aufrufen kann.

### **Hinweis**

Falls eine Funktion sich selbst aufruft, spricht man von einer Rekursion (siehe dazu unten Abschnitt 20.6 »Rekursionen«).

Zur Demonstration definieren wir eine weitere Funktion `mehr_fun()` und rufen diese aus `fun()` heraus auf.

### **Hinweis**

Die Namen von Funktionen unterliegen wie die von Variablen den Regeln der Bezeichnerwahl (siehe dazu Kapitel 10, Abschnitt 10.2 »Regeln zur Bezeichnerwahl«).

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

void mehr_fun(void)
{
    cout << "Hallo von mehr_fun()" << endl;
}

void fun(void)
{
    cout << "Hallo von fun()" << endl;
```

```

    mehr_fun();
    cout << "Wieder in fun()" << endl;
}

int main(void)
{
    cout << "In main()" << endl;
    fun();
    cout << "Wieder in main()";
    return 0;
}

```

Ausgabe:

```

In main()
Hallo von fun()
Hallo von mehr_fun()
Wieder in fun()
Wieder in main()

```

Hier eine Beschreibung des Ablaufs. Die Programmausführung beginnt mit der ersten Anweisung in `main()`:

```
cout << "In main()" << endl;
```

Mit dem Funktionsaufruf

```
fun();
```

wird in die bezeichnete Funktion verzweigt und dort zunächst die Anweisung

```
cout << "Hallo von fun()" << endl;
```

ausgeführt. Die Anweisung

```
mehr_fun();
```

bewirkt den Sprung in diese Funktion. Nach Ausführung der einzigen Anweisung im Rumpf von `mehr_fun()`

```
cout << "Hallo von mehr_fun()" << endl;
```

erfolgt der Rücksprung in die aufrufende Umgebung, also zur Funktion `fun()`, und zwar an die Stelle, die unmittelbar dem Funktionsaufruf folgt, mit dem die Funktion verlassen wurde. Daher wird als Nächstes die Anweisung

```
cout << "Wieder in fun()";
```

ausgeführt. Damit ist der Funktionsblock von `fun()` abgearbeitet und die Programmausführung landet wieder in `main()`, dem Aufrufer von `fun()`. Hier ist noch die Anweisung

```
cout << "Wieder in main()";
```

abzuarbeiten. Mit der `return`-Anweisung endet die Programmausführung.

### Achtung

Beachten Sie, dass Bezeichner auch zwischen Funktionen und Variablen eindeutig bleiben müssen. Das heißt, eine lokale Variable einer Funktion darf nicht den Namen einer im Code dieser Funktion aufgerufenen anderen Funktion tragen.

```
...
void fun(void)
{
    ...
}

int main(void)
{
    int fun;
    fun(); // FEHLER
    ...
    return 0;
}
```

## 20.2 Funktionsprototypen

Bevor eine Funktion verwendet werden kann, muss sie definiert sein. Falls etwa die Definition von `mehr_fun()` im obigen Beispiel unterhalb von `fun()` stünde, ließe sich die Funktion `mehr_fun()` in `fun()` nicht aufrufen – wohl aber in `main()`:

```
...
void fun(void)
{
    cout << "Hallo von fun()" << endl;
    mehr_fun(); // FEHLER
    cout << "Wieder in fun()" << endl;
}

void mehr_fun(void)
{
    cout << "Hallo von mehr_fun()" << endl;
}
```

```
int main(void)
{
    ...
    return 0;
}
```

Das Gleiche gilt natürlich für die Funktion `fun()` in Bezug auf `main()`, falls `fun()` im Code unterhalb von `main()` definiert ist:

```
...
int main(void)
{
    cout << "In main()" << endl;
    fun(); // FEHLER
    cout << "Wieder in main()";
    return 0;
}

void fun(void)
{
    ...
}
```

Dies würde natürlich erhebliche Probleme mit sich bringen, falls im Code eine Vielzahl von Funktionsdefinitionen vorhanden ist, zumal wenn sich die Funktionen gegenseitig aufrufen.

Eine Lösung besteht darin, die betreffenden Funktionen am Anfang der Quelldatei mittels so genannter Prototypen zu deklarieren. Ein Funktionsprototyp besteht ganz einfach aus dem Funktionskopf plus abschließendes Semikolon:

```
void fun(void);
void mehr_fun(void);
```

### Hinweis

Wie an anderer Stelle schon erwähnt, dürfen Sie das `void` in den Klammern auch weglassen, sowohl bei den Prototypen als auch im Kopf der Funktionsdefinitionen.

Die Reihenfolge der Funktionsdeklarationen spielt keine Rolle. Stellen Sie sich einfach vor, dass Ihr Compiler sich die Prototypen ein Mal anschaut und dann über die deklarierten Funktionen ausreichend Bescheid weiß. Das heißt, sobald Ihr Compiler einen Prototyp antrifft, ist es egal, an welcher Stelle im Code sich die entsprechende Funktionsdefinition befindet.



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

void fun(void);
void mehr_fun(void);

int main(void)
{
    cout << "In main()" << endl;
    fun();
    cout << "Wieder in main()";
    return 0;
}

void fun(void)
{
    cout << "Hallo von fun()" << endl;
    mehr_fun();
    cout << "Wieder in fun()" << endl;
}

void mehr_fun(void)
{
    cout << "Hallo von mehr_fun()" << endl;
}
```

Ausgabe:

```
In main()
Hallo von fun()
Hallo von mehr_fun()
Wieder in fun()
Wieder in main()
```

### Warum sich Ihr Compiler mit Prototypen zufrieden gibt

Wie wir im Weiteren noch sehen werden, kann man am Kopf einer Funktion ablesen, ob eine Funktion einen Wert zurückgibt bzw. welchen Datentyp der Rückgabewert besitzt. Das Gleiche gilt für Werte, die an eine Funktion beim Aufruf zu übergeben sind.

Nun ist es so, dass der eigentliche Compiler nicht die *.exe*-Datei erzeugt, das Endprodukt des gesamten Übersetzungsvorgangs, sondern eine oder mehrere *.obj*-Dateien – so genannte Objektdateien – und zwar für jede Quellcodedatei eine *.obj*-Datei. Mehrere *.obj*-Dateien entstehen also für den Fall, dass der Quellcode auf mehrere Dateien verteilt ist. Bei den Objektdateien handelt es sich bereits um Maschinencode.

### Warum sich Ihr Compiler mit Prototypen zufrieden gibt (Forts.)

Funktionsaufrufe im Code werden vom eigentlichen Compiler jedoch nicht aufgelöst, dies besorgt anschließend der Linker. Was der Compiler jedoch durchführt, ist eine Typüberprüfung. Das heißt, er kontrolliert, ob die Werte, die einer Funktion beim Aufruf übergeben werden, in Anzahl und Datentyp mit den Angaben in der Funktionsdefinition übereinstimmen. Das Gleiche geschieht in Bezug auf einen eventuellen Funktionsrückgabewert. Das heißt, der Compiler überprüft, ob an der Stelle des Funktionsaufrufs ein Wert dieses Datentyps stehen darf.

```
cout << fun(); // FEHLER
```

In der obigen Anweisung kann `cout` mit dem Ausdruck `fun()` (der ja eigentlich aus folgend genanntem Grunde keiner ist) nichts anfangen, da die Funktion `fun()` keinen Wert zurückgibt. Als Folge davon wird der Compiler mit einer Fehlermeldung reagieren. Der Compiler stellt dabei über den Funktionskopf fest, dass `fun()` keinen Rückgabewert aufweist. Dort legt das erste `void` fest, dass die Funktion keinen Wert zurückgibt. Alle Informationen, die der Compiler für seine Arbeit benötigt, kann er also ebenso gut über einen Funktionsprototyp erhalten.

## 20.2.1 Den Quellcode auf mehrere Dateien verteilen

Entscheidende Bedeutung kommt den Funktionsprototypen bei der Modularisierung von Programmen zu, das heißt, wenn es darum geht, den Quellcode auf mehrere Dateien zu verteilen. Dazu geht man gewöhnlich wie folgt vor:

- Man schreibt die Funktionsdefinitionen in eine separate Quelldatei.
- Die Prototypen dieser Funktionen schreibt man in eine so genannte Headerdatei mit der Endung `.h`. Der Name »Header« bezieht sich auf die Funktionsköpfe (Englisch »head«, Deutsch »Kopf«). Schließlich besteht ein Funktionsprototyp ja aus dem Kopf einer Funktion plus Semikolon. Die Headerdatei sollte möglichst den Namen der Funktionsdatei tragen, für eine Quelldatei `function.cpp` also `function.h`.
- Diese Headerdatei bindet man in den Quelldateien des Projekts, die im Code eine oder mehrere der in der Headerdatei deklarierten Funktionen verwenden, mit der `#include`-Direktive ein.

Dabei ist zu beachten, dass vom Programmierer bearbeitete Headerdateien nicht mit spitzen Klammern (`<>`), sondern mit doppelten Anführungszeichen angegeben werden:

```
#include "function.h"
```

Der Unterschied liegt im Suchpfad Ihres Compilers. Falls Sie eine Headerdatei mit `<>` einbinden, erwartet er, dass sich diese auf der Festplatte im Include-Verzeichnis der Compilerinstallation befindet. Setzen Sie dagegen die Headerdatei innerhalb der `#include`-Direktive in Anführungszeichen ("`...`"), so sucht Ihr Compiler in dem Verzeichnis, in dem sich die Quelldateien des Projekts befinden.

Wie gesagt, ist eine Headerdatei in jeder Quelldatei einzubinden, in der eine oder mehrere der in ihr deklarierten Funktionen verwendet werden. Dies sollte auch in der Quelldatei geschehen, in der sich die zugehörigen Funktionsdefinitionen befinden, für den Fall, dass sich die Funktionen gegenseitig aufrufen.

Denken Sie daran, dass jede Quellcodedatei zunächst separat in Objektcode übersetzt wird und alle Funktionen, die im Code einer Quelldatei verwendet werden, Ihrem Compiler aus den weiter oben genannten Gründen bekannt sein müssen. Und falls die Funktionsdefinitionen in einer eigenen Quelldatei stehen, ist es der einzige Weg, dem Compiler die notwendigen Informationen über besagte Funktionen mitzuteilen, diese vor der ersten Verwendung zu deklarieren – was idealerweise am Anfang der Quelldatei geschehen sollte. Wenn Sie nun die entsprechende Headerdatei einbinden, dann ersetzt der Präprozessor die `#include`-Direktive mit dem Inhalt der angegebenen Headerdatei. Im Ergebnis stehen dann die erforderlichen Funktionsdeklarationen im Code, wenn der eigentliche Compiler diesen zur Übersetzung erhält.

### Hinweis

Aus dem gleichen Grund ist es notwendig, die Headerdatei *iostream* in allen Quelldateien einzubinden, in deren Code z.B. `cout` bzw. `cin` verwendet wird.

Wenn wir nun auf unser obiges Beispiel bezogen den Code in der eben beschriebenen Weise aufteilen, dann ergeben sich drei Dateien folgenden Inhalts:

*function.h*:

```
void fun(void);
void mehr_fun(void);
```

*function.cpp*:

```
#include <iostream>
#include "function.h"
using namespace std;

void fun(void)
{
    cout << "Hallo von fun()" << endl;
    mehr_fun();
    cout << "Wieder in fun()" << endl;
}

void mehr_fun(void)
{
    cout << "Hallo von mehr_fun()" << endl;
}
```

*haupt.cpp*:

```
#include <iostream>
#include "function.h"
using namespace std;

int main(void)
{
    cout << "In main()" << endl;
    fun();
    cout << "Wieder in main()";
    return 0;
}
```

### Hinweis

Die Namen der Dateien – hier *function* und *haupt* – sind natürlich beliebig wählbar. In der Regel fasst man thematisch verwandte Funktionen in einer Datei zusammen und wählt den Namen der Quell- bzw. Headerdatei entsprechend. Als Beispiel sei hier die in Ihrem Compilersystem enthaltene Headerdatei *math.h* genannt, in der eine Reihe von mathematischen Funktionen deklariert ist.

### Hinweis

Die Dateien des Beispiels – *function.h*, *function.cpp* und *haupt.cpp* – finden Sie gepackt als *k20e.zip* im Ordner *Beispiele/K20* auf der Buch-CD. Wenn Sie den Borland C++-Compiler wie in Kapitel 2, Abschnitt 2.3 »Installation des Borland-Compilers«, beschrieben eingerichtet haben, dann brauchen Sie *bcc32.exe* einfach nur, durch Leerzeichen getrennt, nacheinander die Namen der zu kompilierenden Quelldateien anzugeben – Headerdateien bleiben außen vor.

```
bcc32 haupt.cpp function.cpp
```

Sie sollten aber alle Projektdateien – hier *haupt.cpp*, *function.cpp* und natürlich auch *function.h* – in ein und dasselbe Verzeichnis legen. Die entstehende *.exe*-Datei trägt stets den Namen der Quelldatei, die Sie zuerst angeben – hier heißt sie also *haupt.exe*.

## 20.2.2 Prototypen von vordefinierten Funktionen

Wie Sie ja bereits wissen, stellt jedes Compilersystem eine Vielzahl von vordefinierten Funktionen und anderen Elementen zur Verfügung. Diese sind in so genannten Bibliotheken nach ihrem Verwendungszweck aufgeteilt, wobei man sich eine Bibliothek als eine Zusammenfassung von mehreren Dateien vorzustellen hat. So stellen Visual C++-Compiler neben der so genannten Standardbibliothek gewöhnlich Bibliotheken mit vordefinierten Funktionen, Klassen etc. zur Programmierung von grafischen Benutzeroberflächen zur Verfügung – auf die wir aus bereits genannten Gründen bis auf Weiteres

nicht eingehen. Darüber hinaus ist es auch möglich, nach dem im letzten Abschnitt beschriebenen Verfahren eigene Bibliotheken zu erstellen, indem man z.B. mehrere Funktionen in einer oder auch mehreren Quelldateien nach einem Aufgabengebiet zusammenfasst.

### Hinweis

Von einer Bibliothek wird man womöglich erst dann sprechen, wenn es sich tatsächlich um eine Vielzahl von Funktionsdefinitionen handelt, die dann auf mehrere Quelldateien verteilt sind.

Der Code von vordefinierten Funktionen bleibt jedoch im Verborgenen, Ihr Compilersystem schleppt ihn sozusagen ständig mit sich herum. Dies ist ja auch nicht problematisch, sofern Sie nur wissen, wie eine bestimmte Funktion zu verwenden ist. Was allerdings greifbar ist, sind die zugehörigen Headerdateien. Diese befinden sich im Verzeichnis *Include* Ihrer Compilerinstallation. Falls Sie im Code ein bestimmtes vordefiniertes Element verwenden, müssen Sie die entsprechende Headerdatei nach dem bekannten Muster einbinden.

### Hinweis

Neben Funktionsdeklarationen finden sich in Headerdateien in der Regel weitere Präprozessor-Direktiven, mitunter Klassendefinitionen und auch Vereinbarungen von Variablen bzw. Objekten.

Die vom Compilersystem mitgelieferten Headerdateien lassen sich, an der Endung erkennbar, in drei bzw. vier Kategorien einteilen:

- Aus der Programmiersprache C übernommene Headerdateien tragen die Endung *.h* (z.B. *stdio.h*, *math.h*, *conio.h* usw.).
- Dies gilt auch für C++-Headerdateien im alten Stil, wie z.B. *iostream.h*.
- Headerdateien ohne Erweiterung, die mit einem »c« beginnen, sind neuere Entsprechungen alter C-Headerdateien; so ist etwa *cmath* das Gegenstück der älteren, aus C stammenden Headerdatei *math.h*. Das Gleiche gilt für *ctime* und das Gegenstück *time.h* sowie etliche weitere Headerdateien.
- C++-Headerdateien im neuen Stil haben keine Erweiterung, wie z.B. *iostream*, *iomanip* etc.

Die letzten beiden Kategorien beschreiben in der Regel die Standardbibliothek von C++. Weiter oben ist das Wort »Standardbibliothek« ja bereits gefallen. So wird der Teil vordefinierter bzw. »vorzudefinierender« Elemente bezeichnet, der vom C++-Standard vorgegeben ist. Das heißt, ein Compilerhersteller, der sich am ANSI/ISO-C++-Standard orientiert, wird diejenigen Funktionen, Klassen etc. zur Verfügung stellen, die eben dieser Standard vorschreibt. Was darüber hinausgeht, kann sozusagen als Zugabe angesehen werden. Somit kann jeder Programmierer, der bei der Kodierung ausschließlich diejenigen Elemente verwendet, die in der Standardbibliothek definiert sind, davon ausgehen,

dass sein Programm auch in Verbindung mit anderen Compilern übersetzt werden kann – unter der Voraussetzung natürlich, dass diese sich an den C++-Standard halten.

Elemente der C++-Standardbibliothek sind im Namensbereich `std` definiert. Das bedeutet, bei jeder Verwendung eines Elements aus der Standardbibliothek müssen Sie dem Compiler mitteilen, dass Sie dieses bestimmte Element aus dem Namensbereich `std` meinen. Dies können Sie tun, indem Sie die Bezeichnung des Namensraums dem Element voranstellen, verknüpft durch den Bereichsauflösungsoperator `::`.

```
#include <iostream>

int main(void)
{
    double z;
    std::cin >> z;
    std::cout << z;
    return 0;
}
```

### Hinweis

Für den `»::«`-Operator existieren mehrere Bezeichnungen. Er ist als »Bereichszugriffsoperator«, »Bereichsauflösungsoperator«, »scope-Operator« und »global resolution-Operator« bekannt.

Die wohl bequemere Variante ist es, dem Compiler gleich zu Beginn mitzuteilen, dass mit allen verwendeten Elementen grundsätzlich solche des Namensbereichs `std` gemeint sind.

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

int main(void)
{
    double z;
    cin >> z;
    cout << z;
    return 0;
}
```

## 20.3 Funktionsparameter

Ein wesentlicher Vorteil des Auslagerns von Teilaufgaben in Funktionen besteht darin, dass Funktionen wiederholt verwendet werden können. Das heißt, die zur Erfüllung einer bestimmten Routine erforderlichen Anweisungen müssen nur ein Mal kodiert werden und können danach – mittels Funktionsaufruf – nach Bedarf immer wieder – gegebenenfalls auch in weiteren Programmen – verwendet werden.

Eine gegenüber den weiter oben zur reinen Demonstration erstellten Funktionen `fun()` und `mehr_fun()` vorerst wohl nur geringfügig nützlichere Funktion wäre z.B. eine Funktion `sternchen()`, die – sagen wir drei – Sternchen ausgibt:

```
void sternchen(void)
{
    cout << "***";
}
```

Immerhin ließe sich mit dieser Funktion auf etwas bequemere Weise die Ausgabe

```
***
C++
***
```

erzeugen.

```
...
int main(void)
{
    sternchen();
    cout << endl << "C++" << endl;
    sternchen();
    return 0;
}
```

Natürlich wird man sich des Verfahrens, Code in Funktionen auszulagern, nur unter der Voraussetzung bedienen, dass es sich entweder um eine sehr komplexe Teilaufgabe handelt oder unter dem Aspekt der Wiederverwendbarkeit. Aus letzterem Gesichtspunkt wäre es in Bezug auf die Funktion `sternchen()` wünschenswert, diese in ihrer Verwendbarkeit flexibler zu gestalten. Hätte man eine Funktion zur Verfügung, der sich beim Aufruf die Anzahl der auszugebenden Sternchen mitteilen lässt, dann könnte sich diese Funktion in manchen Situationen durchaus als nützlich erweisen. Noch schöner wäre es, wenn die Funktion hinsichtlich des auszugebenden Zeichens ebenfalls flexibel wäre. Das heißt, es muss ein Datenaustausch zwischen Funktion und Aufrufer erfolgen.

## Hinweis

Beachten Sie, dass die Funktion `sternchen()` in der obigen Fassung keiner Information von außen – das heißt von der aufrufenden Funktion – bedarf. Sie weiß auch so, was sie zu tun hat, nämlich genau drei Sternchen auszugeben.

Sie haben bereits eine Vielzahl von Funktionen verwendet – genauer aufgerufen. Daher haben Sie vermutlich eine Vorstellung davon, wie dies in Bezug auf eine dementsprechend eingerichtete Funktion `sternchen()` auszusehen hat. Angenommen, der Funktion ist bei Verwendung als erster Wert ein Zeichen und als zweiter ein Integerwert für die Anzahl zu übergeben, dann könnte ein Aufruf von `sternchen()` wie folgt aussehen:

```
sternchen('*', 45);
```

Mit obiger Anweisung würde man dann die Ausgabe

```
*****
```

erzielen – vorausgesetzt, die Funktion würde bereits in der gewünschten Fassung existieren, das heißt, eine entsprechende Definition der Funktion `sternchen()` wäre bereits vorhanden. Da das nicht der Fall ist, werden wir dies nun nachholen.

Wie Sie wissen, benötigt man zum Speichern von Daten Variablen. Die Funktion `sternchen()` erfordert also eine Variable zur Aufnahme eines Zeichens und eine weitere, um die Information festzuhalten, wie oft dieses Zeichen ausgegeben werden soll. Die passenden Datentypen sind demnach `char` für die erste Variable und `int` für die zweite. Nennen wir sie `zeichen` und `anzahl`.

Wenn man die oben genannten Variablen im Rumpf der Funktion deklariert, erreicht man jedoch keinesfalls das gewünschte Resultat:

```
void sternchen(void)
{
    char zeichen;
    int anzahl;
    for (int i = 0; i < anzahl; i++)
        cout << zeichen;
}
```

Eigentlich ist man dem Ziel aber schon ziemlich nahe gekommen. Man muss nur wissen, dass Variablen, die mit den Übergabewerten eines Funktionsaufrufs initialisiert werden sollen, im Kopf der aufgerufenen Funktion deklariert werden müssen, und zwar zwischen den runden Klammern. Falls es sich, wie vorliegend der Fall, um mehrere handelt, werden diese durch Kommata voneinander getrennt:



```
void sternchen(char zeichen, int anzahl)
{
    for (int i = 0; i < anzahl; i++)
        cout << zeichen;
}
```

Dabei fällt das Schlüsselwort `void` in den Klammern im Unterschied zur ersten Definition von `sternchen()` weg. Das Schlüsselwort `void` in der Klammer hinter dem Funktionsnamen – im ursprünglichen Beispiel `void sternchen(void)` – gibt an, dass eine Funktion bei ihrem Aufruf keine Werte erwartet – und man ihr natürlich auch keine mitgeben darf. Eine so definierte Funktion besitzt ja keine Variablen, die in der Lage sind, Übergabewerte aufzunehmen. Statt `(void)` hinter dem Funktionsnamen kann man übrigens auch eine Leerklammer schreiben (im Beispiel `void sternchen()`).

## Referenz

Die Bedeutung von `void` am Anfang des Funktionskopfes erfahren Sie im nächsten Abschnitt.

Die Definition der Funktion `sternchen()` ist nun abgeschlossen und wir können sie verwenden. Beachten Sie, dass auch hier die Variablen `zeichen` und `anzahl` lokal zur Funktion `sternchen()` sind, das heißt, sie können im Block der Funktion angesprochen werden. Von daher besteht kein Unterschied zu einer Variablen, die im Block der Funktion deklariert ist, wie hier z.B. die Schleifenvariable `i`.

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

void sternchen(char zeichen, int anzahl)
{
    for (int i = 0; i < anzahl; i++)
        cout << zeichen;
}

int main(void)
{
    sternchen('Y', 8);
    sternchen('\n', 2);
    sternchen('$', 9);
    return 0;
}
```

Ausgabe:

```
YYYYYYYYY
$$$$$$$$$
```

Hier einige Begriffserklärungen:

- Die im Funktionskopf deklarierten Variablen einer Funktion bezeichnet man als »formale Parameter« – oder einfach nur Parameter – dieser Funktion.
- Die im Funktionsaufruf angegebenen Werte – hier 'Y' und 8 bzw. '\n' und 2 bzw. '\$' und 9 – nennt man ebenfalls Parameter oder – um den Unterschied zu den formalen Parametern deutlich zu machen – Argumente bzw. aktuelle Parameter.
- Die Gesamtheit der in Klammern angegebenen Parameter heißt Parameterliste. Diese Bezeichnung wird nicht nur für die formalen Parameter der Funktionsdefinition gebraucht, sondern auch bezüglich der aktuellen Parameter eines Funktionsaufrufs.

Beachten Sie:

In der Parameterliste der Funktionsdefinition ist für jeden formalen Parameter ein Typbezeichner anzugeben. Das gilt nicht nur für den Fall, dass die Parameter wie im obigen Beispiel unterschiedlichen Datentyps sind, sondern auch für Parameter gleichen Datentyps.

```
void function(int a, int b)
{
    cout << a * b;
}
```

Fehlerhaft wäre es demnach, den Kopf der Funktion wie folgt zu schreiben:

```
void function(int a, b) // FEHLER
```

Beim Aufruf einer Funktion ist zu beachten, dass die Argumente in der Reihenfolge anzugeben sind, die der formalen Parameterliste in der Definition der aufgerufenen Funktion entspricht. Das heißt, der erste Formalparameter wird mit dem Wert des ersten Arguments initialisiert, der zweite mit dem Wert des zweiten Arguments usw. Das bedeutet auch, dass sich der Datentyp des ersten Arguments nach dem Datentyp des ersten Formalparameters richten muss, der Datentyp des zweiten Arguments nach dem Datentyp des zweiten Formalparameters usw. Ist dies nicht der Fall, wenn also der im Funktionsaufruf angegebene Datentyp nicht mit dem Datentyp des entsprechenden Formalparameters übereinstimmt, so muss er zumindest implizit in den richtigen Datentyp konvertierbar sein. Andernfalls wird dies als Syntaxfehler erkannt, da der Compiler für jeden Funktionsaufruf ja eine entsprechende Typüberprüfung durchführt, wobei er sich entweder an der Definition der aufgerufenen Funktion orientiert oder an einer entsprechenden Deklaration (siehe Abschnitt 20.2 »Funktionsprototypen«, Kasten »Warum sich Ihr Compiler mit Prototypen zufrieden gibt«).

**Hinweis**

Da Ihren Compiler an den Funktionsprototypen allein die Datentypen der Parameter interessieren, dürfen Sie in der Funktionsdeklaration die Bezeichner der Parameter auch weglassen. Demnach können Sie die obige Funktion `function()` alternativ zu

```
void function(int a, int b);
```

auch wie folgt deklarieren:

```
void function(int, int);
```

Das geht so weit, dass sich die Bezeichner der Parameter – falls Sie diese angeben – sogar von denen in der Funktionsdefinition unterscheiden dürfen:

```
void function(int z1, int z2);
```

Wenn also der Kopf einer Funktion wie folgt aussieht

```
void f(char zeich, int nummer, double betrag, char c)
{
    ...
}
```

dann ist der Funktion, wenn sie verwendet wird, ein `char`-Wert, ein `int`-Wert, ein `double`-Wert und wieder ein `char`-Wert – bzw. Werte, die entsprechend konvertiert werden können – in dieser Reihenfolge zu übergeben:

```
f('Z', 6, 3.11, '$');
```

Natürlich dürfen beim Funktionsaufruf als aktuelle Parameter auch die Bezeichner von Variablen verwendet werden:

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

void sternchen(char zeichen, int anzahl)
{
    for (int i = 0; i < anzahl; i++)
        cout << zeichen;
}

int main(void)
{
    char c;
    cout << "Gib mir ein Zeichen: ";
    cin >> c;
    sternchen(c, 7);
    return 0;
}
```

Ausgabe:

```
Gib mir ein Zeichen: ?
???????
```

### 20.3.1 Wertübergabe

Beachtung verdient die Tatsache, dass die Funktionsblöcke von `sternchen()` und `main()` zwei verschiedene Gültigkeitsbereiche beschreiben. Das heißt, Variablen der einen Funktion können in der anderen nicht angesprochen werden.

#### Referenz

Zu Gültigkeitsbereich und Lebensdauer von Variablen siehe Kapitel 17, Abschnitt 17.5 »Gültigkeitsbereich von Variablen«.

Mit dem Aufruf

```
sternchen(c, 7);
```

wird also an die Funktion `sternchen()` nicht etwa die Variable `c` selbst übergeben, sondern nur der in ihr enthaltene Wert bzw. eine Kopie davon. Es handelt sich somit um eine bloße Wertübergabe, auch *call by value* genannt.

Demnach hat es keine Auswirkung auf die aufrufende Umgebung, wenn ein Parameter in der aufgerufenen Funktion verändert wird. Dies wird besonders deutlich, wenn die formalen und aktuellen Parameter gleichnamig sind:

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

void f(int a)
{
    a = 999;
    cout << a << endl;
}

int main(void)
{
    int a = 70;
    cout << "Wert von a vor Aufruf von f(): ";
    cout << a << endl;
    f(a);
    cout << "Wert von a nach Aufruf von f(): ";
    cout << a << endl;
    return 0;
}
```

Ausgabe:

```
Wert von a vor Aufruf von f(): 70
999
Wert von a nach Aufruf von f(): 70
```

Wie gesagt, müssen Bezeichner innerhalb eines Gültigkeitsbereichs eindeutig sein. Da die Funktion `f()` und die Funktion `main()` zwei sich nicht überschneidende Gültigkeitsbereiche darstellen, ist die Verwendung des Bezeichners `a` in beiden Funktionen gänzlich unproblematisch. Beachten Sie, dass es sich tatsächlich um zwei verschiedene Variablen handelt – die eine Variable ist lokal zur Funktion `f()`, die andere lokal zur Funktion `main()`. Die einzige Schnittstelle zwischen beiden Funktionen ist der Aufruf von `f()` in `main()` mit dem aktuellen Wert der lokalen Variablen `a` von `main()`.

Dabei geschieht Folgendes: Der Programmablauf beginnt in `main()`. Eine zu dieser Funktion lokale Variable `a` wird angelegt und mit dem Wert 70 initialisiert. Zur Kontrolle wird der Wert dieser Variablen ausgegeben:

```
Wert von a vor Aufruf von f(): 70
```

Mit dem Funktionsaufruf

```
f(a);
```

wird die Ausführung von `main()` angehalten und in die Funktion `f()` gesprungen. Dabei wird an die Funktion `f()` eine Kopie des Wertes der zu `main()` lokalen Variablen `a` übergeben. Nun wird für die Funktion `f()` eine lokale Variable `a` angelegt und mit dem Übergabewert initialisiert. Somit sind nunmehr zwei Variablen `a` existent. Beide besitzen zu diesem Zeitpunkt den Wert 70.

Mit der Zuweisung

```
a = 999;
```

erhält die lokale Variable `a` von `f()` einen neuen Wert, der im Anschluss daran ausgegeben wird. Die lokale Variable `a` von `main()` bleibt davon unberührt, sodass der Wert dieser Variablen nach wie vor 70 beträgt.

Nach Ausführung der letzten Anweisung im Funktionsrumpf von `f()` (`cout << a << endl;`) erfolgt der Rücksprung nach `main()`. Die lokale Variable `a` von `f()` wird mit dem Verlassen dieser Funktion gelöscht. Würde die Funktion `f()` ein zweites Mal aufgerufen, so würde für die Funktion eine neue Variable `a` angelegt – und mit dem in diesem Funktionsaufruf aktuell angegebenen Wert initialisiert.

In `main()` wird mit der dem Aufruf von `f()` unmittelbar folgenden Anweisung fortgefahren:

```
cout << "Wert von a nach Aufruf von f(): ";
```

Da die lokale Variable `a` von `main()` mit Ausführung der Funktion `f()`, wie oben dargestellt, nicht verändert wurde, lautet die Ausgabe nach wie vor 70.

```
cout << a << endl;
```

### 20.3.2 Arrays an Funktionen übergeben

Wie bereits erwähnt, werden Arrays über ihre Anfangsadressen verwaltet, wobei der Bezeichner eines Arrays für diese Anfangsadresse steht (Kapitel 19, Abschnitt 19.2 »Arrays und Adressen«). Dementsprechend erfolgt die Übergabe eines Arrays an eine aufgerufene Funktion nicht by value (als Wert), sondern by reference. Das heißt, es wird die Adresse des Arrays übergeben. Das Array wird damit der aufgerufenen Funktion tatsächlich bekannt gemacht. Das hat zur Folge, dass Manipulationen, die im Rumpf der aufgerufenen Funktion am Array stattfinden, sich auch in der aufrufenden Umgebung auswirken. Dies ist nicht weiter verwunderlich, da es sich ja um ein und dasselbe Array handelt:

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

void zunull(int *arr)
{
    int i;
    for (i = 0; i < 5; i++)
        arr[i] = 0;
}

int main(void)
{
    int meinArray[5] = {11, 22, 33, 44, 55}, i;
    cout << "Vorher: ";
    for (i = 0; i < 5; i++)
        cout << meinArray[i] << ' ';
    zunull(meinArray);
    cout << endl << "Nachher: ";
    for (i = 0; i < 5; i++)
        cout << meinArray[i] << ' ';
    return 0;
}
```

Ausgabe:

```
Vorher: 11 22 33 44 55
Nachher: 0 0 0 0 0
```

Eine Funktion, die beim Aufruf die Adresse eines Arrays erhalten soll, muss als entsprechenden Formalparameter einen so genannten »Zeiger« besitzen. Das ist eine spezielle Variable, die in der Lage ist, die Adresse eines Datenobjekts aufzunehmen. Um einen Zeiger zu deklarieren, gibt man den Datentyp dieses Datenobjekts an – hier also `int`, da es sich ja um ein Integerarray handelt – plus eines Sternchens (\*). Dabei kann das Sternchen entweder am Ende des Typbezeichners oder am Anfang des Bezeichners für das Datenobjekt stehen, also

```
int* arr
```

oder

```
int *arr
```

Falls es sich z.B. um ein `char`-Array handelt, schreibt man entsprechend

```
char* arr
```

bzw.

```
char *arr
```

Mit Zeigern werden wir uns in Kapitel 28 beschäftigen. Um Arrays an Funktionen zu übergeben bzw. entsprechende Funktionen zu definieren, ist es fürs Erste ausreichend zu wissen, wie Zeigervariablen vereinbart werden.

Es sei darauf hingewiesen, dass die Zeigervariable `arr` nach Aufruf von `zunull()` zwar die Adresse des Arrays `meinArray` enthält, jedoch keine weiteren Informationen. Diesbezüglich unterscheiden sich die Zeigervariable `arr` und der Bezeichner `meinArray`. Letzterer steht nicht nur für die Anfangsadresse des Arrays, sondern in gewisser Weise auch für das Array als solches. So lässt sich beispielsweise in der Funktion `main()` mit

```
sizeof(meinArray)
```

die Größe des Arrays `meinArray` in Byte bestimmen, nicht aber in der Funktion `zunull()` mit

```
sizeof(arr)
```

Letzter Ausdruck besitzt den Rückgabewert 4, da eine Zeigervariable in der Regel 4 Byte groß ist, dagegen liefert der erste Ausdruck den Wert 20, also die Größe des Integerarrays `meinArray` (das aus fünf Elementen besteht) in Byte.

**Hinweis**

Tatsächlich erfolgt vor der eigentlichen Übergabe der Adresse im Zuge des Funktionsaufrufs

```
zunull(meinArray);
```

eine Konvertierung des Parameters `meinArray` in einen Zeiger auf das erste Element des Arrays, nämlich vom Typ `int[5]` nach `int *`.

## 20.4 Rückgabewerte von Funktionen

Ihnen ist mittlerweile bekannt, dass Funktionen einen Wert zurückgeben können. Sie haben solche Funktionen ja bereits verwendet, wie z.B. die Funktion `rand()`. Was Sie noch nicht wissen ist, wie solche Funktionen definiert werden. Nehmen wir als einfaches Beispiel eine Funktion, die das Produkt von zwei ganzen Zahlen ausrechnet. Mit den jetzigen Mitteln könnten Sie es lediglich erreichen, dass die Funktion – nennen wir sie `prod()` – das Berechnungsergebnis ausgibt:

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

void prod(int a, int b)
{
    cout << a * b;
}

int main(void)
{
    prod(4, 5);
    return 0;
}
```

Ausgabe: 20

In einigen Fällen kann das auch den gewünschten Zweck erfüllen. Allerdings ist eine Funktion, die Ausgaben produziert, nur sehr eingeschränkt verwendbar. Dies gilt sogar für den Fall, dass die Funktion – zusätzlich – einen Rückgabewert liefert. Wenn man eine Funktion innerhalb komplexer Berechnungen einsetzt, dann möchte man in der Regel nicht, dass dies mit Ausgaben verbunden ist. In diesem Fall ist es erforderlich, dass die Funktion die gewünschte Information per Rückgabewert an den Aufrufer übermittelt, ansonsten aber keine weiteren Nebeneffekte produziert. Falls erforderlich, steht es dem Programmierer ja frei, den Rückgabewert der Funktion mit `cout` auszugeben.



Um für eine Funktion einen Rückgabewert festzulegen, müssen zwei Dinge beachtet werden:

- Im Kopf der Funktion ist zu Beginn der Datentyp des Rückgabewertes mit dem entsprechenden Typbezeichner anzugeben – void bedeutet, dass eine Funktion keinen Wert zurückliefert.
- Die Funktion muss mit der return-Anweisung unter Angabe eines Wertes, der im Datentyp mit dem im Funktionskopf angegebenen übereinstimmt bzw. in diesen konvertiert werden kann, beendet werden. Natürlich darf anstelle eines einfachen Wertes auch ein komplexer Ausdruck stehen.

Für unsere Funktion `prod()` ergibt sich somit folgende Definition:

```
int prod(int a, int b)
{
    return (a * b);
}
```

Die Klammern um den Ausdruck `a * b` dienen hier nur der Verdeutlichung, sind ansonsten jedoch nicht notwendig. Die Funktion liefert nun das Produkt beider Parameter als Rückgabewert:

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

int prod(int a, int b)
{
    return (a * b);
}

int main(void)
{
    cout << prod(3, 5);
    return 0;
}
```

Ausgabe: 15

Die `return`-Anweisung bewirkt das sofortige Verlassen einer Funktion. Beachten Sie, dass sie gegebenenfalls für mehrere Programmzweige verwendet werden muss:

```
double absolut(double betrag)
{
    if (betrag < 0)
        return (-betrag);
    else
        return betrag;
}
```

Bezüglich der Verwendung einer Funktion, die einen Wert zurückgibt, gilt:

Der Funktionsaufruf steht gleichzeitig für den Rückgabewert der Funktion. Das heißt, Sie dürfen ihn im Code überall dort verwenden, wo auch ein einfacher Wert dieses Datentyps stehen dürfte, also z.B. auf der rechten Seite einer Zuweisung, als Operand einer arithmetischen Operation und sogar als Argument für andere Funktionen. So könnte z.B. ein verschachtelter Funktionsaufruf von `prod()` wie folgt aussehen:

```
prod(3, prod(5, 4))
```

Obiger Ausdruck ergibt den Wert 60, was dem Rückgabewert des Funktionsaufrufs

```
prod(3, 20)
```

entspricht. Wobei der zweite Parameter von der inneren Funktion

```
prod(5, 4)
```

als Rückgabewert geliefert wird.

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

int prod(int a, int b)
{
    return (a * b);
}

int main(void)
{
    cout << prod(3, prod(5, 4));
    return 0;
}
```

Ausgabe: 60

Tatsächlich handelt es sich bei dem Rückgabewert einer Funktion um eine temporäre, namenlose Variable, die im Programm daher nur vorübergehend genutzt werden kann, nämlich in der Anweisung, in der der Funktionsaufruf steht.

### Hinweis

Anders als das in vielen anderen Programmiersprachen der Fall ist, müssen Rückgabewerte von Funktionen in C++ jedoch nicht zwingend verarbeitet werden. Dementsprechend wäre eine Anweisung wie

```
prod(53, 15);
```

denkbar, obwohl diese natürlich keinerlei Nutzen hat – der Ausdruck wird zwar ausgewertet, es geschieht jedoch nichts damit.

Falls eine Funktion mit `void` definiert ist, ist die Verwendung der `return`-Anweisung optional. Erlaubt ist also

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

oder

```
void f()
{
    // ...
    return;
}
```

Allerdings dürfen Sie `return` in diesem Fall keinen Wert mitgeben:

```
void f()
{
    // ...
    return 23; // FEHLER
}
```

Häufig ist der Einsatz der `return`-Anweisung notwendig, um eine Funktion in bestimmten Programmzweigen unverzüglich zu verlassen:

```
void spiel()
{
    char cancel;
    // ...
    cout << "(A)bbrechen?";
    cin >> cancel;
    if (cancel == 'A')
        return;
    // ...
}
```

### 20.4.1 Rückgabewert von `main()`

Der C++-Standard schreibt vor, dass die Funktion `main()` einen Integerwert zurückgeben muss. Obwohl dies praktisch jeder Compiler erlaubt, ist es deshalb streng genommen nicht korrekt, für `main()` als Rückgabebetyp `void` zu verwenden. Es hat sich eingebürgert, dass der Rückgabewert 0 standardmäßig eine fehlerfreie Programmausführung anzeigt:

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

Genau genommen legt der C++-Standard die Definition von `main()` in der Weise fest, dass die `return`-Anweisung nicht zwingend notwendig ist. Danach gibt eine mit dem Rückgabetyt `int` definierte Funktion `main()` auch dann einen Wert – und zwar `0` – zurück, wenn sie nicht explizit mit der `return`-Anweisung beendet wird. Da jedoch einige Compiler damit Schwierigkeiten haben, ist es besser, die `return`-Anweisung zu verwenden.

### Achtung

Wie gesagt, betrifft diese Besonderheit ausschließlich die Funktion `main()`. Für andere Funktionen gilt: Wenn Sie im Kopf einer Funktion festlegen, dass diese einen Wert zurückgibt, dann muss die Funktion mit einer `return`-Anweisung unter Angabe eines passenden Rückgabewertes beendet werden. Andernfalls beschwert sich Ihr Compiler mit einer Fehlermeldung.

Es sei darauf hingewiesen, dass es sich beim Rückgabewert von `main()` gewöhnlich um eine reine Formsache handelt. Der »Empfänger« ist ja nicht eine andere C++-Funktion, sondern das Betriebssystem.

## 20.4.2 Vorgabeargumente

Es ist möglich, für einzelne Parameter Standardwerte als Vorgabeargumente festzulegen. Dann werden beim Funktionsaufruf für nicht angegebene Argumente die entsprechenden Standardwerte eingesetzt:

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

double kubik(double a, double b = 1, double c = 1)
{
    return (a * b * c);
}

int main(void)
{
    cout << kubik(17, 2);
    return 0;
}
```

Ausgabe: 34

Die obige Funktion `kubik()` lässt sich also entweder mit einem, mit zwei oder mit drei Parametern aufrufen. Für jedes Argument, das in der Aufrufliste nicht angegeben wird, setzt der Compiler das entsprechende Vorgabeargument ein, im obigen Beispiel für den dritten Parameter den Wert 1.

Erlaubt ist das Setzen von Vorgabeargumenten für Parameter sowohl in der Definition als auch in der Deklaration einer Funktion, jedoch nicht in beiden zugleich. Falls Funktionsprototypen verwendet werden, ist es jedoch üblich, die Vorgabeargumente in diesen anzugeben:

```
double kubik(double a, double b = 1, double c = 1);
```

bzw.

```
double kubik(double, double = 1, double = 1);
```

Zwar steht es dem Programmierer grundsätzlich frei, für welche Parameter er Vorgabeargumente vergibt. Falls jedoch für einen Parameter ein Vorgabeargument gesetzt ist, muss dies für alle folgenden ebenfalls geschehen.

```
double func(double a = 2, double b) // FEHLER
```

Bezüglich obiger Deklaration wird Ihr Compiler das Fehlen eines Vorgabearguments für den zweiten Parameter anmahnen.

## 20.5 Überladen von Funktionen

Angenommen, Sie möchten eine Funktion erstellen, die verschiedene Datentypen verarbeiten kann. Denkbar ist eine Berechnungsfunktion, der Integerwerte, aber auch Gleitkommawerte, gegebenenfalls auch Datentypen, bei denen keine implizite Konvertierung möglich ist, z.B. `char`-Werte, übermittelt werden können.

Die einzige Möglichkeit besteht darin, mehrere Funktionen zu definieren. Da eine Funktion normalerweise einen eindeutigen Namen haben muss (schließlich muss Ihr Compiler diese auseinander halten können), ist es denkbar, dass der Datentyp mit in den Funktionsnamen integriert wird, um zu dokumentieren, welcher Datentyp erwartet wird.

Die Funktionen könnten dann z.B. `func_int()`, `func_double()` und `func_char()` heißen. Besonders elegant ist diese Lösung aber nicht, und doch war es in C die einzige Möglichkeit, eine universelle – datentypunabhängige – Funktion zu erzeugen (streng genommen handelt es sich natürlich weiterhin um mehrere Funktionen, aber für den Programmierer haben sie den Charakter einer einzigen Funktion).

Schöner wäre es aber, nur genau einen Funktionsnamen zu haben, um den Charakter zu unterstreichen, dass jede dieser Funktionen den gleichen Zweck erfüllt, nur eben für verschiedene Datentypen. Genau dies erlaubt C++: Unter bestimmten Voraussetzungen

dürfen Sie mehrere Funktionen gleichen Namens definieren. In diesem Zusammenhang spricht man vom »Überladen« dieser Funktionen.

### Hinweis

Wie gesagt, ist es eigentlich der Bezeichner der Funktion, der überladen wird.

Dabei müssen sich die überladenen Funktionen in ihrer Parameterliste voneinander unterscheiden, sei es in Anzahl oder Datentyp der einzelnen Parameter. Ein bloßer Unterschied im Rückgabetyt ist nicht ausreichend, wenngleich es nicht notwendig ist, dass überladene Funktionen im Rückgabetyt übereinstimmen.

Als Beispiel einige Prototypen für eine überladene Funktion `fname()`. Wie gesagt, spielen die Bezeichner von Parametern bei der Betrachtung keine Rolle. Um dies deutlich zu machen, sind sie in den folgenden Deklarationen nicht genannt:

```
int fname(int, int);
int fname(int, int, char);
int fname(int, double);
int fname(double, int);
```

Wie an den letzten zwei Deklarationen zu sehen ist, genügt es, wenn nur ein Parameter im Datentyp abweicht.

### Hinweis

Bezeichner plus Parameterliste werden als »Signatur« einer Funktion bezeichnet, da beide zusammen eine Funktion eindeutig identifizieren. Der Rückgabewert der Funktion sowie die Bezeichner der Parameter zählen demnach nicht dazu.

Ein gültiger Aufruf der Funktion `fname()` – eigentlich müsste man sagen »einer der Funktionen `fname()`«, da es sich ja tatsächlich um mehrere handelt – könnte nun wie folgt aussehen:

```
fname(2, 4);
fname(33, -2, 'A');
fname(49, 0.57);
fname(-1.99, 103);
```

Beachten Sie, dass sich der Programmierer im Grunde nicht darum zu kümmern braucht, wie viele Funktionen `fname()` tatsächlich definiert sind, wenn er sie verwendet. Er muss nur wissen, mit welchen Argumenten er die Funktion aufrufen darf. In diesem Sinne kann eine überladene Funktion tatsächlich wie eine einzige behandelt werden.

**Hinweis**

Aus dem gleichen Grund kann der <<-Operator von cout jeden beliebigen elementaren Datentyp bzw. string übernehmen. Wie an anderer Stelle schon erwähnt, handelt es sich bei diesem eigentlich um eine so genannte Operatorfunktion – für die entsprechend viele Variationen definiert sind.

Besondere Aufmerksamkeit erfordert das Setzen von Vorgabeargumenten bei überladenen Funktionen. Hier muss sichergestellt sein, dass die Zuordnung eindeutig bleibt. Bei Funktionsdefinitionen wie

```
void f(int a, int b = 3)
{
    // ...
}
```

und

```
void f(int zahl)
{
    // ...
}
```

würde man spätestens beim Aufruf

```
f(27)
```

eine Fehlermeldung erhalten, da der Compiler in diesem Fall nicht ermitteln kann, welche der beiden Funktionen f() gemeint ist.

## 20.6 Rekursionen

Man spricht von einer Rekursion, wenn eine Funktion sich selbst aufruft. Dabei muss allerdings gewährleistet sein, dass die Rekursion irgendwann abbricht. Als Beispiel hierzu eine rekursive Variante einer Funktion zur Berechnung der Fakultät einer Zahl:

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

int fakul(int n)
{
    if (n == 0)
        return 1;
```

```

    else
        return n * fakul(n - 1);
}

int main(void)
{
    int f;
    cin >> f;
    cout << f << "! = " << fakul(f);
    return 0;
}

```

Ausgabe:

```

5
5! = 120

```

## Referenz

Eine nicht rekursive Implementation dieses Programms finden Sie in Kapitel 17, Abschnitt 17.1.2 »Fakultät berechnen«.

Lassen Sie uns zur Verdeutlichung den Geschehensablauf für den Eingabewert 2 durchspielen:

```
cin >> f;
```

Dann wird die Funktion `fakul()` in `main()` mit Ausführung der Anweisung

```
cout << f << "! = " << fakul(f);
```

mit diesem Wert aufgerufen.

```
fakul(2)
```

Nun erhält der Parameter `n` diesen Wert und die Bedingungsprüfung

```
if (2 == 0)
```

ergibt `false`. Daher wird die Anweisung des `else`-Zweiges ausgeführt, in dem die Funktion sich selbst aufruft und zwar mit dem Argument `n - 1`, in diesem Fall also mit dem Wert 1.

```
return 2 * fakul(1);
```



Das heißt, die Ausführung der obigen Anweisung wird angehalten und es erfolgt der Sprung in eine weitere Funktion `fakul()` mit dem Übergabewert 1. Deren Parameter `n` wird mit diesem Wert initialisiert, die Bedingungsprüfung

```
if (1 == 0)
```

ergibt `false` und die Programmausführung landet im `else`-Zweig dieser Funktion `fakul()`. Mit der Anweisung

```
return 1 * fakul(0);
```

wird auch die Abarbeitung in der zweiten Funktion `fakul()` angehalten und eine dritte Funktion `fakul()` aufgerufen mit dem Parameter `n - 1`, nunmehr 0. Mithin existieren zu diesem Zeitpunkt drei Funktionen `fakul()`, von denen zwei – ebenso wie `main()` – in Warteposition sind.

In der zuletzt aufgerufenen Funktion `fakul()` ergibt die Bedingungsprüfung `true` – der Parameter `n` dieser Funktion wurde ja bei Aufruf mit dem Übergabewert 0 initialisiert:

```
if (0 == 0)
```

Daher wird mit

```
return 1;
```

die Funktion mit dem Rückgabewert 1 beendet. Es erfolgt der Rücksprung in die aufrufende Umgebung, also in die zweite Funktion `fakul()`. Dort wird der Ausdruck `1 * fakul(0)` in der Anweisung

```
return 1 * fakul(0);
```

zu 1 ausgewertet und mit

```
return 1;
```

auch diese zweite Funktion `fakul()` beendet. Der Rücksprung in den Aufrufer – die Funktion `fakul()`, die als allererste von `main()` aufgerufen wurde – mit dem Rückgabewert 1 ergibt für die Anweisung

```
return 2 * fakul(1);
```

den Rückgabewert 2. Dieser Wert wird nun von `main()` in der Anweisung

```
cout << f << "! = " << fakul(f);
```

als Ergebnis der Berechnung ausgegeben.

Hier die entsprechende Ausgabe:

```
2
2! = 2
```

### Hinweis

Es sei darauf hingewiesen, dass sich rekursive Lösungen mitunter zwar sehr elegant kodieren lassen, wie am obigen Beispiel zu sehen, oft aber bezüglich Ausführungsgeschwindigkeit und vor allem Speicherbedarf sehr ineffizient sind. Zwar lässt sich jedes Problem theoretisch auch ohne Rekursion lösen, aber es gibt Fälle, bei denen man immer auf Rekursionen zurückgreifen wird, da eine alternative Lösung zu kompliziert wäre. Beispielsweise werden Rekursionen eingesetzt, um Verzeichnisbäume (z.B. die Verzeichnisstruktur der Festplatte) einzulesen. Da Verzeichnisse Unterverzeichnisse enthalten können, die gegebenenfalls wiederum Unterverzeichnisse aufweisen usw. bietet sich eine Rekursion geradezu an. Man wird dann eine Funktion zum Lesen genau eines Verzeichnisses erstellen, die sich selbst aufruft, wenn sie auf ein Unterverzeichnis trifft. Die Abbruchbedingung wäre dann, dass ein Verzeichnis kein weiteres Unterverzeichnis aufweist.

## 20.7 inline-Funktionen

Sie können eine Funktion mit dem entsprechenden Schlüsselwort als `inline` definieren und Ihrem Compiler damit empfehlen, diese gar nicht als solche zu behandeln. Ihr Compiler wird dann – falls er der Empfehlung folgt – an jeder Stelle im Code, an der die `inline`-Funktion aufgerufen wird, den Aufruf durch die im Block der Funktion stehenden Anweisungen in einer dem Kontext angepassten Form ersetzen. Es verhält sich dann gewissermaßen so, als ob die Anweisungen der aufgerufenen `inline`-Funktion in der aufrufenden Funktion kodiert sind.

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

inline double func(double a, double b)
{
    return (a * b);
}

int main(void)
{
    cout << func(2.5, 1.5) << endl;
    cout << func(1.3, 1.9) + func(0.5, 3.9);
    return 0;
}
```

Falls Ihr Compiler die Funktion `func()` als `inline` behandelt – wovon hier auszugehen ist –, wird er die entsprechenden Funktionsaufrufe in `main()` wie folgt ersetzen:

```
...
int main(void)
{
    cout << (2.5 * 1.5);
    cout << (1.3 * 1.9) + (0.5 * 3.9);
    return 0;
}
```

Die Verwendung von `inline`-Funktionen bringt in der Regel eine – meist jedoch nur geringe – Steigerung der Ausführungsgeschwindigkeit mit sich. Der Grund ist folgender: Ohne `inline` erzeugt der Compiler für jede Funktion nur genau einen Satz von Anweisungen. Als Folge davon muss für jeden Funktionsaufruf an die Stelle im Speicher, an der sich die Funktion befindet, gesprungen und auch wieder zum Aufrufer zurückgesprungen werden. Diese Sprünge kosten ein klein wenig Zeit. Bei sehr vielen Aufrufen ein und derselben Funktion kann dies durchaus ins Gewicht fallen, also mit einer geringeren Geschwindigkeit verbunden sein. Bei `inline` entfallen diese Sprünge, wodurch gegebenenfalls die Ausführungsgeschwindigkeit steigt.

Allerdings vergrößern die beschriebenen Ersetzungen im Allgemeinen den Umfang der ausführbaren Datei und deren Speicherverbrauch. Der wesentlich höhere Speicherverbrauch steht dann möglicherweise in keinem Verhältnis mehr zum – eher bescheidenen – Gewinn an höherer Ausführungsgeschwindigkeit.

Dieser negative Effekt macht sich umso stärker bemerkbar, je größer die als `inline` behandelte Funktion ist, das heißt je mehr Anweisungen diese im Funktionsrumpf enthält. Die Spezifizierung mit dem Schlüsselwort `inline` eignet sich daher in der Regel nur für sehr kleine Funktionen mit ein oder zwei Anweisungen.

Ein weiterer Nachteil von `inline`-Funktionen besteht darin, dass Ihrem Compiler deren Definition beim Aufruf bekannt sein muss – sonst kann er die zugehörigen Anweisungen ja nicht kopieren. Die reine Deklaration einer `inline`-Funktion ist in dieser Hinsicht daher nicht ausreichend.

### Hinweis

Bedenken Sie, dass es sich mit der Spezifizierung als `inline` lediglich um einen Empfehlung handelt. Sehr große oder komplizierte Funktionen, die z.B. Schleifenkonstrukte enthalten, wird Ihr Compiler vermutlich nicht als `inline` akzeptieren. Das bedeutet, das Schlüsselwort `inline` wird ignoriert und die Funktion wird so behandelt wie eine gewöhnliche Funktion.

**Hinweis**

Die Kenntnis von inline-Funktionen soll für Sie vor allem im Hinblick auf Klassen und deren Methoden von Bedeutung sein. Diesbezüglich ist es wichtig zu wissen, dass Methoden, die innerhalb von Klassen definiert sind, automatisch als inline spezifiziert angesehen werden (zu Klassen und Methoden siehe Kapitel 24 »Klassen und Objekte«).

## 20.8 Globale Variablen

Der Arbeitsspeicher ist in verschiedene Bereiche unterteilt, von denen uns vor allem der so genannte Heap, der Stack sowie ein globaler Namensbereich interessieren sollen. Die einzelnen Bereiche unterscheiden sich in Bezug auf die Verwaltung von Datenobjekten.

Datenobjekte auf dem Heap werden dynamisch verwaltet. Mehr dazu erfahren Sie in Kapitel 28, Abschnitt 28.6 »Speicherplatz dynamisch anfordern«. Lokale Variablen der Speicherklasse auto werden auf dem Stack angelegt. Dieser funktioniert nach dem so genannten LIFO-Prinzip, was für »Last In First Out« steht und auf den Sachverhalt hinweist, dass Datenobjekte, die zuletzt angelegt werden, als Erstes wieder entfernt werden.

```
{  
    int a;  
    {  
        int b;  
    }  
}
```

So wird, bezogen auf obiges Codefragment, die Variable b nach der Variablen a angelegt. Das heißt, b wird dem Stack als Letztes hinzugefügt. Mit Verlassen des inneren Blocks endet auch die Lebensdauer der Variablen b, diese wird vom Stack – zu Deutsch »Stapel(speicher)« – wieder entfernt. Erst danach, mit Verlassen des äußeren Blocks, trifft a dasselbe Schicksal.

Variablen mit der Speicherklasse static werden im globalen Namensbereich verwaltet. Ihre Lebensdauer erstreckt sich über das ganze Programm. Außerdem werden sie bei ihrer Vereinbarung automatisch mit dem Anfangswert 0 versehen, char-Variablen mit dem Nullbyte. Ein Beispiel für Variablen dieser Speicherklasse haben Sie in Kapitel 17, Abschnitt 17.5.2 »static-Variablen« kennen gelernt. Dabei handelte es sich zwar aufgrund der Speicherklasse static hinsichtlich ihrer Lebensdauer um globale, bezüglich ihres Gültigkeitsbereichs jedoch um lokale Variablen.

Nun ist es möglich, Variablen dieser Speicherklasse global zu vereinbaren. Dies geschieht außerhalb von Funktionen, am besten zu Beginn der Quelldatei. Die Variable ist dann in allen folgenden Funktionen bekannt:

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

int ueberall;

void f(void)
{
    ueberall = 11;
    // ...
}

int main(void)
{
    f();
    cout << ueberall;
    return 0;
}
```

Ausgabe: 11

In den Funktionen `f()` und `main()` kann somit auf die globale Variable `ueberall` direkt zugegriffen werden.

### Hinweis

Natürlich dürfen Sie eine globale Variable auch selbst initialisieren:

```
int ueberall = 99;
```

Der Gültigkeitsbereich einer so deklarierten Variablen erstreckt sich per Default auf die ganze Quellcodedatei. Um diesen auf etwaige weitere Quelldateien desselben Programms zu erweitern, ist es erforderlich, die Variable dort unter Angabe des Schlüsselwortes `extern` bekannt zu machen:

```
extern int ueberall;
```

Angenommen, Ihr Programm besteht aus drei Quellcodedateien und in einer davon ist eine Variable `x` wie folgt vereinbart:

```
double x = 22.37;
```

Dann machen Sie die Variable in den anderen beiden Quelldateien jeweils am Anfang mit

```
extern double x;
```

bekannt. Nun können alle Funktionen dieser drei Dateien auf die Variable `x` zugreifen.

Wobei es natürlich für diese Überlegung keine Rolle spielt, ob `x` bei der Deklaration initialisiert wird oder nicht. Allerdings dürfen Sie bei der bloßen Bekanntmachung keine Zuweisung durchführen:

```
extern double x = 56.2; // FEHLER
```

Beachten Sie, dass eine globale Variable die Speicherklasse `static` besitzt, ohne dass bei ihrer Vereinbarung das entsprechende Schlüsselwort angegeben werden muss. Vielmehr besitzt das Schlüsselwort `static` in diesem Kontext eine ganz andere Bedeutung: Es würde den Geltungsbereich der Variablen ausdrücklich auf eine Quelldatei beschränken, sodass eine wie folgt deklarierte globale Variable nicht mehr in anderen Quelldateien bekannt gemacht werden kann:

```
static double x = 56.2; // Variable kann  
// nur in dieser Datei angesprochen werden
```

Die Tatsache, dass die Variable `x` außerhalb von Funktionen vereinbart ist, bewirkt also, dass diese Variable die Speicherklasse `static` erhält. Das bei der Deklaration verwendete Schlüsselwort `static` hat hier zur Folge, dass der Gültigkeitsbereich der globalen Variablen ein für alle Mal auf die Datei beschränkt ist, in der sie vereinbart ist. Demnach kann die so vereinbarte Variable `x` in anderen Dateien nicht mit `extern` bekannt gemacht werden.

### **Tipp**

Obwohl der Gebrauch von globalen Variablen auf den ersten Blick sehr bequem und damit vorteilhaft erscheint, sollten Sie globale Variablen möglichst vermeiden bzw. auf das absolut erforderliche Minimum reduzieren. Denken Sie daran, dass es bei einer Vielzahl von Funktionen nur sehr schwer nachzuvollziehen ist, wie eine Variable in den einzelnen Funktionen verändert wird. Ein mit globalen Elementen durchsetzter Code wird damit sehr schnell fehlerträchtig. Bemühen Sie sich dagegen, beim Programmieren in abgeschlossenen Einheiten zu denken. Dieser Gedanke liegt letzten Endes der Aufteilung des Codes in Funktionen sowie dem ganzen Konzept der objekt-orientierten Programmierung zugrunde.

# 21 Eine Funktionsbibliothek

Wir werden nun im Hinblick auf ein im nächsten Kapitel zu programmierendes Lotteriespiel einige nützliche Funktionen definieren. Es bietet sich an, diese in einer separaten Quelldatei zu speichern.

## Hinweis

Es sei darauf hingewiesen, dass mit einer Funktionsbibliothek in der Regel nicht nur eine, sondern eine Vielzahl von Quelldateien, in denen sich jeweils mehrere Funktionsdefinitionen befinden, assoziiert wird.

## 21.1 Funktion ArrMinIndex()

Als Erstes wollen wir eine Funktion `ArrMinIndex()` schreiben, die aus einem Array von ganzen Zahlen den Index desjenigen Elements zurückgibt, das den kleinsten Wert enthält.

Rückgabewert ist also nicht der kleinste Wert selbst, sondern die Position dieses Wertes im Array, also der entsprechende Index.

Wir werden die Funktion im Weiteren so überarbeiten, dass sie auch auf beliebige Teilausschnitte von Arrays angewendet werden kann. Im Moment setzen wir jedoch voraus, dass es sich bei der Übergabe um ganze Arrays handelt, der Index also bei 0 beginnt. Die Funktion muss natürlich wissen, um welches Array es sich handelt und wie groß dieses ist. Als Parameter ergeben sich daher einmal die Adresse eines Arrays sowie entweder die Elementzahl oder die Indexobergrenze dieses Arrays. Entscheiden wir uns für die Indexobergrenze. Der Datentyp des Rückgabewertes ist `int`. Somit ergibt sich als Funktionsgerüst:

```
int ArrMinIndex(int *arr, int ogrenze)
{
    // ...
}
```

Vorausgesetzt, die Implementation wäre bereits erfolgt, dann würde sich ein Aufruf der Funktion `ArrMinIndex()` wie folgt darstellen:

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

```
int main(void)
{
    int meinArray[5] = {237, 0, 22, -44, 7};
    cout << ArrMinIndex(meinArray, 4);
    return 0;
}
```

Ausgabe: 3

Die kleinste Zahl des Arrays (-44) befindet sich an der Position mit dem Index 3, daher die Ausgabe 3.

### Tipp

Es empfiehlt sich, eine neue Funktion in einer entsprechenden Testumgebung zu entwickeln und erst im Anschluss daran in ein dafür vorgesehenes Modul zu kopieren.

Um einen passenden Algorithmus zu finden, ist es sinnvoll, sich ganz einfach vorzustellen, wie man bei der Lösung einer entsprechenden Aufgabe selbst vorgehen würde. Angenommen Sie haben z.B. eine Zahlenfolge wie

33   45   392   -3   44   -23   9633   -4

vor sich und müssten daraus den kleinsten Wert bzw. dessen Index bestimmen. Dann würden Sie vermutlich intuitiv die erste Zahl betrachten und diese zunächst mit der zweiten vergleichen. Da die erste Zahl hier kleiner ist, würden Sie sich diese bzw. deren Index 0 merken. Danach würden Sie die 33 mit der dritten Zahl vergleichen. Da die erste Zahl wiederum kleiner als diese Zahl (392) ist, bleibt alles beim Alten. Sie halten nach wie vor Index und Größe der ersten Zahl fest. Die beiden anderen sind ja nicht mehr von Interesse, da bereits festgestellt wurde, dass sie größer sind. Im nächsten Schritt vergleichen Sie die bis dato festgehaltene Zahl – also die erste – mit der vierten (-3). Diese ist nun kleiner als 33. Das heißt, Sie verwerfen die 33 samt Index 0 und halten jetzt die Zahl -3 zusammen mit der Indexangabe 3 fest. Dann vergleichen Sie -3 mit der folgenden Zahl usw. Wenn Sie die Zahlenreihe auf diese Weise vollständig betrachtet haben, halten Sie automatisch die kleinste Zahl bzw. deren Index in Händen.

Allein durch diese Überlegung ist bereits ein passender Algorithmus gefunden. Nun gilt es, dieses Rezept in Programmcode umzusetzen. Dazu durchlaufen wir das zu betrachtende Array in einer for-Schleife und ziehen eine Hilfsvariable `min_index` hinzu, um den jeweils kleinsten Index festzuhalten:

```
int ArrMinIndex(int *arr, int ogrenze)
{
    int min_index = 0;
    for(int i = 1; i <= ogrenze; i++)
    {
```



```

        if(arr[i] < arr[min_index])
        {
            min_index = i;
        }
    }
    return min_index;
}

```

Die Variable `min_index` wird mit dem Index des ersten Arrayelements, also mit 0, initialisiert. Danach wird das Array in der oben beschriebenen Weise durchlaufen, wobei in der `if`-Bedingung die Elemente des Arrays – beginnend mit dem zweiten Element – der Reihe nach mit dem jeweils aktuell kleinsten Wert verglichen werden. Ist der Wert des betrachteten Elements (`arr[i]`) kleiner als der bis dato kleinste Wert (`arr[min_index]`), so erhält die Variable `min_index` den Index des kleineren Elements, also `i`.

Die Funktion in der obigen Fassung ist nun anwendbar auf Arrays, aber auch auf Ausschnitte davon, je nachdem, welchen Wert man für `ogrenze` beim Aufruf übergibt:

```

#include <iostream>
using namespace std;

int ArrMinIndex(int *arr, int ogrenze)
{
    int min_index = 0;
    for(int i = 1; i <= ogrenze; i++)
    {
        if(arr[i] < arr[min_index])
        {
            min_index = i;
        }
    }
    return min_index;
}

int main(void)
{
    int meinArray[5] = {237, 0, 22, -44, 7};
    cout << ArrMinIndex(meinArray, 2);
    return 0;
}

```

Ausgabe: 1

Mit dem Aufruf

```
ArrMinIndex(meinArray, 2)
```

wird nicht das ganze Array ausgewertet, sondern nur die ersten drei Elemente des Arrays. Dementsprechend ist die Ausgabe 1, da sich an dieser Position – bezogen auf die Elemente `meinArray[0]`, `meinArray[1]` und `meinArray[2]` – der kleinste Wert (0) befindet.

Allerdings beginnt die Betrachtung stets mit dem ersten Element. Um die Funktion nun auch bezüglich der Untergrenze flexibel zu halten, ist es notwendig, einen dritten Parameter zu definieren und diesen im Anweisungsteil anstelle des Indexwertes 0 einzusetzen:

```
int ArrMinIndex(int *arr, int ugrenze, int ogrenze)
{
    int min_index = ugrenze;
    for(int i = ugrenze + 1; i <= ogrenze; i++)
    {
        if(arr[i] < arr[min_index])
        {
            min_index = i;
        }
    }
    return min_index;
}
```

Nun können Sie sich von der Funktion `ArrMinIndex()` den Index der kleinsten Zahl eines beliebigen Teilausschnitts eines Arrays zurückgeben lassen:

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

int ArrMinIndex(int *arr, int ugrenze, int ogrenze)
{
    int min_index = ugrenze;
    for(int i = ugrenze + 1; i <= ogrenze; i++)
    {
        if(arr[i] < arr[min_index])
        {
            min_index = i;
        }
    }
    return min_index;
}

int main(void)
{
    int meinArray[7] = {8, -22, 7, 14, 64, 28, -10};
    cout << ArrMinIndex(meinArray, 2, 5);
    return 0;
}
```

Ausgabe: 2

Die Ausgabe ist korrekt, da 7 der kleinste Wert im Bereich von `meinArray[2]` bis `meinArray[5]` ist.

## 21.2 Funktion `sortiereArr()`

Wir werden nun daran gehen, eine Funktion `sortiereArr()` zu erstellen, die ein Array von Integerzahlen nach ihrer Größe sortiert. Der Funktion sollen bei Aufruf die Adresse sowie die Elementzahl eines Arrays übergeben werden.

Wir legen unseren folgenden Überlegungen die Existenz der Funktion `ArrMinIndex()` zugrunde. Ein geeigneter Algorithmus zum Sortieren eines Arrays von Zahlen in aufsteigender Reihenfolge ist, jeweils den Index des kleinsten Wertes eines bestimmten Teilbereichs zu ermitteln und diesen Wert mit der Anfangsposition des betrachteten Teilbereichs zu vertauschen. Lassen Sie uns das an der Zahlenfolge

56   545   -4   76   81   23

veranschaulichen. Ruft man die Funktion `ArrMinIndex()` für ein entsprechendes Array von Zahlen auf, so erhält man den Wert 2 zurück, der ja hier dem Index der kleinsten Zahl (-4) entspricht. Nun vertauscht man die Werte an den Positionen 2 und 0, womit sich die Zahlenfolge

-4   545   56   76   81   23

ergibt. Nun wendet man die Funktion `ArrMinIndex()` auf den übrigen Teilbereich

545   56   76   81   23

an und erhält als Rückgabewert die Position der Zahl 23. Diese wird nun wiederum mit dem Wert an der ersten Position des betrachteten Teilbereichs vertauscht:

-4   23   56   76   81   545

Danach ruft man die Funktion `ArrMinIndex()` für den Teilbereich

56   76   81   545

auf usw.

Hier die entsprechende Implementation der Funktion `sortiereArr()`:

```
void sortiereArr(int *arr, int elemente)
{
    int ablage;
    for(int i=0; i < elemente; i++)
    {
        ablage = arr[ArrMinIndex(arr, i, elemente - 1)];
```

```

        arr[ArrMinIndex(arr, i, elemente-1)] = arr[i];
        arr[i] = ablage;
    }
}

```

Beachten Sie, dass die Funktion keinen Wert zurückgibt.

Der jeweils zu vertauschende Wert lässt sich mit

```
arr[ArrMinIndex(arr, i, elemente - 1)]
```

ermitteln, wobei

```
ArrMinIndex(arr, i, elemente - 1)
```

für den Rückgabewert der Funktion `ArrMinIndex()` steht, also für den Index des kleinsten Wertes des jeweils betrachteten Teilbereichs. Beachten Sie, dass der Wert `i` für die Untergrenze mit jedem Schleifendurchgang ansteigt, sodass die betrachteten Teilbereiche immer kleiner werden – die Obergrenze bleibt ja konstant. Da die Variable `elemente` die Elementzahl des übergebenen Arrays enthält, errechnet sich der höchste Index aus `elemente - 1`.

Der zu vertauschende Wert `arr[ArrMinIndex(arr, i, elemente - 1)]` muss in der Variablen `ablage` zwischengespeichert werden, da er sonst mit der folgenden Zuweisung

```
arr[ArrMinIndex(arr, i, elemente-1)] = arr[i];
```

verloren ginge.

Eine Testumgebung für die neue Funktion könnte nun wie folgt aussehen. Es empfiehlt sich, die Funktionen bzw. die entsprechenden Prototypen mit kurzen Kommentaren bezüglich ihrer Verwendungsweise zu versehen:

```

#include <iostream>
using namespace std;

void sortiereArr(int *arr, int elemente);
int ArrMinIndex(int *arr, int ugrenze, int ogrenze);

void sortiereArr(int *arr, int elemente)
// sortiert ein Array aufsteigend
{
    int ablage;
    int i;
    for(i=0; i < elemente; i++)
    {
        ablage = arr[ArrMinIndex(arr, i, elemente - 1)];
    }
}

```

```

        arr[ArrMinIndex(arr, i, elemente-1)] = arr[i];
        arr[i] = ablage;
    }
}

int ArrMinIndex(int *arr, int ugrenze, int ogrenze)
// gibt den kleinsten Wert eines Arrays
// im Bereich ugrenze bis ogrenze zurück
{
    int min_index = ugrenze;
    for(int i = ugrenze + 1; i <= ogrenze; i++)
    {
        if(arr[i] < arr[min_index])
        {
            min_index = i;
        }
    }
    return min_index;
}

int main(void)
{
    int meinArray[7] = {8, -22, 7, 14, 64, 28, -10};
    int i;
    for (i = 0; i < 7; i++)
        cout << meinArray[i] << " ";
    cout << endl;
    sortiereArr(meinArray, 7);
    cout << "Sortiert: " << endl;
    for (i = 0; i < 7; i++)
        cout << meinArray[i] << " ";
    return 0;
}

```

Ausgabe:

```

8 -22 7 14 64 28 -10
Sortiert:
-22 -10 7 8 14 28 64

```

Wie die Ausgabe zeigt, sind nach Aufruf von `sortiereArr()` die Zahlen im Array `meinArray` aufsteigend sortiert.

**Hinweis**

Der hier vorgestellte Algorithmus zum Sortieren von Arrays ist leicht nachzuvollziehen und für unsere Zwecke vollkommen ausreichend. Es sei aber darauf hingewiesen, dass er sich nur bedingt für die Sortierung größerer Arrays eignet, da die Laufzeit des Algorithmus exponentiell mit der Größe des Arrays steigt. Ein weitaus effizienterer Algorithmus ist zum Beispiel das Quicksort-Verfahren von C.A.R. Hoare, das unter anderem von der C-Funktion `qsort()` aus `cstdlib` verwendet wird.

## 21.3 Funktion `doppelte()`

Die Funktion `doppelte()` soll ein Integerarray auf doppelte Werte überprüfen. Dabei soll es keine Rolle spielen, wie viele Werte gegebenenfalls übereinstimmen. Die Funktion soll lediglich feststellen, ob in einem Array doppelte Werte vorkommen oder nicht. Entsprechend soll der Rückgabewert 1 sein, falls doppelte Werte vorhanden sind, ansonsten 0.

Der Funktion sollen beim Aufruf die Adresse eines Arrays sowie dessen Elementzahl übergeben werden. Vorausgesetzt, die Funktion sei bereits implementiert, könnte sie etwa wie folgt verwendet werden;

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

int main(void)
{
    int meinArray[5] = {98, 15, 7, 15, 64};
    if (doppelte(meinArray, 5))
        cout << "Doppelte Werte im Array";
    else
        cout << "Keine doppelten Werte im Array";
    return 0;
}
```

Ausgabe:

```
Doppelte Werte im Array
```

**Hinweis**

Die if-Bedingung

```
if (doppelte(meinArray, 5))
```

entspricht

```
if (doppelte(meinArray, 5) != 0)
```

Die Aufgabe lässt sich verhältnismäßig einfach lösen, indem man zuerst das Array sortiert und anschließend alle benachbarten Werte miteinander vergleicht:

```
int doppelte(int *arr, int elemente)
{
    sortiereArr(arr, elemente);
    for (int i = 0; i < elemente - 1; i++)
    {
        if (arr[i] == arr[i + 1])
            return 1;
    }
    return 0;
}
```

Beachten Sie, dass die Schleifenbedingung  $i < \text{elemente} - 1$  bzw.  $i \leq \text{elemente} - 2$  lauten muss, da ansonsten in der inneren if-Bedingung mit  $\text{arr}[i + 1]$  ein nicht erlaubter Elementzugriff erfolgt – die Elementzahl entspricht ja  $\text{elemente}$  und damit die Indexobergrenze  $\text{elemente} - 1$ .

Falls eine Auswertung der if-Bedingung true ergibt, wird die Funktion unverzüglich mit dem Rückgabewert 1 verlassen:

```
if (arr[i] == arr[i + 1])
    return 1;
```

Die Anweisung

```
return 0;
```

wird also nur dann ausgeführt, wenn in der for-Schleife bzw. in der inneren if-Bedingung keine Übereinstimmung festgestellt wurde:

```
#include <iostream>
using namespace std;
void sortiereArr(int *arr, int elemente);
int ArrMinIndex(int *arr, int ugrenze, int ogrenze);
int doppelte(int *arr, int elemente);
int doppelte(int *arr, int elemente)
// prüft ein Array auf doppelte Werte hin
// Rückgabewert ist 1, falls doppelte Werte vorhanden
{
    sortiereArr(arr, elemente);
    for (int i = 0; i < elemente - 1; i++)
    {
        if (arr[i] == arr[i + 1])
            return 1;
    }
}
```

```

    return 0;
}
void sortiereArr(int *arr, int elemente)
// sortiert ein Array aufsteigend
{
    int ablage;
    for(int i=0; i < elemente; i++)
    {
        ablage = arr[ArrMinIndex(arr, i, elemente - 1)];
        arr[ArrMinIndex(arr, i, elemente-1)] = arr[i];
        arr[i] = ablage;
    }
}
int ArrMinIndex(int *arr, int ugrenze, int ogrenze)
// gibt den kleinsten Wert eines Arrays
// im Bereich ugrenze bis ogrenze zurück
{
    int min_index = ugrenze;
    for(int i = ugrenze + 1; i <= ogrenze; i++)
    {
        if(arr[i] < arr[min_index])
        {
            min_index = i;
        }
    }
    return min_index;
}
int main(void)
{
    int meinArray[5] = {98, 15, 7, 15, 64};
    if (doppelte(meinArray, 5))
        cout << "Doppelte Werte im Array";
    else
        cout << "Keine doppelten Werte im Array";
    return 0;
}

```

Ausgabe:

```
Doppelte Werte im Array
```

Sie könnten nun Ihrem Projekt für die Funktionsdefinitionen von `ArrMinIndex()`, `sortiereArr()` und `doppelte()` eine neue Quelldatei hinzufügen, die Sie später beliebig erweitern und gleichfalls für Ihre zukünftigen Programme nutzen können. Gewöhnlich richtet sich der Name einer solchen Funktionsdatei nach dem Zweck der integrierten Funktionen, wie das am Beispiel der vordefinierten Bibliotheken *cmath*, *ctime*, *string* usw. der Fall ist. Da dieser hier nicht so ganz einheitlich ist – abgesehen davon, dass alle drei Funktio-



nen sich auf Arrays beziehen –, nennen wir die entsprechende Quelldatei hier einfach *function.cpp*. Die Funktionsprototypen schreiben Sie in eine separate Headerdatei *function.h*. Diese binden Sie in Ihre Quelldateien ein. Somit ergibt sich für die Dateien *function.cpp* und *function.h* folgendes Bild.

*function.h*:

```
void sortiereArr(int *arr, int elemente);
int ArrMinIndex(int *arr, int ugrenze, int ogrenze);
int doppelte(int *arr, int elemente);
```

*function.cpp*:

```
#include <iostream>
#include "function.h"
using namespace std;

int doppelte(int *arr, int elemente)
// prüft ein Array auf doppelte Werte hin
// Rückgabewert ist 1, falls doppelte Werte vorhanden
{
    sortiereArr(arr, elemente);
    for (int i = 0; i < elemente - 1; i++)
    {
        if (arr[i] == arr[i + 1])
            return 1;
    }
    return 0;
}

void sortiereArr(int *arr, int elemente)
// sortiert ein Array aufsteigend
{
    int ablage;
    for(int i=0; i < elemente; i++)
    {
        ablage = arr[ArrMinIndex(arr, i, elemente - 1)];
        arr[ArrMinIndex(arr, i, elemente-1)] = arr[i];
        arr[i] = ablage;
    }
}

int ArrMinIndex(int *arr, int ugrenze, int ogrenze)
// gibt den kleinsten Wert eines Arrays
// im Bereich ugrenze bis ogrenze zurück
{
    int min_index = ugrenze;
    for(int i = ugrenze + 1; i <= ogrenze; i++)
    {
```

```
    if(arr[i] < arr[min_index])
    {
        min_index = i;
    }
}
return min_index;
}
```

### CD-ROM

Die Dateien *function.cpp* und *function.h* finden Sie auf der Buch-CD im Verzeichnis *Beispiele/K22*, gepackt zu *Lottospiel.zip*.

Wir werden im folgenden Kapitel von diesen Dateien ausgehen. Falls Sie den Quellcode nicht aufteilen, schreiben Sie die Funktionsdefinitionen am besten samt Prototypen oberhalb von `main()`, wobei es aus bekannten Gründen sinnvoll ist, dass die Funktionsprototypen zuerst genannt werden.

# 22 Ein Lottospiel

Wir wollen nun ein Programm erstellen, das Lottozahlen auslost, die der Benutzer tippen kann. Wir werden dabei an einigen Stellen auf die im letzten Kapitel definierten Funktionen zurückgreifen.

## 22.1 Im Programm Lottozahlen auslosen

Ungeachtet dessen, ob Sie Lottospieler sind oder nicht, wird Ihnen sicherlich bekannt sein, dass beim Lotto sieben Zahlen (inklusive Zusatzzahl) ausgelost werden, die dann vom Spieler zu tippen sind. Es liegt nahe, die sieben Zahlen, die wir uns von der Funktion `rand()` zurückliefern lassen, in einem Integerarray festzuhalten. Dazu eignet sich eine `for`-Schleife – die Anzahl der Wiederholungen steht ja mit sieben fest. Zur Kontrolle geben wir die gelosten Zahlen anschließend aus.

### Referenz

Was zu tun ist, um im Programm Zufallszahlen zu erzeugen, können Sie in Kapitel 16, Abschnitt 16.5 »Zufallszahlen auslosen« nachlesen.

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

int main(void)
{
    int lottozahlen[7], i;
    //Auslosung
    srand(time(NULL));
    for(i = 0; i < 7; i++)
        lottozahlen[i] = (rand() % 49) + 1;
    //Ende Auslosung

    // Ausgabe der gelosten Zahlen
    for(i = 0; i < 7; i++)
        cout << lottozahlen[i] << ' ';
    // Ende Ausgabe
    return 0;
}
```

Ausgabe: 49 16 49 5 32 39 26

**Hinweis**

Der Übersicht halber kann es empfehlenswert sein, Anfang und Ende sinngemäß zusammengehöriger Programmabschnitte mit Kommentaren kenntlich zu machen:

```
// Auslosung
...
// Ende Auslosung
```

Wie gesagt, wird mit

```
srand(time(NULL));
```

der Zufallsgenerator initialisiert. Der Ausdruck

```
(rand() % 49) + 1
```

liefert eine Zufallszahl im Bereich von 1 bis 49 (jeweils inklusive).

Nun kann es vorkommen, dass Zahlen doppelt ausgelost werden, wie an der Ausgabe zu sehen. Um dies zu unterbinden, werden wir die Auslosung gegebenenfalls wiederholen. Der Einfachheit halber werden wir also nicht jede ausgeloste Zahl einzeln mit den zuvor ausgelosten Zahlen vergleichen. Vielmehr prüfen wir nach erfolgter Auslosung das Array `lottozahlen` auf doppelte Werte und zwar mit unserer Funktion `doppelte()`. Um diese verwenden zu können, ist es erforderlich, die Headerdatei `function.h` einzubinden.

**Hinweis**

Dies ist natürlich nicht notwendig, wenn Sie die Funktionen `ArrMinIndex()`, `sortiereArr()` und `doppelte()` mit `main()` zusammen in einer Datei definiert haben.

```
#include <iostream>
#include <ctime>
#include "function.h"
using namespace std;

int main(void)
{
    int lottozahlen[7], i;
    //Auslosung
    srand(time(NULL));
    do {
        for(i = 0; i < 7; i++)
            lottozahlen[i] = (rand() % 49) + 1;
    } while (doppelte(lottozahlen, 7));
```

```
#include <iostream>
#include <ctime>
#include "function.h"
using namespace std;

int main(void)
{
    int lottozahlen[7], i;
    //Auslosung
    srand(time(NULL));
    do {
        for(i = 0; i < 7; i++)
            lottozahlen[i] = (rand() % 49) + 1;
    } while (doppelte(lottozahlen, 7));
    //Ende Auslosung

    // Ausgabe der gelosten Zahlen
    for(i = 0; i < 7; i++)
        cout << lottozahlen[i] << ' ';
    // Ende Ausgabe
    return 0;
}
```

Ausgabe: 11 12 23 24 26 41 43

### Hinweis

Ein durchaus erwünschter Nebeneffekt der Funktion `doppelte()` ist, dass sie das Array aufsteigend sortiert.

Aus programmtechnischen Gründen soll sich die Zusatzzahl im letzten Element des Arrays befinden, also in `lottozahlen[6]`. Im Moment wäre dies aber zwangsläufig immer die größte Zahl. Um dies zu ändern, bestimmen wir im Zufallsverfahren ein Element des Arrays und vertauschen dessen Wert mit dem des letzten Elements. Wobei wir in Kauf nehmen, dass gegebenenfalls die letzte Zahl mit sich selbst vertauscht wird, was aber keine negativen Auswirkungen hat.

```
#include <iostream>
#include <ctime>
#include "function.h"
using namespace std;

int main(void)
{
    int lottozahlen[7], i, z, tmp;
    //Auslosung
```

```

srand(time(NULL));
do {
    for(i = 0; i < 7; i++)
        lottozahlen[i] = (rand() % 49) + 1;
} while (doppelte(lottozahlen, 7));
z = rand() % 7;
tmp = lottozahlen[6];
lottozahlen[6] = lottozahlen[z];
lottozahlen[z] = tmp;
//Ende Auslosung

// Ausgabe der gelosten Zahlen
sortiereArr(lottozahlen, 6);
for(i = 0; i < 6; i++)
    cout << lottozahlen[i] << ' ';
cout << "Zusatzzahl: " << lottozahlen[6];
// Ende Ausgabe
return 0;
}

```

Ausgabe: 6 23 28 31 37 46 Zusatzzahl: 16

Der Ausdruck

```
z = rand() % 7
```

liefert eine Zahl zwischen 0 und 6. Diese wird anschließend als Index für den folgenden Elementzugriff verwendet. Bei dem Codestück

```

tmp = lottozahlen[6];
lottozahlen[6] = lottozahlen[z];
lottozahlen[z] = tmp;

```

handelt es sich um den Ihnen bekannten Dreieckstausch (Kapitel 10, Abschnitt 10.5 »Dreieckstausch«).

Nach dem Tausch ist es erforderlich, die Zahlen im Array – bis auf die Zusatzzahl – erneut zu sortieren. Daher wird vor Ausgabe der Lottozahlen unsere Sortierfunktion `sortiereArr()` auf die ersten sechs Elemente des Arrays angewendet, was ja entsprechend der Implementation dieser Funktion möglich ist.

## CD-ROM

Alle Dateien, die Sie zum Kompilieren des Beispiels benötigen, finden Sie zusammen in *k22b.zip* im Ordner *Beispiele/K22* auf der Buch-CD. Bei den Dateien *function.cpp* und *function.h* handelt es sich um die gleichen wie in *Lottospiel.zip*.

## 22.2 Tippen

Um aus dem Ganzen ein richtiges Spiel zu machen, wollen wir dem Benutzer die Möglichkeit zum Tippen geben. Getippt werden müssen sechs Zahlen, die wir in dem Array `tipp` festhalten. Zur Umsetzung bietet sich wiederum eine `for`-Schleife an:

```
#include <iostream>
#include <ctime>
#include "function.h"
using namespace std;

int main(void)
{
    int lottozahlen[7], tipp[6], i, z, tmp;
    //Auslosung
    srand(time(NULL));
    do {
        for(i = 0; i < 7; i++)
            lottozahlen[i] = (rand() % 49) + 1;
    } while (doppelte(lottozahlen, 7));
    z = rand() % 7;
    tmp = lottozahlen[6];
    lottozahlen[6] = lottozahlen[z];
    lottozahlen[z] = tmp;
    //Ende Auslosung

    // Tipp
    cout << "Ihr Tipp: " << endl;
    for (i = 0; i < 6; i++)
        cin >> tipp[i];
    // Ende Tipp

    // Ausgabe der gelosten Zahlen
    sortiereArr(lottozahlen, 6);
    for(i = 0; i < 6; i++)
        cout << lottozahlen[i] << ' ';
    cout << "Zusatzzahl: " << lottozahlen[6];
    // Ende Ausgabe
    return 0;
}
```

### Hinweis

Hier soll vorausgesetzt werden, dass der Benutzer keine Zahlen doppelt tippt.

Nun gilt es, die Trefferquote zu ermitteln. Für Nicht-Lottospieler: Gewinne gibt es ab drei richtig getippten Zahlen mit den Gewinnrängen

- 3 Richtige
- 3 Richtige plus Zusatzzahl
- 4 Richtige
- 4 Richtige plus Zusatzzahl
- 5 Richtige
- 5 Richtige plus Zusatzzahl
- 6 Richtige

Die Zusatzzahl lassen wir zunächst außen vor, da diese ja erst ab drei richtig getippten Zahlen relevant ist. Wir müssen nun das Array `tipp` mit dem Array `lottozahlen` – abgesehen von der Zusatzzahl `lottozahlen[6]` – vergleichen und zwar jede Zahl des ersten Arrays mit jeder der ersten sechs Zahlen des zweiten Arrays. Mit anderen Worten: Für jede Zahl des einen Arrays muss das andere einmal vollständig durchlaufen werden. Zur Lösung dieser Aufgabe bietet sich die Konstruktion einer verschachtelten `for`-Schleife an:

```
#include <iostream>
#include <ctime>
#include "function.h"
using namespace std;

int main(void)
{
    int lottozahlen[7], tipp[6], i, k, z, tmp, treffer;
    //Auslosung
    srand(time(NULL));
    do {
        for(i = 0; i < 7; i++)
            lottozahlen[i] = (rand() % 49) + 1;
    } while (doppelte(lottozahlen, 7));
    z = rand() % 7;
    tmp = lottozahlen[6];
    lottozahlen[6] = lottozahlen[z];
    lottozahlen[z] = tmp;
    //Ende Auslosung

    // Tipp
    cout << "Ihr Tipp: " << endl;
    for (i = 0; i < 6; i++)
        cin >> tipp[i];
    // Ende Tipp
```



```

// Trefferzahl ermitteln
treffer = 0;
for (i = 0; i < 6; i++) {
    for (k = 0; k < 6; k++) {
        if (tipp[i] == lottozahlen[k])
            treffer++;
    }
}
// Ende Trefferzahl ermitteln

// Ausgabe der gelosten Zahlen
sortiereArr(lottozahlen, 6);
for(i = 0; i < 6; i++)
    cout << lottozahlen[i] << ' ';
cout << "Zusatzzahl: " << lottozahlen[6];
// Ende Ausgabe
return 0;
}

```

Denken Sie daran, die Laufvariable *k* der inneren *for*-Schleife zu deklarieren. Die Anzahl der bis dato richtig getippten Zahlen wird in der Variablen *treffer* festgehalten. Diese muss logischerweise bei Eintritt in die Schleifenkonstruktion den Wert 0 besitzen. Um dies deutlich zu machen – und damit auch die Nachvollziehbarkeit des Codes zu verbessern –, haben wir die Zuweisung

```
treffer = 0;
```

unmittelbar davor gesetzt. Natürlich hätten wir die Variable hier auch bei ihrer Vereinbarung mit diesem Wert initialisieren können. Die innere *if*-Bedingung wertet zu *true* aus, falls der jeweils betrachtete Wert des ersten Arrays mit dem des zweiten übereinstimmt, was bedeutet, dass es sich um eine richtig getippte Zahl handelt. Für diesen Fall wird die Variable *treffer* inkrementiert.

## 22.3 Gewinnanzeige

Nun ist die Ziehung der Lottozahlen erfolgt, der Spieler hat getippt und die Anzahl der Treffer ist auch bekannt, von der Zusatzzahl einmal abgesehen. Was noch fehlt ist eine entsprechende Gewinnanzeige. Allerdings muss für den Fall, dass der Spieler drei oder mehr Richtige hat, noch geprüft werden, ob er die Zusatzzahl ebenfalls richtig getippt hat. Hier das entsprechende Codestück:

```

if (treffer >= 3)
{
    cout << "Sie haben " << treffer << " Richtige";
    // Zusatzzahl richtig?
}

```

```

    for (i = 0; i < 6; i++)
    {
        if (tipp[i] == lottozahlen[6])
        {
            cout << " mit Zusatzzahl";
            break;
        }
    }
    // Ende Zusatzzahl richtig?
}

if (treffer == 0)
    cout << "Leider keine Richtigen";
if (treffer == 1)
    cout << "Leider nur eine Zahl richtig";
if (treffer == 2)
    cout << "Leider nur zwei Richtige";

```

Falls der Spieler drei oder mehr Richtige hat, wird in der for-Schleife bzw. in der inneren if-Bedingung jede getippte Zahl mit der Zusatzzahl `lottozahlen[6]` verglichen. Bei Übereinstimmung kann die for-Schleife sofort mit `break`; beendet werden.

Hier das vollständige Programm:

```

#include <iostream>
#include <ctime>
#include "function.h"
using namespace std;

int main(void)
{
    int lottozahlen[7], tipp[6], i, k, z, tmp, treffer;

    //Auslosung
    srand(time(NULL));
    do {
        for(i = 0; i < 7; i++)
            lottozahlen[i] = (rand() % 49) + 1;
    } while (doppelte(lottozahlen, 7));
    z = rand() % 7;
    tmp = lottozahlen[6];
    lottozahlen[6] = lottozahlen[z];
    lottozahlen[z] = tmp;
    //Ende Auslosung

```

```
// Tipp
cout << "Ihr Tipp: " << endl;
for (i = 0; i < 6; i++)
    cin >> tipp[i];
// Ende Tipp

// Trefferzahl ermitteln
treffer = 0;
for (i = 0; i < 6; i++) {
    for (k = 0; k < 6; k++) {
        if (tipp[i] == lottozahlen[k])
            treffer++;
    }
}
// Ende Trefferzahl ermitteln

// Ausgabe der gelosten Zahlen
cout << endl << "Lottozahlen: " << endl;
sortiereArr(lottozahlen, 6);
for(i = 0; i < 6; i++)
    cout << lottozahlen[i] << ' ';
cout << "Zusatzzahl: " << lottozahlen[6];
cout << endl;
// Ende Ausgabe

// Gewinnanzeige
cout << "Ihre getippten Zahlen:" << endl;
sortiereArr(tipp, 6);
for (i = 0; i < 6; i++)
    cout << tipp[i] << ' ';
cout << endl << endl;
if (treffer >= 3)
{
    cout << "Sie haben " << treffer << " Richtige";
    // Zusatzzahl richtig?
    for (i = 0; i < 6; i++)
    {
        if (tipp[i] == lottozahlen[6])
        {
            cout << " mit Zusatzzahl";
            break;
        }
    }
    // Ende Zusatzzahl richtig?
}
if (treffer == 0)
    cout << "Leider keine Richtigen";
```

```

    if (treffer == 1)
        cout << "Leider nur eine Zahl richtig";
    if (treffer == 2)
        cout << "Leider nur zwei Richtige";
    // Ende Gewinnanzeige
    cout << endl;
    return 0;
} // Ende main()

```

## Tipp

Um sich im Quelltext besser zurechtzufinden, empfiehlt es sich mitunter, hinter die schließende geschweifte Klammer einer Kontrollstruktur oder einer Funktion einen entsprechenden Kommentar zu setzen, z.B.

```

...
} // Ende for

oder

...
} // Ende main()

```

Für den Fall, dass der Benutzer beim Tippen die Zahlen nicht in aufsteigender Reihenfolge angibt, wird das Array `tipp` vor der Ausgabe mit der Anweisung

```
sortiereArr(tipp, 6);
```

sortiert. Das Ergebnis eines Spieles sehen Sie in Abbildung 22.1.

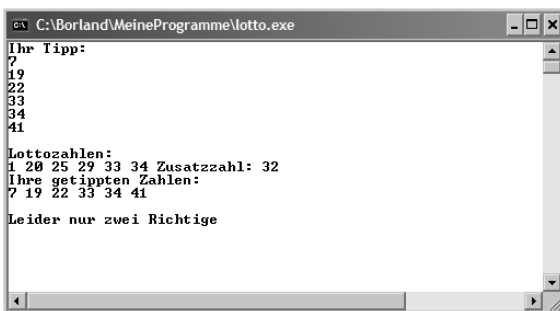


Abbildung 22.1: Bescheidenes Ergebnis eines Lottotipps

## CD-ROM

Das Programm mit der ausführbaren Datei *lotto.exe* sowie den Quelldateien *lotto.cpp* und *function.cpp* plus Headerdatei *function.h* finden Sie auf der Buch-CD im Verzeichnis *Beispiele/K22* gepackt als *Lottospiel.zip*.

# Teil III

## Einführung in die objekt-orientierte Programmierung

Theoretisch ließe sich jedes Programm auch ohne die Mittel der objektorientierten Programmierung und auch der funktionalen Programmierung schreiben. Praktisch sind dem bei sehr großen Programmen jedoch Grenzen gesetzt. In dieser Hinsicht ist bereits die Aufteilung des Codes in Funktionen ein wesentlicher Schritt hin zur besseren Strukturierung mit allen damit verbundenen Vorteilen wie bessere Wartbarkeit, Wiederverwendbarkeit, Reduzierung der Fehleranfälligkeit usw. Während in der funktionalen Programmierung Daten und Funktionen jedoch systematisch voneinander getrennt sind, erlaubt es das Konzept der objektorientierten Programmierung, diese miteinander zu verknüpfen und damit als Einheit zu behandeln. Damit erhebt die objektorientierte Programmierung – kurz OOP genannt – den Anspruch, Dinge einer gedachten oder realen Wirklichkeit im Programm abbilden zu können.

Das nachfolgende Kapitel führt Sie über den Strukturbegriff an das Wesen der Klassen heran. Klassen mit ihren Objekten sind das A und O der objektorientierten Programmierung. In den weiteren Kapiteln werden die wesentlichen Aspekte der OOP vorgestellt, wie z.B. Kapselung, Konstruktoren, statische Klassenelemente etc. Einige weiterführende Konzepte wie Polymorphie, Templates oder Friend-Deklarationen konnten im Rahmen dieses Buches nicht berücksichtigt werden. Zur Vertiefung und Festigung Ihrer OOP-Kenntnisse sei Ihnen daher die entsprechende Fachliteratur empfohlen, beispielsweise das Buch »Objektorientiert in C++« von Dirk Louis (entwickler.press).



# 23 Strukturen

Strukturen, so wie sie in der Programmiersprache C verwendet werden, sind in gewissem Sinne ein erster Schritt hin zur objektorientierten Programmierung. Daher liegt es nahe, eine Einführung in die objektorientierte Programmierung mit diesen zu beginnen.

Aus didaktischen Gründen werden wir Strukturen hier also zunächst in C-Manier behandeln. Dies ist insofern nicht weiter problematisch, als es sich bei den C++-Strukturen um eine reine Erweiterung derer von C handelt. Tatsächlich ist der Unterschied zwischen Klassen und Strukturen in C++ nur sehr gering. Somit gilt alles, was Sie hier über Strukturen erfahren, uneingeschränkt auch für Klassen.

## 23.1 Strukturen definieren

Eine wesentliche Eigenschaft von benutzerdefinierten Datentypen haben Sie am Beispiel von Arrays bereits kennen gelernt: Sie setzen sich mehr oder weniger aus einfachen Datentypen zusammen (auf das »mehr oder weniger« werden wir gleich noch zu sprechen kommen).

### Hinweis

Wir gehen hier davon aus, dass ein Array ein benutzerdefinierter Datentyp ist. Dass Ihr Compiler dies ebenso sieht, können Sie daran erkennen, dass er im Zuge von Warn- bzw. Fehlermeldungen z.B. den Datentyp eines Integerarrays aus neun Elementen mit `int[9]` angibt.

Anders als das jedoch beim Array der Fall ist, dürfen die Elemente einer Struktur auch unterschiedlichen Datentyps sein.

Die Definition einer Struktur wird mit dem Schlüsselwort `struct` eingeleitet, gefolgt von einem Bezeichner für die Struktur und einem zugehörigen Block, der mit einem Semikolon abzuschließen ist. In dem Block sind die Elemente der Struktur anzugeben:

```
struct Adresse
{
    string strasse;
    int hausnr;
    long plz;
    string ort;
};
```

Wie gesagt, handelt es sich hier um die Definition eines neuen Datentyps. Dessen Bezeichner ist Adresse. Dieser kann nun im Code genauso verwendet werden wie z.B. die vordefinierten Typbezeichner `int` oder `char`. Um also eine Variable vom Datentyp Adresse zu deklarieren, schreiben Sie:

```
Adresse meineAdresse;
```

Beachten Sie, dass die alleinige Definition von Adresse noch nichts bewirkt, außer dass nunmehr ein neuer Datentyp zur Verfügung steht. Nehmen wir zum Vergleich den Datentyp `double`. Die Tatsache, dass dieser existiert, hat noch keine Auswirkung auf Ihre Programme. Dies ändert sich erst, wenn Sie im Code Variablen dieses Typs vereinbaren. Von daher besitzt ein benutzerdefinierter Typ die gleiche Qualität wie ein eingebauter Typ. Denken Sie daran, dass es für die Verwendung einer Funktion ebenfalls unerheblich ist, ob diese vom Compilersystem mitgeliefert oder vom Programmierer selbst definiert ist.

Folglich ist erst mit der obigen Anweisung, in der eine Variable `meineAdresse` vom Typ Adresse vereinbart wird, Reservierung von Speicher verbunden. Wobei für die einzelnen Elemente von `meineAdresse` ein zusammenhängender Speicherbereich reserviert wird. Folglich richtet sich die Größe einer Variablen vom Typ einer Struktur nach den Datentypen der einzelnen Komponenten.

Aus dem genannten Grund ist es auch nicht möglich, bei der Definition einer Struktur einzelne Elemente zu initialisieren, da diese zum Zeitpunkt der Definition ja noch nicht existieren, also Variablen dieser Struktur noch nicht angelegt sind:

```
struct Adresse
{
    string strasse;
    int hausnr;
    long plz = 88420; // FEHLER
    string ort;
};
```

Wir hatten eingangs darauf aufmerksam gemacht, dass sich jeder benutzerdefinierte Datentyp mehr oder weniger auf elementare Datentypen zurückführen lässt. Die Einschränkung »mehr oder weniger« bezieht sich auf die Tatsache, dass sich ein benutzerdefinierter Datentyp wiederum aus weiteren komplexen Datentypen zusammensetzen kann, was aber an dem genannten Sachverhalt dennoch nichts ändert, da die Kette irgendwann schließlich bei elementaren Datentypen endet.

So kann es sich bei den Komponenten einer Struktur nicht nur um einfache, sondern auch um komplexe Datentypen wie z.B. Arrays oder andere Strukturen handeln. Beispielsweise ließe sich eine Struktur `Schueler` definieren, die als Element die oben definierte Struktur `Adresse` und ein Array `noten` besitzt:



```
struct Schueler
{
    string vorname;
    string nachname;
    Adresse adr;
    int noten[18];
};
```

Wenn man nun mit der Anweisung

```
Schueler Schlau;
```

eine Variable `Schlau` vereinbart, dann besitzt diese zwei `string`-Variablen, ein Integerarray mit 18 Elementen und eine Variable vom Typ `Adresse`, das heißt weitere zwei `string`-Variablen sowie je eine Variable vom Datentyp `int` bzw. `long`. Alle diese Elemente liegen in einem zusammenhängenden Speicherbereich, sodass auf diese über den Bezeichner `Schlau` zugegriffen werden kann. Wie das geht, erfahren Sie im nächsten Abschnitt.

Natürlich können Sie im Code eine zweite Variable – und natürlich beliebig viele weitere – des Typs `Schueler` vereinbaren:

```
Schueler Dumm;
```

Für diese Variable wird nun ebenfalls ein zusammenhängender Speicherbereich reserviert, in dem die entsprechenden Elemente Platz haben bzw. deren zukünftige Werte – bis jetzt wurden ihnen ja noch keine zugewiesen. Wie jeder Datentyp ist eine Strukturdefinition also gewissermaßen eine Schablone für zukünftige Variablen, sozusagen der Bauplan, nach dem Variablen angelegt werden.

Nun ist es so, dass eine Struktur in C++ tatsächlich bereits eine Klasse ist, also ein Bestandteil der objektorientierten Programmierung. Lassen Sie uns in diesem Zusammenhang auf einen weiteren Sachverhalt aufmerksam machen: Wenn Sie die beiden Variablen `Schlau` und `Dumm` des Typs `Schueler` als »Schüler Schlau« bzw. »Schüler Dumm« bezeichnen, wird offensichtlich, dass damit Objekte der Wirklichkeit gemeint sind. In diesem Sinne bedeutet eine Aussage wie

»Der Schüler Dumm hat als dritte Note eine Sechs.«,

dass das dritte Element des Arrays `noten` der Variablen `Dumm` den Wert 6 besitzt.

In gleicher Weise könnten Sie sagen, dass der Schüler Schlau den Vornamen Daniel hat, was bedeuten würde, dass der Wert des Elements `vorname` der Variablen `Schlau` den Wert "Daniel" besitzt.

In Anbetracht dessen sollte es Sie nicht weiter verwundern, dass man in der Welt der objektorientierten Programmierung von Objekten und deren Eigenschaften bzw. Attributen spricht. Mit anderen Worten: Variablen vom Typ einer Struktur oder einer Klasse werden gewöhnlich als Objekte, deren Elemente als Eigenschaften oder Attribute bezeichnet.

**Hinweis**

Wobei der Begriff »Element« und vor allem bezüglich Strukturen auch die Bezeichnung »Variable« nach wie vor gebraucht wird. Letzteres rührt daher, dass Strukturen, wie sie in der Programmiersprache C – und in diesem Kapitel – verwendet werden, praktisch die nicht objektorientierten Vorläufer von Klassen sind.

Wir sagen also nicht mehr, »das Element vorname der Variablen Dumm«, sondern »die Eigenschaft vorname des Objekts Dumm« oder »das Attribut vorname des Objekts Dumm«, allenfalls »das Element vorname des Objekts Dumm« oder schlicht »der Vorname des Schülers Dumm«.

**Hinweis**

Dementsprechend lassen wir die Bezeichner von Objekten sowie die von selbst definierten Strukturen und Klassen mit einem Großbuchstaben beginnen. Es sei jedoch darauf hingewiesen, dass dies unter C++-Programmierern nicht einheitlich gehandhabt wird.

Obwohl es erlaubt ist, Strukturen innerhalb von Funktionen zu definieren, sollten Sie dies möglichst global tun, also außerhalb von Funktionen zu Beginn des Quellcodes:

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

struct Schueler
{
    string vorname;
    string nachname;
    int dtnoten[6];
    int englnoten[6];
    int mathnoten[6];
};

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

## Hinweis

Natürlich muss Ihrem Compiler die Definition einer Struktur bekannt sein, bevor Sie Objekte davon erzeugen können. Wenn eine Struktur lokal, also innerhalb einer Funktion, definiert ist, dann ist sie allein in dieser Funktion bekannt. Das heißt, Sie können dann in anderen Funktionen keine Variablen (Objekte) vom Typ dieser Struktur vereinbaren.

Um für die Struktur `Schueler` bzw. für zukünftige Objekte dieser Struktur Adressdaten festzulegen, könnte man in der Definition Elemente für die Straße, den Wohnort usw. angeben. Wir greifen hier aber auf die oben angegebene Strukturdefinition von `Adresse` zurück, die die gewünschten Elemente enthält. Das heißt, wir integrieren in der Struktur `Schueler` ein Element des Typs `Adresse`. Folglich muss der Typ `Adresse` in `Schueler` bekannt sein. Daher wird `Adresse` oberhalb von `Schueler` definiert:

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

struct Adresse
{
    string strasse;
    int hausnr;
    long plz;
    string ort;
};

struct Schueler
{
    string vorname;
    string nachname;
    int dtnoten[6];
    int englnoten[6];
    int mathnoten[6];
    Adresse adr;
};

int main(void)
{
    Schueler Schlau;
    Schueler Dumm;
    return 0;
}
```

**Achtung**

Jede Strukturdefinition muss mit einem Semikolon abgeschlossen werden.

Mit den Anweisungen

```
Schueler Schlaue;
```

und

```
Schueler Dumm;
```

werden nun nach dem Bauplan `Schueler` zwei Objekte erzeugt.

**Hinweis**

Dies könnte natürlich auch in einer Anweisung geschehen:

```
Schueler Schlaue, Dumm;
```

Jedes dieser Objekte besitzt danach die Elemente `vorname`, `nachname`, `noten` und außerdem `adr` und damit `strasse`, `hausnr`, `plz` und `ort`. Diese Elemente »gehören« praktisch den Objekten. Es sind gewissermaßen deren ganz spezielle Eigenschaften, die sie von Objekten anderer Typen und gegebenenfalls auch von Objekten des gleichen Typs unterscheiden. Letzteres, da die einzelnen Elemente eines Objekts im weiteren Verlauf ja in der Regel charakteristische Werte erhalten.

Beispielsweise, wenn man der Eigenschaft `nachname` des Objekts `Schlaue` den Wert "Schlaumeier" oder der entsprechenden Eigenschaft des Objekts `Dumm` den Wert "Dummkopf" zuweist.

## 23.2 Auf die Komponenten von Strukturen zugreifen

Um auf ein Element eines Objekts zuzugreifen, ist der Name des Objekts und, getrennt durch einen Punkt, der Name des Elements anzugeben. Demnach lassen sich z.B. die besagten Zuweisungen wie folgt durchführen:

```
Schlaue.nachname = "Schlaumeier";  
Dumm.nachname = "Dummkopf";
```

Es sei darauf hingewiesen, dass alles seine Gültigkeit behält, was Sie über Variablen, Arrays usw. wissen. Nur dass der Zugriff auf die Elemente eines Objekts eben über den Namen dieses Objekts plus eines Punktes zu erfolgen hat. Mit der Angabe des Objekt-

namens verweisen Sie auf einen Speicherbereich und mit Angabe des Elementnamens spezifizieren Sie die Stelle in diesem Bereich, auf die Sie zugreifen möchten:

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

struct Adresse
{
    string strasse;
    int hausnr;
    long plz;
    string ort;
};

struct Schueler
{
    string vorname;
    string nachname;
    int dtnoten[6];
    int englnoten[6];
    int mathnoten[6];
    Adresse adr;
};

int main(void)
{
    Schueler Schlaueier;
    Schueler Dummkopf;
    Schlaueier.nachname = "Schlaumeier";
    Dummkopf.nachname = "Dummkopf";
    cout << Schlaueier.nachname << " und " << Dummkopf.nachname;
    return 0;
}
```

Ausgabe: Schlaumeier und Dummkopf

### Hinweis

Der Punkt repräsentiert somit den Zugriff auf ein Struktur- oder Klassenelement. Es handelt sich also um einen – binären – Operator. Sein linker Operand ist der Bezeichner eines Objekts, sein rechter der eines Struktur- oder Klassenelements. Statt Elementzugriffs- oder Komponentenzugriffsoperator wird er im Allgemeinen einfach »Punktoperator« genannt.

Nun kann es vorkommen, dass ein Objekt als Attribut ein anderes Objekt besitzt, was für Objekte des Typs `Schueler` bezüglich der Eigenschaft `adr` zutrifft. Dann ist beim Zugriff

auf dieses in gleicher Weise zu verfahren. Um etwa die Attribute für Postleitzahl und Wohnort bezüglich der Adresse des Schülers Schlau mit Werten zu versehen, können Sie demnach schreiben:

```
Schlau.adr.plz = 10711;
Schlau.adr.ort = "Berlin";
```

Daran ist erkennbar, dass der Punktoperator linksassoziativ sein muss. Das heißt, der Zugriff erfolgt über das Objekt Schlau auf das innere Objekt Schlau.adr und über dieses auf dessen Element ort bzw. plz, also in der Reihenfolge, die der Klammerung

```
(Schlau.adr).plz = 10711;
```

bzw.

```
(Schlau.adr).ort = "Berlin";
```

entspricht.

## Referenz

Zur Assoziativität von Operatoren siehe Kapitel 11, Abschnitt 11.5 »Priorität von Operatoren«.

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

struct Adresse
{
    string strasse;
    int hausnr;
    long plz;
    string ort;
};

struct Schueler
{
    string vorname;
    string nachname;
    int dtnoten[6];
    int englnoten[6];
    int mathnoten[6];
    Adresse adr;
};
```

```
int main(void)
{
    Schueler Schlaue, Dumm;
    Schlaue.nachname = "Schlaumeier";
    Dumm.nachname = "Dummkopf";
    Schlaue.adr.plz = 10711;
    Schlaue.adr.ort = "Berlin";
    Dumm.adr.plz = 99999;
    Dumm.adr.ort = "Nirgendwo";
    cout << Schlaue.nachname << " wohnt in "
          << Schlaue.adr.plz << ' ' << Schlaue.adr.ort;
    cout << endl;
    cout << Dumm.nachname << " wohnt in " << Dumm.adr.plz
          << ' ' << Dumm.adr.ort;
    return 0;
}
```

Ausgabe:

```
Schlaumeier wohnt in 10711 Berlin
Dummkopf wohnt in 99999 Nirgendwo
```

Falls es sich bei den Komponenten einer Struktur um Arrays handelt, sind beim Zugriff auf deren Elemente diese wie gewohnt zu indizieren:

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

struct Adresse
{
    string strasse;
    int hausnr;
    long plz;
    string ort;
};

struct Schueler
{
    string vorname;
    string nachname;
    int dtnoten[6];
    int englnoten[6];
    int mathnoten[6];
    Adresse adr;
};
```

```

int main(void)
{
    Schueler Schlau, Dumm;
    int i;
    Schlau.nachname = "Schlaumeier";
    Dumm.nachname = "Dummkopf";
    for (i = 0; i < 6; i++)
        Schlau.englnoten[i] = 1;
    for (i = 0; i < 6; i++)
        Dumm.mathnoten[i] = 6;
    cout << "Englischnoten von "
          << Schlau.nachname << ": ";
    for (i = 0; i < 6; i++)
        cout << Schlau.englnoten[i] << ' ';
    cout << endl;
    cout << "Mathematiknoten von "
          << Dumm.nachname << ": ";
    for (i = 0; i < 6; i++)
        cout << Dumm.mathnoten[i] << ' ';
    return 0;
}

```

Ausgabe:

```

Englischnoten von Schlaumeier: 1 1 1 1 1 1
Mathematiknoten von Dummkopf: 6 6 6 6 6 6

```

Ein Zugriff erfolgt also nie auf das ganze Objekt, sondern grundsätzlich auf ein Element des Objekts. Allerdings ist die direkte Zuweisung

```
Objekt_A = Objekt_B
```

möglich. In diesem Fall werden alle Werte von *Objekt\_B* elementweise nach *Objekt\_A* kopiert:

```

#include <iostream>
using namespace std;

struct Punkt
{
    int x;
    int y;
};

```



```
int main(void)
{
    Punkt Eins, Zwei;
    Eins.x = 3;
    Eins.y = 5;
    Zwei = Eins;
    cout << "Der zweite Punkt hat die Koordinaten "
         << Zwei.x << " und " << Zwei.y << '.';
    return 0;
}
```

Ausgabe:

```
Der zweite Punkt hat die Koordinaten 3 und 5.
```

Wie gesagt, kann der Bezeichner eines benutzerdefinierten Datentyps wie der eines elementaren Datentyps eingesetzt werden. Folglich ist es auch möglich, ein Array vom Typ einer Struktur zu vereinbaren. Hier ist zu beachten, dass für den Elementzugriff zunächst eine Indizierung des speziellen Arrayelements, also des gewünschten Objekts, erforderlich ist:

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

struct Person
{
    string vorname;
    string nachname;
    long plz;
    string ort;
};

int main(void)
{
    Person teilnehmer[3];
    int gew;
    teilnehmer[0].vorname = "Daisy";
    teilnehmer[0].nachname = "Duck";
    teilnehmer[0].plz = 87773;
    teilnehmer[0].ort = "Entenhausen";
    teilnehmer[1].vorname = "Gustav";
    teilnehmer[1].nachname = "Gans";
    teilnehmer[1].plz = 87775;
    teilnehmer[1].ort = "Entenhausen";
```

```

    teilnehmer[2].vorname = "Donald";
    teilnehmer[2].nachname = "Duck";
    teilnehmer[2].plz = 87773;
    teilnehmer[2].ort = "Entenhausen";
    srand(time(NULL));
    gew = rand() % 3;
    cout << "Der Gewinner ist " << teilnehmer[gew].vorname
          << ' ' << teilnehmer[gew].nachname << endl;
    cout << "in " << teilnehmer[gew].plz
          << ' ' << teilnehmer[gew].ort << endl;
    return 0;
}

```

Ausgabe:

```

Der Gewinner ist Gustav Gans
in 87775 Entenhausen

```

## 23.3 Ausblick

Wie mehrfach angedeutet, sind Strukturen in C++ bereits Klassen. So könnte man die Struktur Person des letzten Beispiels

```

struct Person
{
    string vorname;
    string nachname;
    long plz;
    string ort;
};

```

ebenso unter Verwendung des Schlüsselwortes `class` definieren:

```

class Person
{
public:
    string vorname;
    string nachname;
    long plz;
    string ort;
};

```

Beide Definitionen sind einander völlig gleichwertig. Der Unterschied einer Struktur – oder anders ausgedrückt: einer mit `struct` eingeleiteten Klasse – gegenüber einer mit dem Schlüsselwort `class` definierten Klasse liegt allein darin, dass die Elemente ersterer

per Default `public`, die letzterer per Default `private` sind. Daher ist in der letzten Definition von `Person` der Spezifizierer `public` angegeben.

### **Hinweis**

Natürlich dürfen nicht zwei Klassendefinitionen von `Person` gemeinsam im Code stehen, es sei denn, diese befinden sich in verschiedenen Namensbereichen (zu Namensbereichen siehe Kapitel 24, Abschnitt 24.4 »Was es mit Namensräumen auf sich hat«). Dabei spielt es keine Rolle, ob diese mit `struct` oder mit `class` definiert sind.

### **Referenz**

Die Bedeutung der Schlüsselwörter `public` und `private` erfahren Sie in Kapitel 24, Abschnitt 24.2 »Zugriffsspezifizierer«.

Wir werden im Weiteren Klassendefinitionen stets mit dem Schlüsselwort `class` einleiten, sodass das Grundgerüst einer Klasse fürs Erste wie folgt aussieht:

```
class Klassenname
{
public:
    ...
};
```



# 24 Klassen und Objekte

Objekte der realen Welt besitzen nicht nur äußere Eigenschaften wie Größe, Farbe usw. Sie zeichnen sich auch durch ihr Verhalten bzw. ihre Fähigkeiten aus. Beispielsweise kann ein Vogel fliegen, ein Auto fahren. Beide Objekte ließen sich bereits anhand dieser Fähigkeiten ausreichend voneinander unterscheiden.

Zentrales Thema dieses Kapitels ist die Zuordnung von Daten und Funktionalität. Wie im letzten Kapitel bereits angedeutet, ist es für das Verständnis der objektorientierten Programmierung wesentlich, zwischen Datentypen und Datenobjekten unterscheiden zu können. Darum soll zunächst auf diesen Aspekt noch näher eingegangen werden.

## 24.1 Methoden

Ein Datenobjekt bezeichnet einen Platz im Arbeitsspeicher, an dem Werte abgelegt werden können. Wobei der Begriff »Datenobjekt« durchaus im allgemeinen Sinne zu verstehen ist. Das heißt, es kann sich dabei um ein Objekt einer Struktur oder Klasse, um ein Array, aber auch um eine einfache Variable handeln.

### Hinweis

Soweit im Weiteren von Klassen bzw. deren Objekten die Rede ist, sind damit natürlich auch Strukturen bzw. deren Objekte gemeint.

Ein Datentyp ist sozusagen der Bauplan – die Schablone – für zukünftige Datenobjekte. Zukünftig, da die bloße Existenz eines Datentyps ja noch keine Wirkungen hat. Auch diese Aussage – ebenso wie die folgenden – gilt grundsätzlich für alle Datentypen, ungeachtet dessen, ob es sich um elementare oder benutzerdefinierte handelt.

Nun legt die Zugehörigkeit eines Datenobjekts zu einem bestimmten Datentyp zum einen fest, wie viel Platz Datenobjekte im Arbeitsspeicher beanspruchen bzw. welche Art von Werten dort abgelegt werden können. Zum anderen – und dies ist für das Verständnis der OOP entscheidend – bestimmt ein Datentyp, welche Operationen auf Daten dieses Typs ausgeführt werden können. Mit anderen Worten: Ein Datentyp legt fest, wie sich zukünftige Datenobjekte verhalten. So lassen sich Zeichenketten miteinander verknüpfen, man kann aber mit ihnen keine arithmetischen Operationen durchführen.

Wie gesagt, ist die Definition einer Klasse die Beschreibung eines neuen Datentyps. Folgerichtig kann eine Klasse neben Datenelementen auch Funktionen enthalten. Diese werden im Zusammenhang mit einer Klasse Elementfunktionen oder Methoden genannt.

**Hinweis**

Dies sowie alles, was im Weiteren über Klassen gesagt wird, gilt natürlich in gleicher Weise auch für Strukturen.

Nehmen wir als Beispiel eine Klasse Rechteck, die Datenelemente für Länge und Breite sowie eine Methode enthält, die die Fläche eines Rechtecks als Rückgabewert liefert:

```
class Rechteck
{
public:
    int Laenge;
    int Breite;

    int berechneFl()
    {
        return Laenge * Breite;
    }
};
```

Somit besitzt jedes Rechteck – jedes Objekt der Klasse Rechteck – die Eigenschaften Laenge und Breite sowie eine Methode berechneFl().

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

class Rechteck
{
public:
    int Laenge;
    int Breite;

    int berechneFl()
    {
        return Laenge * Breite;
    }
};

int main(void)
{
    Rechteck A, B;
    A.Laenge = 7;
    A.Breite = 11;
    B.Laenge = 15;
    B.Breite = 4;
```

```
cout << "Fl\x84" << "che von A: "  
    << A.berechneFl() << endl;  
cout << "Fl\x84" << "che von B: "  
    << B.berechneFl() << endl;  
return 0;  
}
```

Ausgabe:

```
Fläche von A: 77  
Fläche von B: 60
```

Beachten Sie, dass Methoden immer auf dem Objekt operieren, für das sie aufgerufen werden. Der Aufruf einer Methode erfolgt dementsprechend analog zu dem Zugriff auf ein Datenelement, das heißt über den Objektnamen: *objekt.methode()*.

Daraus folgt: Jede Methode greift auf die Datenelemente des Objekts zurück, für das sie aufgerufen wird. So »weiß« die Methode `berechneFl()`, dass sie für einen Aufruf

```
A.berechneFl()
```

in ihrer Anweisung

```
return Laenge * Breite;
```

auf die entsprechenden Werte des Objekts A zurückgreifen muss, im Beispiel also 7 für Laenge und 11 für Breite.

### Hinweis

Methoden können auf die Datenelemente ihrer Klasse direkt zugreifen. Es ist nicht nötig, Datenelemente, die die Methode benötigt, als Parameter zu definieren.

Falls eine Methode jedoch Informationen über ein weiteres Objekt derselben Klasse benötigt, müssen ihr diese beim Aufruf übergeben werden. So z.B. wenn man der Klasse Rechteck eine Methode hinzufügt, die die Werte eines anderen Rechtecks zum aktuellen addiert:

```
#include <iostream>  
using namespace std;  
  
class Rechteck  
{  
public:  
    int Laenge;  
    int Breite;
```

```
int berechneFl()
{
    return Laenge * Breite;
}

void addiereRe(Recteck einAnderes)
{
    Laenge += einAnderes.Laenge;
    Breite += einAnderes.Breite;
}

};

int main(void)
{
    Recteck A, B;
    A.Laenge = 7;
    A.Breite = 11;
    B.Laenge = 15;
    B.Breite = 4;
    cout << "Fläche von A: "
         << A.berechneFl() << endl;
    cout << "Fläche von B: "
         << B.berechneFl() << endl;
    A.addiereRe(B);
    cout << "Fläche von A nach Addition von B: "
         << A.berechneFl() << endl;
    return 0;
}
```

Ausgabe:

```
Fläche von A: 77
Fläche von B: 60
Fläche von A nach Addition von B: 330
```

### Hinweis

Wobei wir hier die Addition zweier Rechtecke als die Addition ihrer Längen und Breiten definiert haben.

In der Methodendefinition von `addiereRe()` können die Eigenschaften `Laenge` und `Breite` nach wie vor ohne weitere Spezifizierung verwendet werden. Die Eigenschaften des Objekts, für das die Methode aufgerufen wird, sind dieser ja bekannt:



```
void addiereRe(Rchteck einAnderes)
{
    Laenge += einAnderes.Laenge;
    Breite += einAnderes.Breite;
}
```

Um die zu addierenden Werte für Länge und Breite eines weiteren Rechtecks zu erhalten, ist es ausreichend, ein zweites Rechteck zu übergeben. Damit werden ja dessen Eigenschaften der Methode bekannt gemacht. Datentyp des Parameters ist also Rechteck. Beachten Sie, dass die Methode keinen Rückgabewert benötigt. Die Zuweisung neuer Werte für die Eigenschaften `Laenge` und `Breite` des aktuellen Objekts spielt sich gewissermaßen klassenintern ab.

Es bietet sich an, die Klasse Rechteck mit einer zusätzlichen Methode zu versehen, die es erlaubt, die Werte für Länge und Breite individuell neu zu setzen:

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

class Rechteck
{
public:
    int Laenge;
    int Breite;

    int berechneFl()
    {
        return Laenge * Breite;
    }

    void addiereRe(Rchteck einAnderes)
    {
        Laenge += einAnderes.Laenge;
        Breite += einAnderes.Breite;
    }

    void setLB(int l, int b)
    {
        Laenge = l;
        Breite = b;
    }
};

int main(void)
{
    Rechteck A, B;
    A.setLB(3, 11);
```

```

    B.setLB(5, 10);
    cout << "Fl\x84" << "che von A: "
         << A.berechneFl() << endl;
    cout << "Fl\x84" << "che von B: "
         << B.berechneFl() << endl;
    return 0;
}

```

Ausgabe:

```

Fläche von A: 33
Fläche von B: 50

```

### 24.1.1 Methoden außerhalb einer Klasse definieren

Es ist zu berücksichtigen, dass Methoden, die innerhalb einer Klasse definiert sind, automatisch als inline-spezifiziert angesehen werden.

#### Referenz

Zu inline-Funktionen siehe Kapitel 20, Abschnitt 20.7 »inline-Funktionen«.

Demnach ist es empfehlenswert, zumindest größere Methoden außerhalb ihrer Klasse zu definieren. In diesem Fall verwendet man Prototypen, um die Methoden innerhalb der Klasse zu deklarieren. Die entsprechende Methodendefinition außerhalb der Klasse wird eingeleitet mit Angabe des Klassennamens plus Bereichsauflösungsoperator (::). Damit macht man die Zugehörigkeit einer Methode zu einer bestimmten Klasse deutlich. Dies sei an folgendem Beispiel gezeigt, in dem die Methoden `addiereRe()` und `setLB()` der Klasse `Rechteck` außerhalb dieser Klasse definiert sind:

```

#include <iostream>
using namespace std;

class Rechteck
{
public:
    int Laenge;
    int Breite;
    int berechneFl()
    {return Laenge * Breite;}
    void addiereRe(Rechteck einAnderes);
    void setLB(int l, int b);
};

```

```

void Rechteck::addiereRe(Recteck einAnderes)
{
    Laenge += einAnderes.Laenge;
    Breite += einAnderes.Breite;
}

void Rechteck::setLB(int l, int b)
{
    Laenge = l;
    Breite = b;
}

int main(void)
{
    Rechteck A, B;
    A.setLB(8, 9);
    B.setLB(10, 10);
    cout << "Fl\x84" << "che von A: "
         << A.berechneFl() << endl;
    cout << "Fl\x84" << "che von B: "
         << B.berechneFl() << endl;
    return 0;
}

```

Ausgabe:

```

Fläche von A: 72
Fläche von B: 100

```

Da die Methode `berechneFl()` lediglich aus einer Anweisung besteht, ist es unproblematisch, diese als `inline` zu definieren – was eigentlich auch für die beiden anderen Methoden zutrifft. In der Regel definiert man Methoden mit mehr als zwei Anweisungen aber außerhalb ihrer Klasse.

### Hinweis

Es ist üblich, die Definition von sehr kleinen Methoden in kompakter Form zu notieren. Der Quellcode wird dadurch übersichtlicher und kürzer:

```

int berechneFl()
{return Laenge * Breite;}

```

bzw.

```

int berechneFl() {return Laenge * Breite;}

```

### 24.1.2 Den Code von Klassendefinitionen auf mehrere Dateien verteilen

Falls man die Definition einer Klasse im Hinblick auf deren Wiederverwendbarkeit in anderen Programmen separieren möchte, schreibt man das Gerüst der Klassendefinition in eine eigens dafür vorgesehene Headerdatei, der man in der Regel den Namen der Klasse gibt. Bei mehreren Klassen empfiehlt es sich, für jede Klasse eine eigene Headerdatei anzulegen.

Den Code von zugehörigen Methoden, die außerhalb der Klasse definiert sind, schreibt man in eine gleichnamige Quellcodedatei. Für unser Beispiel ergeben sich damit die Dateien *Rechteck.h* und *Rechteck.cpp* sowie die Datei *Haupt.cpp*, in der die Klasse Rechteck verwendet wird.

*Rechteck.h:*

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

class Rechteck
{
public:
    int Laenge;
    int Breite;
    int berechneFl()
    {return Laenge * Breite;}
    void addiereRe(Rchteck einAnderes);
    void setLB(int l, int b);
};
```

*Rechteck.cpp:*

```
#include <iostream>
#include "Rechteck.h"
using namespace std;

void Rechteck::addiereRe(Rchteck einAnderes)
{
    Laenge += einAnderes.Laenge;
    Breite += einAnderes.Breite;
}

void Rechteck::setLB(int l, int b)
{
    Laenge = l;
    Breite = b;
}
```

*Haupt.cpp*:

```
#include <iostream>
#include "Rechteck.h"
using namespace std;

int main(void)
{
    Rechteck A, B;
    A.setLB(8, 9);
    B.setLB(10, 10);
    cout << "Fl\x84" << "che von A: "
         << A.berechneFl() << endl;
    cout << "Fl\x84" << "che von B: "
         << B.berechneFl() << endl;
    return 0;
}
```

Ausgabe:

```
Fläche von A: 72
Fläche von B: 100
```

### CD-ROM

Die Dateien *Haupt.cpp*, *Rechteck.cpp* und *Rechteck.h* finden Sie gepackt zu *k24e.zip* auf der Buch-CD im Ordner *Beispiele/K24*.

Beachten Sie, dass im Code von *Haupt.cpp* allein die Headerdatei *Rechteck.h* einzubinden ist, nicht aber etwa die Quelldatei *Rechteck.cpp*, in der sich die Definitionen der zugehörigen Methoden *addiereRe()* und *setLB()* befinden.

```
#include "Rechteck.h"
```

Da die genannten Methoden in *Rechteck.h* deklariert sind, können sie im Code der Datei *Haupt.cpp* ohne weiteres verwendet werden.

### 24.1.3 Wie man vermeidet, dass eine Klassendefinition mehrfach eingebunden wird

Nun kann es vorkommen, dass eine andere Klasse die Klasse *Rechteck* verwendet. Nennen wir als Beispiel eine fiktive Klasse *XY*. Gemeint ist damit z.B. der Fall, dass *XY* als Element ein Objekt des Typs *Rechteck* enthält:

```
class XY
{
public:
    ...
    Rechteck Eins;
    ...
};
```

Natürlich wäre genauso gut der Fall denkbar, dass in einer Methode von XY ein lokales Objekt des Typs Rechteck oder ein Parameter dieses Typs definiert ist:

```
class XY
{
public:
    ...
    void eineMethode(Rchteck A);
    ...
};
```

In beiden Fällen ist es für die Definition der Klasse XY erforderlich, die Headerdatei *Rechteck.h* einzubinden. Demnach würde sich für eine Headerdatei *XY.h* in etwa folgendes Bild ergeben:

*XY.h:*

```
#include <iostream>
#include "Rechteck.h"
using namespace std;

class XY
{
public:
    ...
    Rechteck Eins;
    ...
};
```

Wenn Sie nun an einem Programm arbeiten, das beide Klassen – Rechteck und XY – verwendet, kann es leicht passieren, dass die Klasse Rechteck doppelt eingebunden wird, nämlich einmal mit der Anweisung

```
#include "Rechteck.h"
```

und mit

```
#include "XY.h"
```

ein zweites Mal, da *XY.h* die Headerdatei *Rechteck.h* bereits mit der gleichen `#include`-Direktive einbindet. Im Ergebnis enthält beispielsweise der Code folgender Datei zwei Klassendefinitionen *Rechteck*, was nicht sein darf:

```
#include <iostream>
#include "Rechteck.h"
#include "XY.h"
using namespace std;

int main(void)
{
    ...
    return 0;
}
```

### Hinweis

Es sei darauf hingewiesen, dass der genannte Sachverhalt in Bezug auf Funktionsdeklarationen kein Problem darstellt (zur Deklaration von Funktionen siehe Kapitel 20, Abschnitt 20.2 »Funktionsprototypen«). Im Gegensatz zu Klassendefinitionen dürfen Funktionsprototypen auch mehrfach im Code stehen.

Zwar lässt sich das Problem im obigen Beispiel leicht beheben, indem man die Zeile

```
#include "Rechteck.h"
```

einfach weglässt. Wenn ein Programm jedoch eine Vielzahl von benutzerdefinierten Klassen verwendet, ist jedoch diesbezüglich eine allzu große Aufmerksamkeit gefordert. Sicherer ist es, mit folgender Konstruktion ein für alle Mal zu verhindern, dass Klassendefinitionen doppelt in den Code integriert werden:

```
#ifndef KONSTANTE
#define KONSTANTE
Code
#endif
```

Die angegebenen Präprozessor-Direktiven sind jeweils in den entsprechenden Headerdateien zu verwenden. *Konstante* ist dabei eine beliebige Konstante. Sie sollte sich jedoch nach dem Namen der Klasse bzw. der Headerdatei richten, damit nicht in anderen Headerdateien versehentlich eine gleichnamige Konstante definiert wird. Zwischen die Zeilen

```
#define KONSTANTE
```

und

```
#endif
```

schreiben Sie den eigentlichen Code der Headerdatei, also die Klassendefinition, eventuell weitere Präprozessor-Direktiven wie z.B. `#include <iostream>` etc.

### Referenz

Die Bedeutung der Präprozessor-Direktiven `#ifndef`, `#define` und `#endif` erfahren Sie in Kapitel 27 in den Abschnitten 27.2 »Symbolische Konstanten mit `#define` vereinbaren« und 27.3 »Bedingte Kompilierung«.

Damit ergibt sich für unsere Headerdatei *Rechteck.h* folgender Inhalt:

```
#ifndef RECHTECK_H
#define RECHTECK_H

#include <iostream>
using namespace std;

class Rechteck
{
public:
    int Laenge;
    int Breite;
    int berechneFl()
    {return Laenge * Breite;}
    void addiereRe(Rchteck einAnderes);
    void setLB(int l, int b);
};

#endif
```

## 24.2 Zugriffsspezifizierer

Wir werden im Folgenden eine Klasse einrichten, die den Kunden einer Bank wiedergeben soll. Dementsprechend nennen wir die Klasse *Bankkunde*. Ein Bankkunde hat einen Vor- und einen Nachnamen, eine Konto- bzw. Kundennummer, einen Kontostand, einen Kreditrahmen, eine Geheimzahl für sein Konto, eine Telefonnummer, möglicherweise eine E-Mail-Adresse oder Faxnummer, eine Adresse, also Straße, Postleitzahl und Wohnort etc. Als Methoden kommen beispielsweise `einzahlen()` und `abheben()` in Frage.

Versorgen wir die Klasse *Bankkunde* zunächst mit dem Allernotwendigsten: Name, Kontonummer und Kontostand. Damit soll unser Bankkunde fürs Erste ausreichend identifiziert sein:



```
class Bankkunde
{
public:
    string vorname;
    string nachname;
    int kto_nr;
    double kto_stand;
};
```

Wenn man in obiger Klassendefinition auf den Spezifizierer `public` verzichtet, kann auf keines der Klassenelemente von außen zugegriffen werden. Daher resultiert aus dem folgenden Versuch, unseren ersten Bankkunden um 20.000 Euro zu bereichern, lediglich eine Fehlermeldung:

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

class Bankkunde
{
    string vorname;
    string nachname;
    int kto_nr;
    double kto_stand;
};

int main(void)
{
    Bankkunde Erster;
    Erster.kto_stand = 20000; // FEHLER
    return 0;
}
```

Den Grund hierfür kennen Sie bereits. Anders als die Elemente einer mit dem Schlüsselwort `struct` definierten Klasse sind die einer mit `class` definierten Klasse per Voreinstellung `private`, und auf `private`-Elemente einer Klasse kann nicht direkt zugegriffen werden, wie am obigen Beispiel zu sehen. Direkt heißt von außerhalb der Klasse. Methoden einer Klasse können immer auf die Datenelemente ihrer Klasse zugreifen. Dabei spielt es keine Rolle, wo die Methodendefinition steht, ob innerhalb oder außerhalb der Klassendefinition. Entscheidend ist allein die Zugehörigkeit einer Methode zu einer bestimmten Klasse.

Um dem Bankkunden `Erster` dennoch den genannten Betrag zukommen zu lassen, ließe sich nun selbstverständlich das betreffende Element – oder gleich alle – als `public` definieren. Dies geschieht mit dem entsprechenden Schlüsselwort plus eines Doppelpunktes. Der Doppelpunkt besagt, dass ab dem betreffenden Element alle weiteren Elemente

public sind. Entsprechendes gilt natürlich für den Zugriffsspezifizierer private. Beide dürfen auch abwechselnd verwendet werden:

```
class XYZ
{
public:
    int a;
    ...
private:
    double b;
    ...
public:
    char c;
    ...
};
```

In der Regel führt man die private-Elemente einer Klasse zuerst auf und im Anschluss daran die public-Elemente. Wobei man zur Verdeutlichung den Zugriffsspezifizierer private auch explizit angibt, obwohl dazu keine syntaktische Notwendigkeit besteht:

```
class XYZ
{
private:
    ...
public:
    ...
};
```

Bezüglich der Klasse Bankkunde könnte man es nun auch bei der private-Einstellung des Attributs kto\_stand belassen und für den Zugriff geeignete public-Methoden definieren:

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

class Bankkunde
{
private:
    string vorname;
    string nachname;
    int kto_nr;
    double kto_stand;
public:
    void setzeKontostand(double betrag)
    {kto_stand = betrag;}
    double zeigeKontostand()
```

```
{return kto_stand;}
};

int main(void)
{
    Bankkunde Erster;
    Erster.setzeKontostand(20000);
    cout << Erster.zeigeKontostand();
    return 0;
}
```

Ausgabe: 20000

Nun ist die Eigenschaft `kto_stand` nach außen verborgen. Der Zugriff erfolgt allein über die Methoden `setzeKontostand()` und `zeigeKontostand()`.

### Hinweis

Die Gewohnheit, Methoden, die den Wert eines Datenelements zurückgeben, als »get-Methoden«, und Methoden, die Datenelementen einen neuen Wert zuweisen, als »set-Methoden« zu bezeichnen, ist sehr weit verbreitet. Demgemäß benennen wir im Folgenden die Methode `setzeKontostand()` in `setKontostand()` und die Methode `zeigeKontostand()` in `getKontostand()` um.

Denken Sie daran, dass eine Klasse für andere Programmierer oder für Sie selbst als Programmierer implementiert wird. Ziel ist es einmal, den Benutzern einer Klasse möglichst umfangreiche Möglichkeiten in die Hand zu geben, aber auch den Programmierer vor Fehlern zu schützen. In diesem Sinne wird man versuchen, die Schnittstellen zwischen Klasse und Programmierer möglichst sicher zu gestalten. Will man etwa von vornherein unterbinden, dass eine Eigenschaft eines Klassenobjekts durch ein Versehen des Programmierers – dem Benutzer dieser Klasse – einen negativen Wert erhält, so lässt sich dies allein dadurch erreichen, dass man das Datenelement selbst vor dem direkten Zugriff schützt und solche inkorrekten Zuweisungen in der entsprechenden Methode verhindert:

```
class Mensch
{
private:
    int alter;
    ...
public:
    void setAlter(int a)
    {
        if (a >= 0)
            alter = a;
    }
    ...
};
```

Das Prinzip, Daten in der beschriebenen Weise nach außen hin zu verstecken, bezeichnet man als »Kapselung«. Bei einer konsequent durchgeführten Kapselung bleibt der innere Aufbau einer Klasse und damit der eines Objekts dieser Klasse für den Benutzer verborgen. Dies hat einmal den Vorteil, dass Objekte ihre Zustände nur in genau definierten Bahnen ändern. Des Weiteren lassen sich innerhalb der Klasse Veränderungen durchführen, ohne dass sich dies auf übrige Codeteile nachteilig auswirkt, solange man die Schnittstelle zur Außenwelt – also z.B. Parameterliste und Rückgabetypp von öffentlichen Methoden – nicht ändert. Dies gilt auch für andere Programme (anderer Programmierer), die Ihre Klasse nutzen. Es ist dann in aller Regel problemlos möglich, eine neue Version Ihrer Klasse einzubinden, ohne dass am Programm Veränderungen durchgeführt werden müssen.

### Hinweis

Dem objektorientierten Programmierkonzept folgend, sollten Datenelemente einer Klasse möglichst `private` sein, der Zugriff auf diese ausschließlich über Methoden erfolgen. Daneben sind natürlich auch `private`-Methoden denkbar, sofern diese allein dazu vorgesehen sind, innerhalb der Klasse, also von anderen Methoden dieser Klasse, verwendet zu werden.

Setzt man das eben Dargelegte auf die Klasse `Bankkunde` um, so ergibt sich für diese nunmehr folgendes Aussehen.

```
class Bankkunde
{
private:
    string vorname;
    string nachname;
    int kto_nr;
    double kto_stand;
public:
    void setVorname(string v)
    {vorname = v;}
    string getVorname()
    {return vorname;}
    void setNachname(string n)
    {nachname = n;}
    string getNachname()
    {return nachname;}
    void setKontonr(int nr)
    {kto_nr = nr;}
    int getKontonr()
    {return kto_nr;}
    void setKontostand(double betrag)
    {kto_stand = betrag;}
```

```
double getKontostand()
{return kto_stand;}
};
```

Hinzugefügt werden soll nun noch ein privates Datenelement `limit` mit den entsprechenden öffentlichen Methoden `setLimit()` und `getLimit()`, das den Kreditrahmen eines Kunden anzeigt. Das heißt, der Kontostand eines Bankkunden soll nicht den Betrag `-limit` unterschreiten (also den negativen Betrag von `limit`). Des Weiteren soll die Klasse um die Methoden `einzahlen()` und `auszahlen()` ergänzt werden. In der Methode `auszahlen()` wird dafür gesorgt, dass eine Auszahlung nicht stattfindet, falls diese den Kreditrahmen des betreffenden Kunden sprengen würde:

```
class Bankkunde
{
private:
    string vorname;
    string nachname;
    int kto_nr;
    double kto_stand;
    double limit;
public:
    void setVorname(string v)
    {vorname = v;}
    string getVorname()
    {return vorname;}
    void setNachname(string n)
    {nachname = n;}
    string getNachname()
    {return nachname;}
    void setKontonr(int nr)
    {kto_nr = nr;}
    int getKontonr()
    {return kto_nr;}
    void setKontostand(double betrag)
    {kto_stand = betrag;}
    double getKontostand()
    {return kto_stand;}
    void setLimit(double l);
    double getLimit()
    {return limit;}
    void einzahlen(double betrag);
    int auszahlen(double betrag);
};
```

```

void Bankkunde::setLimit(double l)
{
    if (l >= 0)
        limit = l;
}

void Bankkunde::einzahlen(double betrag)
{
    if (betrag > 0)
        kto_stand += betrag;
}

int Bankkunde::auszahlen(double betrag)
{
    if (betrag <= 0)
        return 2; // negativer Auszahlungsbetrag
    if (kto_stand - betrag >= -limit)
    {
        kto_stand -= betrag;
        return 0; // Auszahlung ist erfolgt
    }
    else
        return 1; // Auszahlung nicht moeglich
}

```

Die Methode `auszahlen()` ist mit einem Rückgabewert definiert, der anzeigt, ob die Auszahlung stattgefunden hat bzw. aus welchen Gründen sie nicht erfolgt ist. Der Rückgabewert 1 weist darauf hin, dass die Auszahlung nicht möglich ist – und deshalb nicht durchgeführt wird –, da diese den Kreditrahmen des betreffenden Kunden überschreiten würde. Falls die Methode vom Programmierer inkorrekt verwendet wird (negativer Auszahlungsbetrag oder Auszahlungsbetrag von 0), liefert sie den Wert 2 zurück. In einem Programm könnte die Methode `auszahlen()` in etwa wie folgt verwendet werden:

```

...
Bankkunde einKunde;
...
if (einKunde.auszahlen(1000) == 1)
    cout << "Betrag kann nicht ausgezahlt werden";
...

```

Beachten Sie, dass der Auszahlungsbetrag verbucht wurde, falls die `if`-Bedingung zu `false` auswertet. Die Methode `auszahlen()` wird ja in jedem Fall ausgeführt. Die Tatsache, dass deren Rückgabewert in einer `if`-Bedingung verwendet wird, ändert daran nichts. Über die Durchführung der Kontobewegung entscheidet allein die Methode, wobei sie das Ergebnis nach außen mit einem entsprechenden Rückgabewert kundtut.

## 24.3 Konstruktoren und Destruktoren

Für jede Klasse lassen sich zwei spezielle Methoden definieren: ein so genannter »Konstruktor« und ein so genannter »Destruktor«, wobei Ersterer auch überladen werden kann. Das Besondere an beiden Methoden ist, dass diese automatisch aufgerufen werden, der Konstruktor beim Instanzieren eines Objekts, der Destruktor, wenn die Lebensdauer eines Objekts endet.

### Hinweis

Ein Klassenobjekt wird auch als Instanz dieser Klasse bezeichnet. Dementsprechend spricht man vom »Instanzieren« eines Objekts, wenn dieses erzeugt wird.

Für die Definition eines Konstruktors gilt:

- Er muss den Namen der Klasse tragen.
- Ein Konstruktor kann niemals einen Wert zurückgeben. Daher ist in der Definition auch kein entsprechender Rückgabetyt anzugeben, also auch nicht `void`.
- Mindestens ein Konstruktor der Klasse sollte in der Regel `public` sein.

Damit ergibt sich für einen Konstruktor folgende allgemeine Syntax:

```
Klassenname(Parameterliste)
{
    Anweisung(en)
}
```

Die oben genannten drei Punkte gelten auch für Destruktoren. Allerdings ist die Definition eines Destruktors mit einer Tilde (~) einzuleiten. Zudem können Destruktoren keine Werte übernehmen. Das bedeutet, für einen Destruktor dürfen keine formalen Parameter definiert werden.

```
~Klassenname()
{
    Anweisung(en)
}
```

### Hinweis

Als Rückgabewert darf wie beschrieben auch nicht `void` verwendet werden, die Angabe von `void` in den Klammern (Parameterliste) ist jedoch erlaubt:

```
~Klassenname(void)
{
    Anweisung(en)
}
```

Theoretisch dürfen in beiden Methoden zwar beliebige Anweisungen stehen, im Allgemeinen wird man dort aber nur solche Anweisungen verwenden, die dem eigentlichen Sinn und Zweck dieser Methoden entsprechen (dazu gleich mehr). Abweichend davon ist im folgenden Beispiel der Konstruktor sowie der Destruktor der Punkt-Klasse mit einer Ausgabeanweisung versehen. Dies geschieht, um deutlich zu machen, wie bzw. wann diese Methoden zur Ausführung gelangen:

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

class Punkt
{
public:
    int x;
    int y;
    Punkt()
    {cout << "Konstruktor" << endl;}
    ~Punkt()
    {cout << "Destruktor" << endl;}
};

int main(void)
{
    Punkt punkte[3];
    cout << "Vor innerem Block" << endl;
    {
        Punkt einPunkt;
    }
    cout << "Nach innerem Block" << endl;
    return 0;
}
```

Ausgabe:

```
Konstruktor
Konstruktor
Konstruktor
Vor innerem Block
Konstruktor
Destruktor
Nach innerem Block
Destruktor
Destruktor
Destruktor
```



Mit der Anweisung

```
Punkt punkte[3];
```

entstehen drei Objekte des Typs `Punkt`. Für jedes dieser Objekte wird die Konstruktormethode genau ein Mal aufgerufen. Im inneren Block wird das Objekt `einPunkt` instanziiert, für das die Konstruktormethode ebenfalls ein Mal ausgeführt wird:

```
Punkt einPunkt;
```

Dieses Objekt hört mit dem Verlassen des inneren Blocks auf zu existieren. Daher wird zunächst der Destruktor für dieses Objekt aufgerufen. Bei Programmende, wenn auch die Lebensdauer der Objekte `punkte[0]`, `punkte[1]` und `punkte[2]` endet, wird der Destruktor für die Objekte ausgeführt.

Destruktoren werden vorwiegend benutzt, um Speicherplatz, der dynamisch angefordert wurde, mit dem Erlöschen eines Objekts wieder freizugeben, also den Speicher aufzuräumen. Genauer dazu erfahren Sie in Kapitel 28 im Abschnitt 28.6.2 »Destruktoren sorgen für die Aufräumarbeiten«.

Konstruktoren werden dagegen in der Regel dazu verwendet, Datenelemente eines Objekts zu initialisieren oder sonstige Initialisierungsarbeiten zur Einrichtung des Objekts auszuführen:

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

class Punkt
{
public:
    int x;
    int y;
    Punkt()
    {x = 0; y = 0;}
};

int main(void)
{
    Punkt einPunkt;
    cout << "x-Koordinate von einPunkt: "
          << einPunkt.x << endl;
    cout << "y-Koordinate von einPunkt: "
          << einPunkt.y << endl;
    return 0;
}
```

Ausgabe:

```
x-Koordinate von einPunkt: 0
y-Koordinate von einPunkt: 0
```

### 24.3.1 Überladen von Konstruktoren

Wie bereits angedeutet, lassen sich Konstruktoren auch überladen:

```
class Punkt
{
public:
    int x;
    int y;
    Punkt()
    {x = 0; y = 0;}
    Punkt(int xko, int yko)
    {x = xko; y = yko;}
};
```

Bei der obigen Klasse Punkt kann es sich der Programmierer also aussuchen, welcher der beiden Konstruktoren beim Erzeugen eines Objekts der Klasse Punkt aufgerufen werden soll: Der parameterlose oder derjenige, dem beim Aufruf zwei Integerwerte zu übergeben sind. Für Letzteren sind die Argumente bei der Objektinstanzierung in Klammern anzugeben:

```
Punkt A(3, 5);
```

Falls Sie den Standardkonstruktor verwenden möchten, schreiben Sie

```
Punkt B;
```

#### Hinweis

Parameterlose Konstruktoren werden »Standardkonstruktoren« genannt.

Nach Ausführung obiger Anweisungen besitzt Punkt A somit die Koordinaten 3, 5 und Punkt B die Koordinaten 0, 0:

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

class Punkt
{
```

```
public:
    int x;
    int y;
    Punkt()
    {x = 0; y = 0;}
    Punkt(int xko, int yko)
    {x = xko; y = yko;}
};

int main(void)
{
    Punkt A(3, 5), B;
    cout << "Koordinaten von A: " << A.x
         << ", " << A.y << endl;
    cout << "Koordinaten von B: " << B.x
         << ", " << B.y << endl;
    return 0;
}
```

Ausgabe:

```
Koordinaten von A: 3, 5
Koordinaten von B: 0, 0
```

### Hinweis

Dementsprechend können Sie in der gleichen Weise auch eine string-Variable mit einem Anfangswert versehen. Wie Sie ja wissen, handelt es sich dabei um ein Objekt der Klasse string. In diesem Sinne sind die Initialisierungen

```
string vorname = "Nicole";
```

und

```
string vorname("Nicole");
```

gleichwertig.

### Achtung

Der Aufruf eines Konstruktors erfolgt ausschließlich automatisch. Anweisungen wie

```
einPunkt.Punkt(7, 11); // FEHLER
```

oder

```
einPunkt.Punkt() // FEHLER
```

sind daher fehlerhaft. Wobei einPunkt ein Objekt der obigen Klasse Punkt sein soll.

### 24.3.2 Ersatzkonstruktor

Falls in einer Klasse überhaupt kein Konstruktor definiert ist, fügt Ihr Compiler automatisch einen Standardkonstruktor als Ersatz hinzu, mehr aus formellen Gründen, da dieser rein gar nichts macht. Demgemäß hat man ihn sich wie folgt vorzustellen:

```
Klassenname() {}
```

Sobald sich in einer Klasse jedoch ein benutzerdefinierter Konstruktor befindet, sorgt Ihr Compiler nicht mehr für einen Ersatzkonstruktor. Dies kann zu Problemen führen, falls in einer Klasse ein Konstruktor mit Parametern definiert ist, aber kein Standardkonstruktor. In diesem Fall ist es nicht möglich, Objekte dieser Klasse per Automatismus zu erzeugen. So führt der folgende Versuch, ein Objekt der Klasse *Strecke* zu instanzieren, zu einer Fehlermeldung:

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

class Punkt
{
public:
    int x;
    int y;
    Punkt(int xko, int yko)
    {x = xko; y = yko;}
};

class Strecke
{
public:
    Punkt A;
    Punkt B;
};

int main(void)
{
    Strecke Erste; // FEHLER
    return 0;
}
```

Beachten Sie, dass das Instanzieren eines *Strecke*-Objekts zugleich das zweier *Punkt*-Objekte zur Folge hat. Hier besteht jedoch keine Möglichkeit, der einzigen Konstruktormethode der Klasse *Punkt* die geforderten Werte zu übergeben.

Abhilfe schafft man bereits dadurch, indem man die obige Klasse Punkt mit einem minimalen Standardkonstruktor ausstattet:

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

class Punkt
{
public:
    int x;
    int y;
    Punkt() {}
    Punkt(int xko, int yko)
    {x = xko; y = yko;}
};

class Strecke
{
public:
    Punkt A;
    Punkt B;
};

int main(void)
{
    Strecke Erste; // OK
    // ...
    return 0;
}
```

### **Tipp**

Es ist also auf jeden Fall zu empfehlen, für jede Klasse einen Standardkonstruktor zu definieren.

### **Hat-Beziehung/Ist-Beziehung**

Eine Strecke hat zwei Punkte, ein Auto hat einen Motor. In der objektorientierten Programmierung spricht man von der »Hat-Beziehung« (Englisch »has a«), wenn man sagen kann, dass ein Objekt ein anderes enthält. Das Gegenstück dazu ist die so genannte »Ist-Beziehung« (Englisch »is a«): Ein Auto ist ein Fahrzeug, ein Mensch ist ein Lebewesen. Die Ist-Beziehung ist für die Vererbung bedeutsam (zum Thema Vererbung siehe Kapitel 30).

### Hat-Beziehung/Ist-Beziehung (Forts.)

In der Regel wird man versuchen, in den Beziehungen der Klassen untereinander die Wirklichkeit widerzuspiegeln. So wird man eine Klasse `Auto` möglicherweise von einer Klasse `Fahrzeug` erben lassen, da es sich in der realen Welt so verhält, dass ein `Auto` eine spezielle Art von `Fahrzeug` ist. Andererseits bietet es sich an, einer Klasse `Auto` ein Objekt einer Klasse `Motor` hinzuzufügen.

Wir versehen nun unsere Klasse `Bankkunde` mit einem Standardkonstruktor, der das Attribut `kto_stand` mit dem Wert 0 und das Attribut `limit` mit dem Wert 1000 initialisiert. Das bedeutet, jeder neue `Bankkunde` beginnt mit Kontostand Null und einem Kreditrahmen von 1.000 Euro:

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

class Bankkunde
{
private:
    string vorname;
    string nachname;
    int kto_nr;
    double kto_stand;
    double limit;
public:
    Bankkunde()
    {kto_stand = 0; limit = 1000;}
    void setVorname(string v)
    {vorname = v;}
    string getVorname()
    {return vorname;}
    void setNachname(string n)
    {nachname = n;}
    string getNachname()
    {return nachname;}
    void setKontonr(int nr)
    {kto_nr = nr;}
    int getKontonr()
    {return kto_nr;}
    void setKontostand(double betrag)
    {kto_stand = betrag;}
    double getKontostand()
    {return kto_stand;}
```

```
void setLimit(double l);
double getLimit()
{return limit;}
void einzahlen(double betrag);
int auszahlen(double betrag);
};

void Bankkunde::setLimit(double l)
{
    if (l >= 0)
        limit = l;
}

void Bankkunde::einzahlen(double betrag)
{
    if (betrag > 0)
        kto_stand += betrag;
}

int Bankkunde::auszahlen(double betrag)
{
    if (betrag <= 0)
        return 2; // negativer Auszahlungsbetrag
    if (kto_stand -betrag >= -limit)
    {
        kto_stand -= betrag;
        return 0; // Auszahlung ist erfolgt
    }
    else
        return 1; // Auszahlung nicht moeglich
}

int main(void)
{
    Bankkunde Neu;
    double geld;
    cout << fixed << setprecision(2);
    cout << "Dein Kontostand: " << Neu.getKontostand()
        << " Euro" << endl;
    cout << "Wie viel Geld soll ich Dir geben? ";
    do {
        cin >> geld;
    } while (geld < 0);
    if (geld == 0)
        cout << "Bist Du bescheiden!" << endl;
    else if (Neu.auszahlen(geld) == 1)
        cout << "Kriegst Du nicht!" << endl;
```

```

else
{
    cout << "Kriegst Du!" << endl;
    cout << "Dein Kontostand: "
        << Neu.getKontostand() << " Euro" << endl;
}
return 0;
}

```

Ausgabe:

```

Dein Kontostand: 0.00 Euro
Wie viel Geld soll ich Dir geben? 900
Kriegst Du!
Dein Kontostand: -900.00 Euro

```

## 24.4 Was es mit Namensräumen auf sich hat

Ein Namensraum, auch Namensbereich genannt, beschreibt einen eigenen Gültigkeitsbereich. Das heißt, zwischen zwei Namensräumen können keine Namenskonflikte entstehen. Folglich ist es erlaubt, in verschiedenen Namensräumen gleich lautende Klassen, Funktionen, Variablen, Objekte etc. zu definieren.

### Hinweis

Wie Sie sich vermutlich denken können, spielt die Definition von Namensräumen allein bei sehr großen Programmen eine Rolle.

Die Definition eines Namensraums wird mit dem Schlüsselwort `namespace` plus Bezeichner für den Namensraum eingeleitet. Innerhalb des zugehörigen Blocks kann man sowohl Typen als auch Funktionen, Variablen usw. definieren.

```

namespace meinNamensraum
{
    double v;

    class ABC
    {
        ...
    };

    class XYZ
    {
        ...
    };
}

```



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

Die Definition eines Namensraums kann sogar in mehreren Teilen erfolgen:

```
namespace meinNamensraum
{
    double v;

    class ABC
    {
        ...
    };
    ...
}
...
namespace meinNamensraum
{
    class XYZ
    {
        ...
    };

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

Beide Teile werden praktisch addiert.

Der Zugriff auf die Elemente eines Namensraums erfolgt außerhalb desselben über den Bereichsauflösungsoperator (::).

```
meinNamensraum::v = 2.99;
meinNamensraum::f();
meinNamensraum::ABC obj;
```

Mit der letzten Anweisung wird ein Objekt der Klasse ABC des Namensraums meinNamensraum instanziiert, mit der vorletzten die Funktion f() desselben Namensraums aufgerufen.

Eine andere Möglichkeit besteht darin, mit der `using`-Direktive von vornherein bekannt zu geben, dass man Elemente eines bestimmten Namensraums verwenden möchte:

```
using namespace meinNamensraum;
```

Im Weiteren ist dann der Zugriff auch ohne Bereichsauflösung möglich:

```
v = 64.1;  
f();  
ABC einObjekt;
```

### Hinweis

Allerdings können bei dieser Art des Zugriffs Namenskonflikte entstehen, da nun Elemente des Namensbereichs gegebenfalls mit gleich lautenden Elementen des aktuellen Gültigkeitsbereichs kollidieren.

Auf die gleiche Weise lässt sich der unqualifizierte Zugriff auch nur für ein einziges Element eines Namensbereichs festlegen:

```
using namespace meinNamensraum::ABC;
```

Nach obiger Anweisung kann allein der Typbezeichner `ABC` ohne Bereichsauflösung verwendet werden. Die anderen Elemente des Namensbereichs `meinNamensraum` müssen nach wie vor qualifiziert angesprochen werden:

```
ABC eineInstanz;
```

aber

```
meinNamensraum::v = 745.81;  
meinNamensraum::f();
```

### Hinweis

Alle Elemente der Standardbibliothek sind im Namensraum `std` definiert (siehe dazu Kapitel 20, Abschnitt 20.2.2 »Prototypen von vordefinierten Funktionen«).

# 25

## Statische Elemente einer Klasse

Manchmal möchte man, dass Elemente zwar zu einer bestimmten Klasse gehören, aber nicht zu bestimmten Objekten dieser Klasse. Vielmehr sollen diese Elemente, bei denen es sich sowohl um Eigenschaften als auch um Methoden handeln kann, für alle Objekte einer Klasse gleichzeitig verfügbar sein. In diesem Fall spricht man von statischen Elementen. Diese werden mit dem Schlüsselwort `static` definiert.

Es sei daran erinnert, dass ein Attribut `x` einer Klasse `XY` z.B. zehnmal existiert, falls im Programm zum betrachteten Zeitpunkt genauso viele Objekte existent sind. Jedes Objekt besitzt dann ja sein eigenes `x`. Das Schlüsselwort `static` besitzt in diesem Zusammenhang eine ganz besondere Bedeutung: Es legt fest, dass das Element zu einem »echten« Klassenelement wird, das über den Klassennamen aufgerufen werden kann und keinen Bezug zu einem speziellen Objekt hat.

### 25.1 Statische Attribute

Die Vereinbarung eines statischen Klassenattributs gestaltet sich wie folgt:

- Innerhalb einer Klasse wird ein statisches Datenelement mit dem Schlüsselwort `static` plus Datentypangabe – und natürlich einem Bezeichner für das Datenelement – bekannt gemacht.
- Die eigentliche Vereinbarung erfolgt außerhalb der Klasse ohne das Schlüsselwort `static`. Dabei ist der qualifizierte Name, also Klassenname plus Bereichsauflösungsoperator, zu verwenden:

```
class XY
{
public:
    static int statElem;
    ...
};
int XY::statElem = 0;
...
```

Demnach ist erst die Anweisung

```
int XY::statElem = 0;
```

mit Speicherreservierung verbunden.

Da statische Datenelemente einer Klasse wie globale Variablen die Speicherklasse `static` besitzen, werden sie automatisch mit dem Anfangswert 0 versehen. Von daher ist die explizite Initialisierung im obigen Beispiel nicht notwendig. Natürlich können Sie ein statisches Datenelement auch mit jedem anderen Wert initialisieren:

```
int XY::statElem = 50;
```

Wie gesagt, muss die Initialisierung jedoch außerhalb der Klasse erfolgen. Innerhalb dieser ist das Datenelement ja noch nicht existent.

### Hinweis

Denken Sie daran, dass auch nicht statische Datenelemente erst mit dem Instanzieren von Klassenobjekten anfangen zu existieren. Die bloße Definition einer Klasse bewirkt in dieser Hinsicht rein gar nichts, abgesehen davon, dass damit der Bauplan für zukünftige Objekte geschaffen wird.

In der Regel legt man in `static`-Datenelementen solche Informationen ab, die man ansonsten in einer globalen Variablen speichern würde. Was man natürlich theoretisch ebenfalls tun könnte. Allerdings widerspricht die Verwendung von globalen Variablen für Daten, die sinngemäß zu einer bestimmten Klasse gehören, dem Prinzip der Kapselung. Statische Klassenattribute haben dagegen den Vorzug, dass sie die gleiche Lebensdauer wie globale Variablen, ihr Gültigkeitsbereich aber auf die Klasse, in der sie definiert sind, beschränkt ist.

### Hinweis

Die Lebensdauer eines `static`-Datenelements einer Klasse erstreckt sich vom Zeitpunkt seiner Vereinbarung bis zum Programmende. Das bedeutet, dass es auch dann existiert, wenn kein Objekt der betreffenden Klasse instanziiert ist.

Der Einsatz eines `static`-Attributs eignet sich beispielsweise, um für eine Klasse `Auto` im Programm jederzeit feststellen zu können, wie viele Objekte dieser Klasse – also Autos – existieren. Dazu inkrementiert man das entsprechende Datenelement im Konstruktor und dekrementiert es im Destruktor. Folgerichtig vereinbart man es mit dem Anfangswert 0, da zu Beginn ja noch keine Klassenobjekte existent sind:

```
class Auto
{
public:
    static int anzahl;
    // ...
    Auto()
    {anzahl++;}
    ~Auto()
```

```
Auto einAudi;
```

```
einAudi.anzahl
```

Auto::anzahl

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

class Auto
{
public:
    static int anzahl;
    // ...
    Auto()
    {anzahl++;}
    ~Auto()
    {anzahl--;}
};

int Auto::anzahl = 0;

int main(void)
{
    Auto Volkswagen[11];
    cout << "Anzahl Autos: " << Auto::anzahl << endl;
    {
        Auto Audis[15];
        cout << "Anzahl Autos: " << Auto::anzahl << endl;
        {
            Auto meinPeugeot;
```

```

        cout << "Anzahl Autos: " << Auto::anzahl << endl;
    }
    cout << "Anzahl Autos: " << Auto::anzahl << endl;
}
cout << "Anzahl Autos: " << Auto::anzahl << endl;
return 0;
}

```

Ausgabe:

```

Anzahl Autos: 11
Anzahl Autos: 26
Anzahl Autos: 27
Anzahl Autos: 26
Anzahl Autos: 11

```

### Hinweis

Ein statisches Datenelement muss nicht notwendig public sein. Die Definition eines statischen private-Datenelements erfolgt in der gleichen Weise.

## 25.2 Statische Methoden

Was für statische Datenelemente gilt, trifft auch auf statische Elementfunktionen zu: Sie beziehen sich nicht auf einzelne Objekte ihrer Klasse. Folglich bleibt ihnen der Zugriff auf nicht statische Datenelemente verwehrt. Das heißt, eine statische Methode kann nur auf static-Attribute derselben Klasse zugreifen:

```

#include <iostream>
using namespace std;

class Auto
{
private:
    static int anzahl;
    // ...
public:
    Auto()
    {anzahl++;}
    ~Auto()
    {anzahl--;}
    static int getAnzahl()
    {return anzahl;}
    // ...
};

```

```
int Auto::anzahl = 0;

int main(void)
{
    Auto einPeugeot, einVW, einRenault;
    cout << "Anzahl Autos: " << Auto::getAnzahl() << endl;
    return 0;
}
```

Ausgabe: Anzahl Autos: 3

Der Aufruf von statischen Elementfunktionen außerhalb der Klasse kann ebenfalls entweder über den Klassennamen plus Bereichsauflösungsoperator oder mit einem Objekt-namen plus Punktoperator erfolgen. Demnach ließe sich im obigen Beispiel die Anzahl der Autos außer mit

```
Auto::getAnzahl()
```

auch mit

```
einRenault.getAnzahl()
```

oder

```
einVW.getAnzahl()
```

oder

```
einPeugeot.getAnzahl()
```

feststellen. Allerdings kann gesagt werden, dass der Zugriff über den Klassennamen mehr zur Deutlichkeit beiträgt.

Klassenintern kann eine statische Methode von jeder anderen nicht statischen oder statischen ohne qualifizierten Zugriff verwendet werden.

### Hinweis

Nämliches gilt natürlich auch für statische Datenelemente.

Falls eine statische Methode außerhalb ihrer Klasse definiert wird, so ist das Schlüsselwort `static` – ähnlich wie das bei statischen Datenelementen der Fall ist – nur bei der Deklaration innerhalb der Klasse anzugeben:

```
class XY
{
    ...
}
```

```

    static void fun();
    ...
};

void XY::fun()
{
    ...
}

```

Zum Abschluss dieses Kapitels fügen wir der Klasse `Bankkunde` ein statisches Datenelement `count` hinzu, das im Konstruktor der Klasse dazu benutzt wird, Kontonummern automatisch zu vergeben. Aus kosmetischen Gründen sollen die Kontonummern fünfstellig sein. Daher wird `count` mit dem Wert 10000 initialisiert. Die Definition des Konstruktors verlegen wir nach außerhalb der Klasse.

```

class Bankkunde
{
private:
    static int count;
    string vorname;
    string nachname;
    int kto_nr;
    double kto_stand;
    double limit;
public:
    Bankkunde();
    void setVorname(string v)
    {vorname = v;}
    string getVorname()
    {return vorname;}
    void setNachname(string n)
    {nachname = n;}
    string getNachname()
    {return nachname;}
    void setKontonr(int nr)
    {kto_nr = nr;}
    int getKontonr()
    {return kto_nr;}
    void setKontostand(double betrag)
    {kto_stand = betrag;}
    double getKontostand()
    {return kto_stand;}
    void setLimit(double l);
    double getLimit()
    {return limit;}
    void einzahlen(double betrag);
}

```



```

    int auszahlen(double betrag);
};

Bankkunde::Bankkunde()
{
    kto_stand = 0;
    limit = 1000;
    kto_nr = ++count;
}

void Bankkunde::setLimit(double l)
{
    if (l >= 0)
        limit = l;
}

void Bankkunde::einzahlen(double betrag)
{
    if (betrag > 0)
        kto_stand += betrag;
}

int Bankkunde::auszahlen(double betrag)
{
    if (betrag <= 0)
        return 2; // negativer Auszahlungsbetrag
    if (kto_stand - betrag >= -limit)
    {
        kto_stand -= betrag;
        return 0; // Auszahlung ist erfolgt
    }
    else
        return 1; // Auszahlung nicht moeglich
}

int Bankkunde::count = 10000;

```

## Achtung

Beachten Sie, dass für die Definition eines Konstruktors, sofern diese außerhalb der Klasse erfolgt, ebenfalls der qualifizierte Name anzugeben ist, also:

```

Bankkunde::Bankkunde()
{
    ...
}

```

Der Name links vom Bereichsauflösungsoperator ist der Bezeichner für die Klasse, der rechts davon ist der Bezeichner für die Methode.

Hier ein Programm, das die Klasse Bankkunde verwendet:

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

class Bankkunde
{
private:
    static int count;
    string vorname;
    string nachname;
    int kto_nr;
    double kto_stand;
    double limit;
public:
    Bankkunde();
    void setVorname(string v)
    {vorname = v;}
    string getVorname()
    {return vorname;}
    void setNachname(string n)
    {nachname = n;}
    string getNachname()
    {return nachname;}
    void setKontonr(int nr)
    {kto_nr = nr;}
    int getKontonr()
    {return kto_nr;}
    void setKontostand(double betrag)
    {kto_stand = betrag;}
    double getKontostand()
    {return kto_stand;}
    void setLimit(double l);
    double getLimit()
    {return limit;}
    void einzahlen(double betrag);
    int auszahlen(double betrag);
};

Bankkunde::Bankkunde()
{
    kto_stand = 0;
    limit = 1000;
    kto_nr = ++count;
}
```

```

void Bankkunde::setLimit(double l)
{
    if (l >= 0)
        limit = l;
}

void Bankkunde::einzahlen(double betrag)
{
    if (betrag > 0)
        kto_stand += betrag;
}

int Bankkunde::auszahlen(double betrag)
{
    if (betrag <= 0)
        return 2; // negativer Auszahlungsbetrag
    if (kto_stand - betrag >= -limit)
    {
        kto_stand -= betrag;
        return 0; // Auszahlung ist erfolgt
    }
    else
        return 1; // Auszahlung nicht moeglich
}

int Bankkunde::count = 10000;

int main(void)
{
    Bankkunde Kunden[10];
    int index, ktonr, i;
    double betrag;
    char wahl;
    cout << "Deine Kontonummer: ";
    cin >> ktonr;
    index = -1;
    for (i = 0; i < 10; i++)
    {
        if (Kunden[i].getKontonr() == ktonr) {
            index = i; break;
        }
    }
    if (index == -1) {
        cout << "Falsche Kontonummer" << endl;
        return 0;
    }
    cout << fixed << setprecision(2);

```

```

do {
    cout << endl;
    cout << "(K)ontostandsanzeige" << endl;
    cout << "(E)inzahlung" << endl;
    cout << "(A)uszahlung" << endl;
    cout << "E(N)DE" << endl;
    cin >> wahl;
    switch (wahl)
    {
        case 'k':
        case 'K':
            cout << "Dein Kontostand: "
                << Kunden[index].getKontostand()
                << " Euro" << endl;
            break;
        case 'e':
        case 'E':
            cout << "Betrag: ";
            do {
                cin >> betrag;
            } while (betrag < 0);
            Kunden[index].einzahlen(betrag);
            break;
        case 'a':
        case 'A':
            cout << "Betrag: ";
            do {
                cin >> betrag;
            } while (betrag < 0);
            if (Kunden[index].auszahlen(betrag) == 1)
                cout << "Nicht liquide!" << endl;
            else
                cout << "Sollst Du haben!" << endl;
            break;
        case 'n':
        case 'N':
            cout << "Tsch\x81ss" << endl;
            break;
        default:
            cout << "Falsche Eingabe" << endl;
    } // end switch
} while (wahl != 'n' && wahl != 'N');
return 0;
} // end main()

```

Ausgabe:

Deine Kontonummer: 10005

(K)ontostandsanzeige

(E)inzahlung

(A)uszahlung

E(N)DE

k

Dein Kontostand: 0.00 Euro

(K)ontostandsanzeige

(E)inzahlung

(A)uszahlung

E(N)DE

e

Betrag: 3700

(K)ontostandsanzeige

(E)inzahlung

(A)uszahlung

E(N)DE

k

Dein Kontostand: 3700.00 Euro

(K)ontostandsanzeige

(E)inzahlung

(A)uszahlung

E(N)DE

a

Betrag: 3999.97

Sollst Du haben!

(K)ontostandsanzeige

(E)inzahlung

(A)uszahlung

E(N)DE

k

Dein Kontostand: -299.97 Euro

(K)ontostandsanzeige

(E)inzahlung

(A)uszahlung

E(N)DE

n

Tschüss

## CD-ROM

Quellcode und ausführbare Datei finden Sie als *k25c.cpp* und *k25c.exe* im Ordner *Beispiele/K25* auf der Buch-CD.

Im Programm werden mit der Anweisung

```
Bankkunde Kunden[10];
```

zehn Bankkunden angelegt. Somit erstrecken sich die Kontonummern, die ja nun für jedes Bankkunde-Objekt mit Ausführung dessen Konstruktormethode – also automatisch – vergeben werden, entsprechend der Implementation der Klasse von 10001 bis 10010. In folgendem Codeteil wird geprüft, ob ein zuvor eingegebener und in der Variablen *ktionr* gespeicherter Wert als Kontonummer vorhanden ist:

```
index = -1;
for (i = 0; i < 10; i++)
{
    if (Kunden[i].getKontonr() == ktonr) {
        index = i; break;
    }
}
if (index == -1) {
    cout << "Falsche Kontonummer";
    return 0;
}
```

Ist das der Fall, dann erhält die Variable *index* in der *for*-Schleife den Index des betreffenden Kunden. Dieser kann dann im weiteren Code mit

```
Kunden[index]
```

angesprochen werden. Ist eine entsprechende Kontonummer nicht vorhanden, wird der Wert von *index* in der *for*-Schleife nicht verändert. Die Variable besitzt dann genau den Wert, der ihr vor Eintritt in die *for*-Schleife zugewiesen wurde, also -1. Auf diesen Wert kann *index* nach Austritt aus der *for*-Schleife geprüft werden. Ist die Bedingung

```
index == -1
```

wahr, so wird die Funktion *main()* und damit das Programm unverzüglich beendet.

## Hinweis

Natürlich hätte ein anderer Wert als -1 den gleichen Zweck erfüllt. Allerdings muss sichergestellt sein, dass der Anfangswert von *index* nicht als Kontonummer vorkommt. Daher erscheint ein negativer Wert am besten geeignet.

# 26 Dateioperationen

Häufig ist es wünschenswert, Daten nicht nur temporär im Arbeitsspeicher, sondern permanent zu speichern, sei es für spätere Programmläufe oder für andere Programme. Zu diesem Zweck stellt C++ verschiedene Klassen zur Verfügung, von denen wir hier die Klasse `ifstream` verwenden, um aus Textdateien zu lesen, und die Klasse `ofstream`, um in Textdateien zu schreiben.

## 26.1 In Textdateien schreiben

Für die genannten Klassen ist die Headerdatei *fstream* einzubinden:

```
#include <fstream>
```

Um in eine Datei schreiben zu können, kann man eine Instanz der Klasse `ofstream` bilden, z.B.

```
ofstream dataus;
```

Als Nächstes muss die Datei geöffnet werden, was nichts anderes heißt, als dass eine Verbindung zwischen Stream und Datei herzustellen ist. Dazu ruft man die `open()`-Methode des `ofstream`-Objekts auf und übergibt dieser den Namen der zu öffnenden Datei, gegebenenfalls mit Pfadangabe. Falls die Datei nicht existiert, wird sie automatisch angelegt.

```
dataus.open("C:\\Borland\\MeineProgramme\\eineDatei.txt");
```

### Achtung

Denken Sie daran, dass einem auszugebenden Backslash in einer Zeichenkette ein weiterer vorangestellt werden muss (siehe dazu Kapitel 9, Abschnitt 9.1 »Die Escape-Sequenzen `\"`, `'` und `\\«`).

Allerdings ist auch der Konstruktor der `ofstream`-Klasse in der Lage, die Zeichenkette zu übernehmen, sodass das Öffnen einer Datei bereits beim Instanzieren erfolgen kann:

```
ofstream dataus("C:\\Borland\\MeineProgramme\\eineDatei.txt");
```

## Hinweis

Wie Sie daraus schließen können, ist der Konstruktor der `ofstream`-Klasse entsprechend überladen.

Nachdem die Datei nun geöffnet, das heißt eine Verbindung zwischen dieser und dem Streamobjekt hergestellt ist, können die Ausgabeoperationen mit den bekannten Methoden durchgeführt werden.

## Streamobjekte

Wie Sie wissen, sind `cout` und `cin` ebenfalls Streamobjekte, wobei `cout` eine Instanz der Klasse `ostream` und `cin` eine der Klasse `istream` ist. Beide Objekte sind in der Headerdatei `iostream` instanziiert, sodass man sie nach Einbindung von `iostream` mit der `#include`-Direktive einfach verwenden kann. Im Unterschied dazu müssen Objekte der Streamklassen `ofstream` und `ifstream` vom Programmierer im Code erst erzeugt werden.

Was den Umgang mit Objekten der Streamklassen sehr angenehm macht, ist die Tatsache, dass in den Ausgabestreamklassen hinsichtlich Namen und Verwendung weitgehend dieselben Methoden definiert sind. Das Gleiche gilt auch für die Streamklassen zur Dateneingabe. Beispielsweise können Sie nun mit dem oben vereinbarten Objekt `dataus` in der gewohnten Weise in die geöffnete Datei schreiben, indem Sie dem Ausgabeoperator `<<` die gewünschten Daten einfach übergeben:

```
dataus << "In eine Textdatei schreiben";
```

Der einzige Unterschied besteht, wie gesagt, darin, dass die Ausgabe nicht mit dem Objekt `cout`, sondern mit einem Objekt der Klasse `ofstream` erfolgt. Dies hat zur Folge, dass die Daten nicht auf den Bildschirm, sondern in eine Datei ausgegeben werden.

Folgendes Programm legt im Verzeichnis `C:\Borland\MeineProgramme`, das bereits vorhanden sein muss, eine Textdatei `eineDatei.txt` an und schreibt in diese die Sätze »In eine Textdatei schreiben.« und »Noch ein Satz.« (Abbildung 26.1).

## Achtung

Falls die Datei bereits existiert, wird der bestehende Inhalt überschrieben.

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

int main(void)
{
    ofstream dataus;
    dataus.open("C:\\Borland\\MeineProgramme\\eineDatei.txt");
```



```

dataus << "In eine Textdatei schreiben." << endl
      << "Noch ein Satz.";
dataus.close();
return 0;
}

```

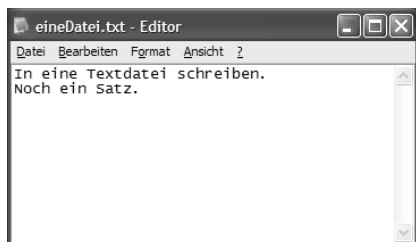


Abbildung 26.1: Ausgabe in die Textdatei eineDatei.txt

Die Datei wird geschlossen, das heißt die Verbindung zwischen dieser und dem Streamobjekt gelöst, wenn das Streamobjekt seine Gültigkeit verliert. Dies ist mit dem Aufruf der `close()`-Methode des `ofstream`-Objekts der Fall oder mit Ende des Programms.

### Hinweis

Das bedeutet, dass der Destruktor des `ofstream`-Objekts nötigenfalls dafür sorgt, dass die Datei geschlossen wird. Was nichts anderes heißt, als dass im Rumpf der Destruktormethode geprüft wird, ob die Datei noch geöffnet ist und für diesen Fall die `close()`-Methode aufgerufen wird.

Folglich könnte im Beispiel auf das explizite Schließen der Datei mit der `close()`-Methode verzichtet werden. Trotzdem sollte man die Datei auf jeden Fall selbst schließen – dies ist nicht nur die korrekte Art und Weise, es führt auch zu einer besseren Lesbarkeit des Codes. Es wird damit dokumentiert, dass die Schreiboperation an dieser Stelle beendet ist.

Wie Ihnen bekannt ist, können Sie dem Ausgabeoperator jeden beliebigen elementaren Datentyp sowie `string` übergeben:

```

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main(void)
{
    ofstream dataus;
    string ist = " = ";
    dataus.open("C:\\Borland\\MeineProgramme\\eineDatei.txt");
}

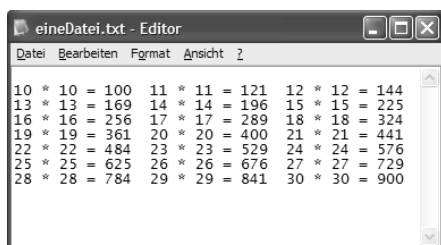
```

```

for (int i = 10; i <= 30; i++)
{
    if ((i-10) % 3 == 0)
        dataus << endl;
    dataus << i << " * " << i << ist << i * i << " ";
}
dataus.close();
return 0;
}

```

Das Ergebnis des obigen Programms ist Abbildung 26.2 zu entnehmen.



**Abbildung 26.2:** Ausgabe von Werten unterschiedlichen Datentyps

Wie oben bereits erwähnt und wie Sie am Inhalt der Textdatei nunmehr erkennen können, wurde die vorige Ausgabe überschrieben. Falls Sie dies nicht wünschen, geben Sie beim Öffnen der Datei als zweiten Parameter `ios_base::app` an (für »append«, zu Deutsch »anhängen«):

```

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

int main(void)
{
    ofstream dataus;
    dataus.open("C:\\text.txt", ios_base::app);
    dataus << '*';
    dataus.close();
    return 0;
}

```

Obiges Programm ergänzt den Inhalt der Datei *text.txt* mit jeder Ausführung um ein Sternchen.

### Hinweis

Bei weniger aktuellen Compilern kann es sein, dass die Konstante `app` in der Klasse `ios` definiert ist. In diesem Fall müssen Sie anstelle von `ios_base::app` als Argument `ios::app` angeben:

```
dataus.open("C:\\text.txt", ios::app);
```

Falls dem Benutzer die Möglichkeit gegeben werden soll, Ort und Namen der Datei zu bestimmen, in die geschrieben werden soll, ist zu beachten, dass der Konstruktor der `ofstream`-Klasse einen C-String erwartet. Dies gilt im Übrigen auch für die Klasse `ifstream`, die für das Öffnen einer Datei zum Lesen zuständig ist (dazu mehr im nächsten Abschnitt). Folglich muss ein `string`-Wert gegebenenfalls mit der `c_str()`-Methode der `string`-Klasse erst in einen C-String umgewandelt werden:

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

int main(void)
{
    ofstream dataus;
    string datei;
    cout << "In welche Datei soll geschrieben werden: ";
    cin >> datei;
    dataus.open(datei.c_str(), ios_base::app);
    dataus << "ABC";
    dataus.close();
    return 0;
}
```

Ausgabe:

```
In welche Datei soll geschrieben werden: C:\text.txt
```

### Achtung

Vom Benutzer ist der Backslash natürlich nur ein Mal einzutippen.

Mit Ausführung wird der Inhalt der vom Benutzer angegebenen Datei um die Zeichenfolge `ABC` ergänzt.

## Referenz

Zur Methode `c_str()` der `string`-Klasse siehe Kapitel 19, Abschnitt 19.4.1 »Die Methode `c_str()`«.

Falls beim Öffnen einer Datei kein Pfad angegeben ist, wird diese im Programmverzeichnis erwartet bzw. angelegt, also in demjenigen Verzeichnis, in dem sich die `.exe`-Datei befindet.

## Achtung

Das gilt immer, insofern Sie die `.exe`-Datei eigenständig ausführen, also z.B. über die Konsole oder im Windows-Explorer. Anders verhält es sich zum Beispiel, wenn Sie das Programm direkt von der Oberfläche des Visual Studios aus starten. In diesem Fall wird die Textdatei in dem Verzeichnis erwartet bzw. angelegt, in dem sich die Quelldateien befinden. Falls Sie das Programm später von der Konsole aus ausführen wollen, ist es gegebenenfalls anzuraten, eine Kopie der Textdatei in das Programmverzeichnis zu legen (Unterverzeichnis *Debug* eines Projekts).

Hier ein Programm, das dem Benutzer gestattet, nicht nur den Dateinamen für die Ausgabe festzulegen, sondern auch den zu schreibenden Inhalt zu bestimmen:

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

int main(void)
{
    ofstream dataus;
    string datei;
    char zeichen;
    int um;
    cout << "In welche Datei soll geschrieben werden: ";
    cin >> datei;
    cin.get();
    dataus.open(datei.c_str(), ios_base::app);
    um = 0;
    do
    {
        um++;
        zeichen = cin.get();
        dataus.put(zeichen);
        if (um % 80 == 0)
            dataus << endl;
    } while (zeichen != '\n');
```

```
dataus.close();
return 0;
}
```

## Referenz

Zur Methode `get()` des `cin`-Objekts siehe Kapitel 19, Abschnitt 19.2.1 »`cin.get()`«. Zur Methode `put()` des `cout`-Objekts siehe im gleichen Kapitel Abschnitt 19.2.2 »`cout.put()`«. Die in den Klassen `ifstream` bzw. `ofstream` definierten gleichnamigen Methoden lassen sich für Objekte dieser Klassen in der gleichen Weise verwenden: `put()` für ein `ofstream`-Objekt (wie am obigen Beispiel zu sehen) und `get()` für ein `ifstream`-Objekt (siehe dazu das folgende Beispiel im nächsten Abschnitt).

Die Eingabe des Benutzers wird vom Programm zeilenweise in die Datei geschrieben, wobei die `if`-Bedingung

```
if (um % 80 == 0)
    dataus << endl;
```

jeweils nach 80 Zeichen in der Datei für einen Zeilenvorschub sorgt. Die Anweisung

```
cin.get();
```

nach Eingabe der gewünschten Datei (`cin >> datei;`) ist notwendig, um das verbleibende Newline-Zeichen aus dem Eingabepuffer zu entfernen. Ansonsten würde mit der ersten Anweisung

```
zeichen = cin.get();
```

innerhalb der `do while`-Schleife die Variable `zeichen` dieses Newline-Zeichen erhalten und die Schleife entsprechend der Bedingung (`zeichen != '\n'`) sofort abbrechen.

## 26.2 Aus Textdateien lesen

Analog zur Klasse `ofstream` ermöglicht es die Klasse `ifstream`, Daten aus einer Datei zu lesen. Dazu ist wiederum ein Objekt dieser Klasse zu instanzieren. Danach können Sie dieses Objekt ebenso benutzen, wie Sie es von `cin` gewohnt sind.

Folgendes Programm liest den Inhalt einer vom Benutzer anzugebenden Datei ein und gibt ihn auf den Bildschirm aus:

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

```

int main(void)
{
    ifstream datein;
    string datei;
    char zeichen;
    cout << "Welche Datei soll gelesen werden: ";
    cin >> datei;
    datein.open(datei.c_str());

    if ( ((bool)datein) == false)
    {
        cerr << "Datei konnte nicht ge\x94"
              << "ffnet werden" << endl;
        return 0;
    }

    while (datein.eof() != true)
    {
        zeichen = datein.get();
        cout << zeichen;
    }

    datein.close();
    return 0;
}

```

In Bezug auf die `close()`-Methode der `ifstream`-Klasse gilt das oben zur entsprechenden Methode der `ofstream`-Klasse Gesagte.

Ein Streamobjekt liefert in einem logischen Ausdruck den Wert `true` zurück, falls eine vorangegangene Operation erfolgreich war, andernfalls `false`. Das ist sowohl für das Öffnen einer Datei als auch für das Schreiben in eine oder das Lesen aus einer Datei zutreffend.

### Hinweis

Ähnliches gilt im Übrigen auch für die Objekte der `ostream`- bzw. der `istream`-Klasse. So lässt sich das `cin`-Objekt in einer Schleifenbedingung etwa wie folgt verwenden:

```

double sum = 0, zahl;
while (cin >> zahl)
    sum += zahl;

```

Die Schleifenbedingung `(cin >> zahl)` wertet zu `false` aus, falls die Zuweisung eines vom Benutzer eingegebenen Wertes an die Variable `zahl` nicht durchgeführt werden konnte, z.B. weil die Eingabe des Benutzers nicht als numerischer Wert interpretierbar ist.

Folglich gibt der Ausdruck

```
((bool)datein) == false)1
```

unmittelbar nach dem Aufruf der `open()`-Methode Auskunft darüber, ob die Datei geöffnet werden konnte.

### Hinweis

Statt

```
if ((bool)datein) == false)
```

hätten wir auch kürzer

```
if (!datein)
```

formulieren können (siehe dazu Kapitel 16, Abschnitt 16.2.2 »Konvertierung in logischen Ausdrücken«).

Falls die Datei z.B. nicht existiert, würde die obige `if`-Bedingung zu `true` auswerten. In diesem Fall wird mit

```
cerr << "Datei konnte nicht ge\x94"
      << "ffnet werden" << endl;
return 0;
```

eine entsprechende Fehlermeldung ausgegeben und das Programm unverzüglich beendet.

### Hinweis

Im Gegensatz zu `cout`, das die auszugebenden Daten an die Standardausgabe leitet, sendet das Objekt `cerr` die Daten an die Standardfehlerausgabe – was in der Regel ebenfalls der Bildschirm ist. Von daher hätte man für die Meldung, dass die Datei nicht geöffnet werden konnte, wie gewohnt auf `cout` zurückgreifen können. Da das Objekt `cerr` – wie `cout` eine Instanz der Klasse `ostream` – jedoch für die Ausgabe von Fehlermeldungen vorgesehen ist, sei dem Leser empfohlen, gegebenenfalls dieses zu verwenden.

Die Methode `eof()` der `ifstream`-Klasse liefert `true` zurück, wenn das Dateiende erreicht ist. Sie wird daher in der Bedingung der `while`-Schleife zur Prüfung herangezogen. Innerhalb der `while`-Schleife wird der Inhalt der Datei mit der `get()`-Methode des `ifstream`-Objekts `datein` zeichenweise eingelesen und auf den Bildschirm ausgegeben:

```
zeichen = datein.get();
cout << zeichen;
```

<sup>1</sup> Die explizite Typumwandlung ist je nach verwendetem Compiler nicht zwingend erforderlich.

**Hinweis**

Dem weiter oben Gesagten zufolge hätte man in der Schleifenbedingung auch das Streamobjekt selbst prüfen können:

```
while ( ((bool)datein) == true)
```

bzw.

```
while (datein)
```

Allerdings ist der Rückgabewert der eof()-Methode spezieller und damit genauer.

Beim Einlesen von Daten aus einer Datei kann sich der Umstand als nützlich erweisen, dass Zwischenraumzeichen in der Datei in der gleichen Weise interpretiert werden wie bei der Dateneingabe auf der Konsole. Wenn man beispielsweise weiß, dass eine Datei elf Datensätze enthält, von denen jeder, sagen wir, aus einer Stückzahl und – getrennt durch mindestens ein Leerzeichen – einem Artikelnamen besteht, dann können diese vom Programm auf einfache Weise in Variablen eingelesen werden:

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

int main(void)
{
    ifstream datein;
    int anzahl[11], i;
    string artikel[11];
    datein.open("Warenbestand.txt");

    if (!datein)
    {
        cerr << "Datei konnte nicht ge\x94"
              << "ffnet werden" << endl;
        return 0;
    }

    for (i = 0; i < 11; i++)
    {
        datein >> anzahl[i];
        datein >> artikel[i];
    }
}
```



```

for (i = 0; i < 11; i++)
{
    cout << anzahl[i] << ' ';
    cout << artikel[i] << endl;
}

datein.close();
return 0;
}

```

## CD-ROM

Die Datei *Warenbestand.txt* finden Sie zusammen mit der Quellcodedatei *k26g.cpp* auf der Buch-CD im Ordner *Beispiele/K26*.

Obiges Programm liest den Inhalt der Datei *Warenbestand.txt* (Abbildung 26.3) in die Arrays *anzahl* und *artikel* ein und gibt die Werte im Anschluss daran zur Kontrolle auf den Bildschirm aus.



Abbildung 26.3: Datensätze aus Textdatei lesen

Die Bildschirmausgabe können Sie Abbildung 26.4 entnehmen.



Abbildung 26.4: Ausgabe der eingelesenen Werte auf den Bildschirm

### Textdateien im Visual Studio verwalten

Falls Sie mit Visual Studio oder der Visual C++ 2005 Express Edition arbeiten, empfiehlt es sich, eine bestehende Textdatei zunächst in das Projektverzeichnis – gemeint ist das Verzeichnis, in dem sich die Quelldatei(en) befindet (befinden) – zu legen und diese anschließend mit dem Menübefehl **PROJEKT/VORHANDENES ELEMENT HINZUFÜGEN...** dem Projekt hinzuzufügen. Stellen Sie im folgenden Dialog als Dateityp **ALLE DATEIEN (\*.\*)** ein, damit Sie die Textdatei auswählen können.

Alternativ können Sie mit dem Menübefehl **DATEI/NEU/DATEI...** dem aktuellen Projekt auch eine neue – leere – Textdatei hinzufügen. Wählen Sie dazu im anschließenden »Neue Datei«-Dialog in der Kategorie **ALLGEMEIN** als Vorlage **TEXTDATEI**.

Denken Sie daran, dass Sie auf jeden Fall eine Textdatei in das Projektverzeichnis legen müssen. Damit Sie das Programm später auch von der Konsole aus starten können, ist es zudem sinnvoll, eine aktuelle Kopie der Textdatei in das Programmverzeichnis zu legen, also in das Verzeichnis, in dem sich die *.exe*-Datei befindet (Verzeichnisname *Debug*).

## 26.2.1 Programm zum Verwalten von (Entenhausener) Bankkunden

Das folgende Programm liest Kontonummer, Vor- und Nachname, Dispolimit und Kontostand von zehn Bankkunden aus der Textdatei *Kunden.txt* und schreibt nach Ausführung von Kundenaktionen die gegebenenfalls aktualisierten Werte in dieselbe Datei. Eine Initialisierung der entsprechenden Werte im Konstruktor der Bankkunde-Klasse entfällt damit.

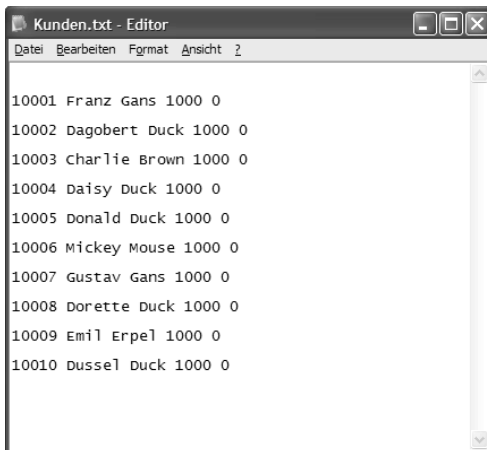


Abbildung 26.5: Textdatei Kunden.txt

Gegenüber der Version des letzten Kapitels wurde zudem eine weitere Klasse *Bank* definiert, welche die zehn Kunden enthält. Diese Klasse wurde außerdem mit einer Methode

zum Suchen von Kunden, einer Methode ueberweisen() und mit Methoden zum Lesen und Schreiben aus bzw. in die Datei *Kunden.txt* versehen.

Um das folgende Programm korrekt ausführen zu können, ist es notwendig, die obige Textdatei *Kunden.txt* in das Programmverzeichnis zu legen (bzw. in das Projektverzeichnis, in dem sich die Quelldatei befindet, falls Sie das Programm aus Visual Studio heraus ausführen). Sie finden *Kunden.txt* – ebenso wie *.exe*- und Quelldatei – in *Bank\_K26.zip* auf der Buch-CD im Ordner *Beispiele/K26*.

```
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
using namespace std;

class Bankkunde
{
private:
    string vorname;
    string nachname;
    int kto_nr;
    double kto_stand;
    double limit;
public:
    Bankkunde() {}
    void setVorname(string v)
    {vorname = v;}
    string getVorname()
    {return vorname;}
    void setNachname(string n)
    {nachname = n;}
    string getNachname()
    {return nachname;}
    void setKontonr(int nr)
    {kto_nr = nr;}
    int getKontonr()
    {return kto_nr;}
    void setKontostand(double betrag)
    {kto_stand = betrag;}
    double getKontostand()
    {return kto_stand;}
    void setLimit(double l);
    double getLimit()
    {return limit;}
    void einzahlen(double betrag);
    int auszahlen(double betrag);
};
```

```
void Bankkunde::setLimit(double l)
{
    if (l >= 0)
        limit = l;
}

void Bankkunde::einzahlen(double betrag)
{
    if (betrag > 0)
        kto_stand += betrag;
}

int Bankkunde::auszahlen(double betrag)
{
    if (betrag <= 0)
        return 2; // negativer Auszahlungsbetrag
    if (kto_stand - betrag >= -limit)
    {
        kto_stand -= betrag;
        return 0; // Auszahlung ist erfolgt
    }
    else
        return 1; // Auszahlung nicht moeglich
}

class Bank
{
public:
    Bankkunde Kunden[10];
    int KundeSuchen(int nr);
    int ueberweisen(int auftr, int empf, double betrag);
    void KundenEinlesen();
    void KundenInDatei();
};

int Bank::KundeSuchen(int nr)
// Rueckgabewert ist -1, falls Konto nicht vorhanden,
// ansonsten der Index des Kunden
{
    for (int i = 0; i < 10; i++)
    {
        if (Kunden[i].getKontonr() == nr)
            return i;
    }
    return -1;
}
```

```
int Bank::ueberweisen(int auftr, int empf, double betrag)
{
    if (betrag <= 0)
        return -1; // Programmierfehler
    if (Kunden[auftr].auszahlen(betrag) == 0) {
        Kunden[empf].einzahlen(betrag);
        return 0; // Ueberweisung ausgefuehrt
    }
    else
        return 1; // Konto nicht gedeckt
}

void Bank::KundenEinlesen()
{
    int hilf;
    string name;
    double help;
    ifstream lesen("Kunden.txt");
    for (int i = 0; i < 10; i++)
    {
        lesen >> hilf; Kunden[i].setKontonr(hilf);
        lesen >> name; Kunden[i].setVorname(name);
        lesen >> name; Kunden[i].setNachname(name);
        lesen >> help; Kunden[i].setLimit(help);
        lesen >> help; Kunden[i].setKontostand(help);
    }
    lesen.close();
}

void Bank::KundenInDatei()
{
    int hilf;
    string name;
    double help;
    ofstream schreiben("Kunden.txt");
    for (int i = 0; i < 10; i++)
    {
        hilf = Kunden[i].getKontonr();
        schreiben << endl << endl << hilf << ' ';
        name = Kunden[i].getVorname();
        schreiben << name << ' ';
        name = Kunden[i].getNachname();
        schreiben << name << ' ';
        help = Kunden[i].getLimit();
        schreiben << help << ' ';
        help = Kunden[i].getKontostand();
        schreiben << help;
    }
}
```

```

int main(void)
{
    Bank Entenhausener;
    Entenhausener.KundenEinlesen();
    int index, index_empf, ktonr, ktonr_empf;
    double betrag;
    char wahl;
    cout << "Deine Kontonummer: ";
    cin >> ktonr;
    index = Entenhausener.KundeSuchen(ktonr);
    if (index == -1) {
        cout << "Falsche Kontonummer" << endl;
        return 0;
    }
    cout << fixed << setprecision(2);
    cout << "Hallo "
        << Entenhausener.Kunden[index].getVorname() << ' '
        << Entenhausener.Kunden[index].getNachname()
        << endl;
    do {
        cout << endl;
        cout << "(K)ontostandsanzeige" << endl;
        cout << "(E)inzahlung" << endl;
        cout << "(A)uszahlung" << endl;
        cout << "(U)eberweisung" << endl;
        cout << "E(N)DE" << endl;
        cin >> wahl;
        switch (wahl)
        {
            case 'k':
            case 'K':
                cout << "Dein Kontostand: "
                    << Entenhausener.Kunden[index].getKontostand()
                    << " Euro" << endl;
                break;
            case 'e':
            case 'E':
                cout << "Betrag: ";
                do {
                    cin >> betrag;
                } while (betrag < 0);
                Entenhausener.Kunden[index].einzahlen(betrag);
                break;
        }
    }
}

```

```
case 'a':
case 'A':
    cout << "Betrag: ";
    do {
        cin >> betrag;
    } while (betrag < 0);
    if (Entenhausener.Kunden[index].auszahlen(betrag) == 1)
        cout << "Nicht liquide!" << endl;
    else
        cout << "Sollst Du haben!" << endl;
    break;
case 'u':
case 'U':
    cout << "Kontonummer des Empf\x84ngers: ";
    cin >> ktonr_empf;
    index_empf = Entenhausener.KundeSuchen(ktonr_empf);
    if (index_empf == -1) {
        cout << "Falsche Kontonummer" << endl;
        break;
    }
    cout << "An "
    << Entenhausener.Kunden[index_empf].getVorname()
    << ' ' <<
    Entenhausener.Kunden[index_empf].getNachname();
    cout << '!' << endl << "Betrag: ";
    do {
        cin >> betrag;
    } while (betrag < 0);
    if (Entenhausener.ueberweisen
        (index, index_empf, betrag) != 0)
        cout << '\x9A' << "berweisung konnte nicht"
        << " ausgef\x81hrt werden" << endl;
    break;
case 'n':
case 'N':
    cout << "Tsch\x81ss" << endl;
    break;
default:
    cout << "Falsche Eingabe" << endl;
} // end switch
} while (wahl != 'n' && wahl != 'N');
Entenhausener.KundenInDatei();
return 0;
} // end main()
```

Ausgabe:

```
Deine Kontonummer: 10003
Hallo Charlie Brown

(K)ontostandsanzeige
(E)inzahlung
(A)uszahlung
(U)eberweisung
E(N)DE
e
Betrag: 100.90

(K)ontostandsanzeige
(E)inzahlung
(A)uszahlung
(U)eberweisung
E(N)DE
u
Kontonummer des Empfängers: 10006
An Mickey Mouse!
Betrag: 50.45

(K)ontostandsanzeige
(E)inzahlung
(A)uszahlung
(U)eberweisung
E(N)DE
n
Tschüss
```

Das Ergebnis dieser Aktionen können Sie Abbildung 26.6 entnehmen. Der letzte Wert in einer Zeile bezieht sich jeweils auf den Kontostand.



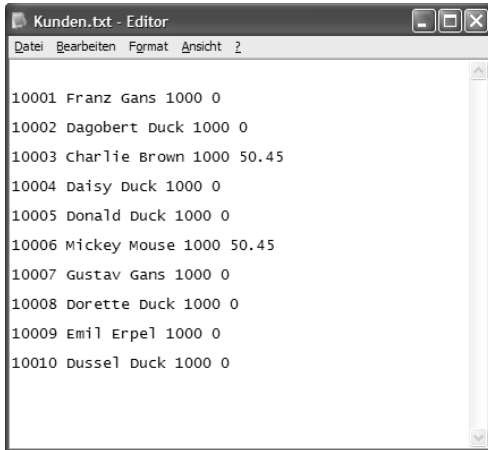


Abbildung 26.6: Nach oben beschriebener Programmausführung

### CD-ROM

Textdatei, .exe-Datei und Quellcodedatei des Beispiels finden Sie gepackt zu *Bank\_K26.zip* auf der Buch-CD im Ordner *Beispiele/K26*.



# Teil IV

## Fortgeschrittene Programmierkonzepte

In diesem letzten Teil sollen einige fortgeschrittene Konzepte vorgestellt werden. Speziell in den Bereich der objektorientierten Programmierung fallen darunter Vererbung sowie das Überladen von Operatoren. Zentrales Thema dieses Teils wird jedoch der Gebrauch von Zeigern und Referenzen sein.



# 27

## Präprozessor-Direktiven

Im Zuge des Kompiliervorgangs wird der Quelltext vor der eigentlichen Übersetzung in Maschinencode zunächst von einer speziellen Software<sup>1</sup> überarbeitet, dem so genannten Präprozessor. Auf dessen Verhalten kann der Programmierer ebenfalls Einfluss nehmen.

Anweisungen an den Präprozessor werden im Allgemeinen Direktiven genannt, um den Unterschied zu den Anweisungen deutlich zu machen, die an den Compiler gerichtet sind. Alle Präprozessor-Direktiven beginnen mit einem Doppelkreuz (#) und werden im Gegensatz zu gewöhnlichen Anweisungen nicht mit einem Semikolon abgeschlossen. Beides ist Ihnen ja bereits von der `#include`-Anweisung her bekannt (nach dem eben Gesagten sollte man sie wohl `#include`-Direktive nennen). In diesem Kapitel sollen die wichtigsten Präprozessor-Direktiven besprochen werden. Grundsätzlich dürfen diese an jeder Stelle im Code verwendet werden. In der Regel wird dies jedoch am Anfang einer Quelldatei geschehen.

### 27.1 `#include`

Wie Ihnen ja bereits bekannt ist, wird mit Ausführung der `#include`-Direktive die entsprechende Zeile im Code mit dem Inhalt einer angegebenen Datei ersetzt. Dabei kann es sich zwar grundsätzlich um jede beliebige Datei handeln, solange diese aus gültigen C++-Anweisungen besteht (andernfalls wird dies natürlich Fehlermeldungen des Compilers zur Folge haben). In der Regel wird man die `#include`-Direktive jedoch dazu verwenden, Headerdateien, die vom Compilersystem zur Verfügung gestellt werden, oder solche, die vom Programmierer selbst erstellt wurden, einzubinden. Erstere sind in spitzen Klammern, Letztere in doppelten Anführungszeichen anzugeben.

Die Anweisung

```
#include <iostream>
```

ersetzt also diese Zeile mit dem Inhalt der Datei *iostream*. Das Gleiche gilt für die Anweisung

```
#include "myfunctions.h"
```

nur dass in diesem Fall die selbst definierte Headerdatei *myfunctions.h* eingebunden wird. Der Unterschied liegt darin, dass Dateien, die in spitzen Klammern angegeben sind, im Verzeichnis *Include* der Compilerinstallation gesucht werden, Dateien, die in

---

1 Moderne Compiler verfügen in der Regel über einen integrierten Präprozessor

doppelten Anführungszeichen stehen, dagegen in dem Verzeichnis, in dem sich die Quelldatei befindet.

## 27.2 Symbolische Konstanten mit #define vereinbaren

Außer mit dem Schlüsselwort `const` lassen sich symbolische Konstanten auch mit der `#define`-Direktive definieren, wobei zunächst der Name der Konstanten und danach deren Wert anzugeben ist.

### Referenz

Zu `const`-Konstanten siehe Kapitel 12, Abschnitt 12.4 »Konstanten mit `const` deklarieren«.

Beispielsweise legt man mit

```
#define PI 3.14
```

eine Konstante `PI` mit dem Wert `3.14` fest. Ebenso mit

```
#define MWST 19
```

eine Konstante `MWST` mit dem Wert `19`. Somit lässt sich etwa das letzte Beispiel in Kapitel 12 ebenso unter Verwendung einer `#define`- anstelle einer `const`-Konstanten realisieren:

```
#include <iostream>
#define MWST 0.19
using namespace std;

int main(void)
{
    double netto;
    cout << "Nettobetrag: ";
    cin >> netto;
    cout << "Mwst: " << netto * MWST;
    cout << endl << "Gesamtbetrag: ";
    cout << netto + (netto * MWST) << endl;
    return 0;
}
```

bzw.

```
#include <iostream>
#define MWST 19
using namespace std;

int main(void)
{
    double netto;
    cout << "Nettobetrag: ";
    cin >> netto;
    cout << "Mwst: " << netto * MWST / 100;
    cout << endl << "Gesambetrag: ";
    cout << netto + (netto * MWST / 100) << endl;
    return 0;
}
```

Ausgabe:

```
Nettobetrag: 300
Mwst: 57
Gesambetrag: 357
```

Die Verwendung von mit `#define` definierten Konstanten erfolgt unter den gleichen Gesichtspunkten wie `const`-Konstanten. Allerdings handelt es sich bei Ersteren um reine Textersetzungen seitens des Präprozessors. Im Zweifel sind daher `const`-Konstanten wegen der damit verbundenen Typüberprüfung vorzuziehen.

Anders als das bei `const`-Konstanten der Fall ist, können mit `#define` auch Konstanten ohne Ersatztext definiert werden:

```
#define KONSTANTE
```

Im Ergebnis wird dann die Konstante bzw. deren Name an jeder Stelle, an der sie im nachfolgenden Code auftritt, ersatzlos entfernt. Ungeachtet dessen gilt die Konstante als definiert, was im Hinblick auf eine bedingte Kompilierung von Bedeutung sein kann (dazu mehr im nächsten Abschnitt).

## 27.3 Bedingte Kompilierung

Es ist möglich, Codeteile bedingungsabhängig kompilieren zu lassen. Das heißt, bestimmte C++-Anweisungen werden nur für den Fall übersetzt, dass eine bestimmte Bedingung wahr ist. Zur Umsetzung der bedingten Kompilierung können die Präprozessor-Direktiven `#if`, `#else`, `#elif`, `#endif` sowie – zur Formulierung einer entsprechenden Bedingung – `#ifdef` und `#ifndef` in Verbindung mit der `#define`-Direktive eingesetzt werden. Wobei `#if`, `#else` die Pendanten zu den C++-Schlüsselwörtern `if` und `else` sind. Das Gleiche gilt für `#elif` in Bezug auf die Kombination `else if`.

Dem Gesagten zufolge können Sie mit einer entsprechenden Konstruktion Ihren Compiler daran hindern, bestimmte Codeteile zu übersetzen. Genau genommen sorgt der Präprozessor dann dafür, dass der Compiler diese gar nicht erhält. Hierzu ein Beispiel:

```
#include <iostream>
#define WELCHER 'B'
using namespace std;

int main(void)
{
    // ...
    #if WELCHER == 'A'
        cout << "Teil A wird";
        cout << " kompiliert";
    #elif WELCHER == 'B'
        cout << "Teil B wird";
        cout << " kompiliert";
    #elif WELCHER == 'C'
        cout << "Teil C wird";
        cout << " kompiliert";
    #else
        cout << "Teil D wird";
        cout << " kompiliert";
    #endif
    // ...
    return 0;
}
```

Ausgabe:

```
Teil B wird kompiliert
```

### Achtung

Im Gegensatz zu `if`- bzw. `if else`-Anweisungen müssen die entsprechenden Präprozessor-Direktiven mit `#endif` abgeschlossen werden.

Natürlich hätte man für die Konstante `WELCHER` ebenso gut numerische Werte heranziehen können. Beachten Sie, dass der `#else`-Teil auch für den Fall kompiliert wird, dass gar keine Konstante `WELCHER` definiert ist.

Wie bei der `else if`-Konstruktion kann der `#else`-Zweig auch weggelassen werden. In diesem Fall wird keiner der Codeteile übersetzt, falls die Konstante `WELCHER` einen Wert ungleich `'A'`, `'B'` oder `'C'` besitzt bzw. eine Konstante dieses Namens überhaupt nicht definiert ist.



## Referenz

Zur `else if`-Konstruktion siehe Kapitel 16, Abschnitt 16.2.5 »else if«.

Wie Sie sich vermutlich denken können, kann die bedingte Kompilierung vor allem in der Entstehungsphase eines Programms hilfreich sein. Man muss dann nur den Ersetzungswert in der `#define`-Direktive entsprechend ändern, um einen anderen Teil übersetzen zu lassen. In der Praxis werden die verschiedenen Programmteile, die die C++-Anweisungen enthalten, entsprechend umfangreich sein, während diese im obigen Beispiel allein den Zweck der Demonstration erfüllen.

Eine weitere Möglichkeit besteht darin, mit den Direktiven `#ifdef` (»if defined«) bzw. `#ifndef` (»if not defined«) zu prüfen, ob eine bestimmte Konstante definiert ist bzw. nicht definiert ist:

```
#ifdef KONSTANTE
    C++-Code
#endif
```

In diesem Fall wird der von `#ifdef KONSTANTE` und `#endif` eingeschlossene Teil übersprungen, falls eine Konstante mit dem angegebenen Namen nicht definiert ist. Das Gegenteil lässt sich mit

```
#ifndef KONSTANTE
...
#endif
```

erreichen. Jetzt wird der Code zwischen den beiden Direktiven nur dann kompiliert, wenn `KONSTANTE` nicht definiert ist. Beide Präprozessor-Direktiven lassen sich auch in Verbindung mit einem `#else`-Zweig einsetzen:

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

int main(void)
{
    // ...
    #ifdef DIESER
        cout << "Dieser Teil I wird";
        cout << " kompiliert" << endl;
    #else
        cout << "Der alternative Teil I";
        cout << " wird kompiliert" << endl;
    #endif
    #ifndef JENER
        cout << "Dieser Teil II wird";
```

```

    cout << " kompiliert" << endl;
#else
    cout << "Der alternative Teil II";
    cout << " wird kompiliert" << endl;
#endif
// ...
return 0;
}

```

Ausgabe:

```

Der alternative Teil I wird kompiliert
Der alternative Teil II wird kompiliert

```

In beiden `#ifdef` `#else`-Konstrukten wird der `#else`-Teil kompiliert, da im ersten Fall eine Konstante `DIESER` nicht definiert, im zweiten Fall eine Konstante `JENER` eben definiert ist.

Es sei darauf hingewiesen, dass im Code selbstverständlich beliebig viele Konstanten mit der `#define`-Direktive definiert werden dürfen. Wie schon erwähnt, kann dies grundsätzlich auch an beliebigen Stellen geschehen. Natürlich gilt eine Konstante erst ab der Position im Quellcode als definiert, an der sich die entsprechende `#define`-Direktive befindet. Aus diesem Grund ändert sich am Ergebnis des obigen Beispiels nichts, falls man eine Konstante `DIESER` nach der ersten `#ifdef`-Direktive vereinbart:

```

#include <iostream>
#define JENER
using namespace std;

int main(void)
{
    // ...
#ifdef DIESEr
    cout << "Dieser Teil I wird";
    cout << " kompiliert" << endl;
#else
    cout << "Der alternative Teil I";
    cout << " wird kompiliert" << endl;
#endif
#define DIESEr
#ifdef JENER
    cout << "Dieser Teil II wird";
    cout << " kompiliert" << endl;
#else
    cout << "Der alternative Teil II";
    cout << " wird kompiliert" << endl;
#endif
    // ...
}

```

```
    return 0;  
}
```

Ausgabe:

```
Der alternative Teil I wird kompiliert  
Der alternative Teil II wird kompiliert
```

Die Bedingungsauswertung

```
#ifdef DIESER
```

ergibt nach wie vor `false`, da eine Konstante `DIESER` zum Zeitpunkt noch nicht definiert ist.

Eine häufig angewandte Methode, um zu verhindern, dass der Code von Headerdateien doppelt kompiliert wird, besteht darin, mit `#ifndef` zu prüfen, ob eine bestimmte Konstante noch nicht definiert ist. Falls dies zutrifft – die Konstante also noch nicht definiert ist –, legt man innerhalb des `#ifndef` `#endif`-Konstrukts eine entsprechende Konstante fest. Dann ergibt die Auswertung der Bedingung

```
#ifndef KONSTANTE
```

beim nächsten Mal `false`, da diese Konstante ja nunmehr definiert ist, und die inneren Anweisungen werden nicht mehr übersetzt:

```
#ifndef KONSTANTE  
#define KONSTANTE  
Code_der_Headerdatei  
#endif
```

## Hinweis

Auch in den mitgelieferten Headerdateien Ihres Compilersystems ist der eigentliche Inhalt in der Regel auf die beschriebene Weise eingekreist. Wir hatten uns dieses Verfahrens in Kapitel 24 im Abschnitt 24.1.3 »Wie man vermeidet, dass eine Klassendefinition mehrfach eingebunden wird« bereits bedient.



# 28 Zeiger

Zeigervariablen – oder kurz Zeiger genannt (Englisch »pointer«) – sind spezielle Variablen, die die Adressen von anderen Datenobjekten aufnehmen können. Der Einsatz von Zeigern erlaubt es, Funktionen oder Methoden so einzurichten, dass diese die übergebenen Parameter des Aufrufers tatsächlich verändern. Des Weiteren – und dies kann in C++ als die Hauptaufgabe von Zeigern angesehen werden – ist es mithilfe von Zeigern möglich, Speicherplatz dynamisch – also zur Laufzeit – anzufordern. Bevor wir uns den eigentlichen Zeigern zuwenden, wollen wir zunächst auf den Adressoperator zu sprechen kommen.

## 28.1 Der Adressoperator (&)

Der Adressoperator & liefert die Adresse eines Datenobjekts in hexadezimaler Form zurück:

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

int main(void)
{
    long num;
    cout << &num;
    return 0;
}
```

Ausgabe: 0012FF7C

Die Ausgabe wird natürlich von Fall zu Fall verschieden aussehen, je nachdem, an welcher Stelle Ihr Compiler Platz für die Variable `num` reserviert. Es sei daran erinnert, dass es sich bei der Adresse eines Datenobjekts jeweils um dessen Anfangsadresse handelt. Diese wird vom Adressoperator zurückgeliefert. Nach der Ausgabe des obigen Beispiels nimmt die Variable `num` im Arbeitsspeicher daher die Speicherstellen mit den Adressen 0012FF7C, 0012FF7D, 0012FF7E und 0012FF7F ein, da es sich ja um eine Variable des Datentyps `long` handelt, die stets 4 Bytes beansprucht.

Wie Sie wissen, steht der Name eines Arrays für dessen Adresse (Kapitel 19, Abschnitt 19.2 »Arrays und Adressen«). Dabei handelt es sich um die Adresse der ersten Speicherzelle des Arrays und damit zugleich um die Anfangsadresse des ersten Elements. Daher geben bei dem Array

```
int arr[9];
```

die Ausdrücke

```
arr
```

und

```
&arr[0]
```

den gleichen Wert zurück, wie die Ausgabe des folgenden Beispiels beweist:

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

int main(void)
{
    int arr[9];
    cout << arr << endl;
    cout << &arr[0] << endl;
    return 0;
}
```

Ausgabe:

```
0012FF5C
0012FF5C
```

Natürlich lässt sich der Adressoperator auch auf benutzerdefinierte Datentypen anwenden:

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

class Auto
{
private:
    int geschw;
    int leistung;
    string farbe;
    // ...
public:
    // ...
};
```

```
int main(void)
{
    Auto meinAuto;
    cout << &meinAuto;
    return 0;
}
```

Ausgabe: 0012FF5C

## 28.2 Zeigervariablen deklarieren und verwenden

Wie eingangs erwähnt, ist eine Zeigervariable dazu gedacht, Adressen anderer Datenobjekte aufzunehmen. Allerdings trifft dies für einen bestimmten Zeiger jeweils nur auf bestimmte Adressen zu. Das heißt, ein konkreter Zeiger kann allein Adressen von Datenobjekten des Typs `int`, der andere des Typs `double` usw. enthalten. Wobei natürlich auch komplexe Datentypen erlaubt sind. Der Datentyp einer Zeigervariablen richtet sich also nach dem Datentyp der Datenobjekte, deren Adressen er aufnehmen soll. Um eine Zeigervariable zu deklarieren, gibt man in gewohnter Weise den gewünschten Datentyp und zusätzlich ein Sternchen (\*) an, um deutlich zu machen, dass es sich um eine Zeigervariable und nicht um eine gewöhnliche handeln soll. Mit der Anweisung

```
int *zgr;
```

vereinbart man also einen Zeiger mit dem Bezeichner `zgr`, der in der Lage ist, die Adressen von Integerdatenobjekten aufzunehmen. Das Sternchen muss dabei nicht notwendigerweise als Teil des Bezeichners geschrieben werden, sondern kann auch unmittelbar dem Datentyp folgen. Die Anweisungen

```
int * zgr;
```

oder

```
int* zgr;
```

sind daher gleichwertig. Auch die Form

```
int*zgr;
```

ist erlaubt, aber wegen der schlechteren Lesbarkeit nicht empfehlenswert.

Vorsicht ist jedoch geboten, wenn man mehrere Zeiger in einer Anweisung deklarieren möchte. So werden mit

```
int* zgr, p;
```

nicht etwa zwei Zeigervariablen `zgr` und `p` vereinbart, sondern eine Zeigervariable `zgr` und eine Integervariable `p`, was der Notation

```
int* zgr;
int p;
```

oder eben

```
int *zgr, p;
```

bzw.

```
int p, *zgr;
```

entspricht. Es ist also erforderlich, für jede Zeigervariable bei deren Deklaration das Sternchen anzugeben. Somit werden mit der Anweisung

```
int *zgr, *p;
```

tatsächlich zwei Zeiger `zgr` und `p` deklariert. Beachten Sie, dass der Name, mit dem die beiden Zeigervariablen im weiteren Code angesprochen werden, tatsächlich `zgr` bzw. `p` ist, nicht etwa `*zgr` bzw. `*p`. Das Sternchen, das im Zuge der Deklaration zu notieren ist, bezieht sich allein auf den Datentyp, ist also im weiteren Code wie der Typbezeichner nicht mehr anzugeben.

## Hinweis

Den Datentyp eines Zeigers bezeichnet man als »Zeiger auf ...«. Beispielsweise ist der Datentyp der Variablen `zgr` »Zeiger auf `int`«, geschrieben `int*`.

Eine Anweisung wie

```
int *zeiger1, *zeiger2;
```

bedeutet also die Vereinbarung zweier Variablen mit den Bezeichnern `zeiger1` und `zeiger2`, deren Datentyp jeweils `int*` (»Zeiger auf `int`«) ist. In dieser Hinsicht ist ein Zeiger eine Variable wie jede andere auch, nur dass sie eben für die Aufnahme von Adressen vorgesehen ist. Vereinbart man beispielsweise mit

```
int zahl;
```

eine Variable vom Datentyp `int`, so kann, wie Ihnen mittlerweile bekannt ist, die Adresse dieser Variablen mit

```
&zahl
```



angegeben werden. Folglich lässt sich dem oben vereinbarten Zeiger `zgr` die Adresse der Variablen `zahl` wie folgt zuweisen:

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

int main(void)
{
    int zahl;
    int *zgr;
    zgr = &zahl;
    cout << zgr << endl;
    cout << &zahl << endl;
    return 0;
}
```

Ausgabe:

```
0012FF7C
0012FF7C
```

Wie an der Ausgabe zu erkennen ist, enthält die Zeigervariable `zgr` nach der Zuweisung die Adresse von `zahl`.

### Hinweis

Wenn eine Zeigervariable die Adresse eines anderen Datenobjekts enthält, sagt man, sie »zeigt« auf dieses Datenobjekt. Der Zeiger `zgr` zeigt somit nach der Zuweisung

```
zgr = &zahl;
```

auf die Variable `zahl`.

### Hinweis

Zur Verdeutlichung ist im obigen Listing der Zeiger `zgr` sowie die Integervariable `zahl` in einer jeweils eigenen Anweisung deklariert. Nach dem oben Gesagten hätte dies natürlich auch in einer einzigen Anweisung geschehen können:

```
int *zgr, zahl;
```

Natürlich ist es auch möglich, eine Zeigervariable bei deren Deklaration zu initialisieren:

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

```
int main(void)
{
    int zahl;
    int *zgr = &zahl;
    cout << zgr << endl;
    cout << &zahl << endl;
    return 0;
}
```

Ausgabe:

```
0012FF7C
0012FF7C
```

## 28.2.1 Wilde Zeiger

Betrachtet man folgendes Codestück, so fällt auf, dass der Zeiger `zgr` nach Verlassen des inneren Blocks auf ein Datenobjekt verweist, das nicht mehr existent ist. Der für die Variable `a` vom Programm reservierte Speicherbereich wird ja an dieser Stelle wieder freigegeben:

```
{
    int *zgr ;
    {
        int a;
        zgr = &a;
    }
    // ...
}
```

Nichtsdestoweniger behält die Zeigervariable `zgr` ihre Gültigkeit. Sie ist ja im äußeren Block deklariert. Nur dass es sich bei dem Wert, den sie nach dem Ende des inneren Blocks enthält, um die Adresse eines nicht bzw. nicht mehr reservierten Speicherbereichs handelt. Dies kann auf jeden Fall dann zu Problemen führen, wenn an dieser Stelle im weiteren Verlauf Daten eines anderen Typs gespeichert werden, sei es vom eigenen Programm oder von anderen Programmen. In so einem Fall spricht man von ungerichteten bzw. von wilden Zeigern. Deren Auftreten sollte man in seinen Programmen tunlichst vermeiden. Dazu stellt C++ einen wohldefinierten Wert für Zeiger zur Verfügung, der mit

```
0
```

oder der symbolischen Konstanten

```
NULL
```

dargestellt wird. Dem Leser sei empfohlen, die letztgenannte Variante vorzuziehen, um im Code deutlich zu machen, dass es sich um einen Zeiger handelt:

```
{
    int *zgr ;
    {
        int a;
        zgr = &a;
    }
    zgr = NULL;
    // ...
}
```

### Tipp

Um Komplikationen von vornherein zu vermeiden, sollten Sie in Ihren Programmen dafür sorgen, dass Zeigervariablen an jeder Stelle im Code einen wohldefinierten Wert besitzen.

## 28.2.2 Dereferenzierung von Zeigern

Eine Variable, auf die ein Zeiger verweist (zeigt), kann auch über diesen Zeiger angesprochen werden, indem man diesen *dereferenziert*. Dadurch greift man auf den *Inhalt* einer Variablen zu. Dazu dient der Dereferenzierungsoperator, der durch ein Sternchen (\*) dargestellt wird und der dem entsprechenden Bezeichner für eine Zeigervariable vorangestellt wird.

Um – bezogen auf obiges Beispiel – die Variable `zahl` anzusprechen, kann natürlich nach wie vor per

```
zahl
```

zugegriffen werden, aber auch über den Zeiger `zgr` mit

```
*zgr
```

### Hinweis

Der Dereferenzierungsoperator hat das gleiche Symbol wie das Zeichen, das bei der Deklaration eines Zeigers angegeben wird, nämlich das Sternchen (\*). Dennoch handelt es sich hierbei um zwei völlig verschiedene Dinge. Was jeweils gemeint ist, erkennt der Compiler am Kontext.

Um also der Variablen `zahl` z.B. den Wert 23 zuzuweisen, könnte man

```
zahl = 23;
```

oder eben

```
*zgr = 23;
```

schreiben:

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

int main(void)
{
    int zahl;
    int *zgr = &zahl;
    *zgr = 23;
    cout << *zgr << endl;
    cout << zahl << endl;
    return 0;
}
```

Ausgabe:

```
23
23
```

Das Gleiche gilt natürlich, wenn man eine Benutzereingabe in der Variablen `zahl` speichern möchte. Dies lässt sich ebenfalls mit

```
cin >> zahl;
```

oder mit

```
cin >> *zgr;
```

erreichen, indem man die Variable `zahl` über den Zeiger anspricht. Ein dereferenzierter Zeiger kann also auf die gleiche Weise verwendet werden wie der Bezeichner einer Variablen, deren Adresse er enthält.

Natürlich kann sich der Wert eines Zeigers – wie das bei allen Variablen der Fall ist – im weiteren Code auch ändern:

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

int main(void)
{
    double z1 = -171.5, z2 = 99.89;
    double *zeig = &z1;
    cout << *zeig << endl;
    zeig = &z2;
    cout << *zeig << endl;
    return 0;
}
```

Ausgabe:

```
-171.5
99.89
```

### 28.2.3 Konstante Zeiger

Allerdings ist es auch möglich, konstante Zeiger zu vereinbaren. Diese dürfen ihren Wert im weiteren Verlauf nicht mehr ändern. Dazu ist bei der Deklaration das Schlüsselwort `const` anzugeben. Außerdem muss ein konstanter Zeiger initialisiert werden:

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

int main(void)
{
    int zahl = 15;
    int * const intPointer = &zahl;
    cout << *intPointer;
    return 0;
}
```

Ausgabe: 15

#### Referenz

Zu `const` siehe Kapitel 12, Abschnitt 12.4 »Konstanten mit `const` deklarieren«.

### Hinweis

Als Bezeichner für Zeiger werden häufig `z`, `zgr`, `zeig`, `zeiger`, `p` und `pointer` verwendet. Mitunter stellt man auch ein Kürzel voran, um auf den Datentyp hinzuweisen:

```
intZgr bzw. iZgr
dPointer
doubleZeig
```

usw. Es steht Ihnen grundsätzlich frei, wie Sie das in Ihren Programmen halten wollen. Sinnvoll kann es aber sein, eine einmal eingeführte Konvention beizubehalten.

Der Versuch, dem konstanten Zeiger `intPointer` im weiteren Code die Adresse eines anderen Datenobjekts zuzuweisen, würde nun eine Fehlermeldung nach sich ziehen:

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

int main(void)
{
    int zahl = 15, nummer = 9;
    int * const intPointer = &zahl;
    cout << *intPointer;
    intPointer = &nummer; // FEHLER
    return 0;
}
```

Es sei darauf hingewiesen, dass es für die Vereinbarung von `intPointer` nicht von Bedeutung ist, ob `zahl` an dieser Stelle einen definierten Wert besitzt oder nicht. Auch darf sich der Wert der Variablen `zahl`, auf die `intPointer` zeigt, sehr wohl ändern. Diese ist ja nicht konstant.

### Hinweis

Anders verhält es sich natürlich, wenn man `zahl` ebenfalls mit `const` vereinbart.

Beachten Sie, dass das Schlüsselwort `const` – wie im obigen Listing – unmittelbar vor dem Bezeichner stehen muss, um einen konstanten Zeiger zu vereinbaren:

```
double * const p = &betrag;
```

Entsprechend würde man mit

```
int * const p = &num, * const zeig = &zahl;
```

zwei konstante Zeiger `p` und `zeig` vereinbaren, deren Werte – die Adressen der Variablen `num` bzw. `zahl` – sich im weiteren Verlauf nicht ändern dürfen.

## Hinweis

Verwendet man das Schlüsselwort `const` dagegen wie gewohnt vor dem Datentypbezeichner, so bedeutet dies, dass der Inhalt einer Variablen, auf die der Zeiger zeigt, nicht über den Zeiger geändert werden kann:

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

int main(void)
{
    double z = 8.98;
    const double *zgr = &z;
    cout << *zgr; // Ausgabe: 8.98
    z = 100.0; // OK
    cout << *zgr; // Ausgabe: 100
    *zgr = 555; // FEHLER
    return 0;
}
```

Die Anweisung

```
*zgr = 555;
```

ist fehlerhaft, da die Variable `z`, wie gesagt, nicht über den Zeiger `zgr` geändert werden darf. Andererseits ist es möglich, `zgr` im weiteren Code die Adressen anderer Variablen des Typs `double` zuzuweisen (in diesem Sinne wäre es prinzipiell auch nicht zwingend notwendig gewesen, `zgr` in der Deklarationsanweisung mit der Adresse von `z` zu initialisieren). Im Übrigen würde man mit

```
const int * const zgr = &zahl;
```

einen konstanten Zeiger vereinbaren, über den der Wert der Variablen `zahl` nicht verändert werden darf.

## 28.2.4 Elementzugriff über Zeiger

Wie bereits erwähnt, sind die möglichen Datentypen von Zeigern nicht etwa auf die elementaren beschränkt. Grundsätzlich können Zeiger jeden beliebigen Datentyps vereinbart werden:

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

class Mitarbeiter
{
public:
    string vorname;
```

```

    string nachname;
    int gehalt;
    // ...
};

int main(void)
{
    Mitarbeiter Mueller, Meier;
    Mitarbeiter *zgrMit;
    zgrMit = &Meier;
    return 0;
}

```

### Hinweis

Der Einfachheit halber werden wir in den folgenden Beispielen dieses Kapitels alle Elemente einer Klasse als `public` spezifizieren.

Mit der Zuweisung

```
zgrMit = &Meier;
```

erhält der Zeiger `zgrMit` die Adresse des Mitarbeiter-Objekts `Meier`. Somit ist beispielsweise eine Zuweisung

```
Mueller = *zgrMit;
```

möglich, die die Werte der Instanz `Meier` elementweise in die von `Mueller` kopiert. Umgekehrt würde die Anweisung

```
*zgrMit = Mueller;
```

die Werte von `Mueller` in die Instanz `Meier` kopieren. Beides, da mit

```
*zgrMit
```

das Objekt `Meier` angesprochen wird.

Um jedoch über die Zeigervariable `zgrMit` beispielsweise auf das Attribut `gehalt` von `Meier` zuzugreifen, ist neben der Dereferenzierung des Weiteren der Elementzugriff notwendig. Es handelt sich also um zwei Operationen, die in der genannten Reihenfolge durchzuführen sind:

```
(*zgrMit).gehalt = 3500;
```



Mit obiger Anweisung wird dem Attribut `gehalt` der Instanz `Meier` über den Zeiger `zgrMit` der Wert 3500 zugewiesen. Die Klammerung ist erforderlich, da der Dereferenzierungsoperator (Prioritätsstufe 15) eine niedrigere Priorität als der Punktoperator (Prioritätsstufe 16) hat. Andernfalls würde die Reihenfolge der Operationen

```
*(zgrMit.gehalt) = 3500;
```

entsprechen, was nicht interpretierbar ist, da eine Adresse – hier `zgrMit` – ja keine Datenelemente besitzt.

Alternativ stellt C++ den Operator `->` (Pfeiloperator) zur Verfügung, der beide Operationen – Dereferenzierung und Elementzugriff – in einem symbolisiert, sodass die Anweisungen

```
(*zgrMit).gehalt = 3500;
```

bzw.

```
zgrMit->gehalt = 3500;
```

gleichwertig sind:

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

class Mitarbeiter
{
public:
    string vorname;
    string nachname;
    int gehalt;
    // ...
};

int main(void)
{
    Mitarbeiter Mueller, Meier;
    Mitarbeiter *zgrMit;
    zgrMit = &Meier;
    zgrMit->vorname = "Hans";
    zgrMit->nachname = "Meier";
    zgrMit->gehalt = 3500;
    cout << "Gehalt von " << zgrMit->vorname << ' '
         << zgrMit->nachname << ": " << zgrMit->gehalt;
    return 0;
}
```

Ausgabe:

Gehalt von Hans Meier: 3500

## 28.3 Zeiger als Parameter von Funktionen

Wenn man möchte, dass Funktionen tatsächlich Veränderungen an Variablen der aufrufenden Umgebung durchführen, kann man als formale Parameter Zeiger definieren. Der Funktion sind dann beim Aufruf die Adressen der entsprechenden Variablen zu übergeben, wodurch diese der Funktion bekannt gemacht werden.

### Hinweis

Was hier und im Folgenden über Funktionen gesagt wird, gilt natürlich in gleicher Weise für Methoden. Auch muss es sich bei den Parametern nicht notwendig um Zeiger auf elementare Datentypen handeln. Methoden wie Funktionen lassen sich selbstverständlich auch so einrichten, dass diese beim Aufruf die Adresse von Klassenobjekten erhalten.

So lässt sich eine Funktion `tausch()` einrichten, die das in Kapitel 10 im Abschnitt 10.5 »Dreieckstausch« gezeigte Verfahren mit dem Ergebnis durchführt, dass die Werte zweier Variablen der aufrufenden Umgebung vertauscht werden:

```
void tausch(double *a, double *b)
{
    double tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Der Funktion sind beim Aufruf die Adressen zweier `double`-Variablen zu übergeben, deren Werte miteinander vertauscht werden:

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

void tausch(double *a, double *b)
{
    double tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
int main(void)
{
    double zahl1 = -876.9, zahl2 = 45.7;
    cout << "Vor dem Tausch:" << endl;
    cout << "Wert von zahl1: " << zahl1 << endl;
    cout << "Wert von zahl2: " << zahl2 << endl;
    tauschk(&zahl1, &zahl2);
    cout << "Nach dem Tausch:" << endl;
    cout << "Wert von zahl1: " << zahl1 << endl;
    cout << "Wert von zahl2: " << zahl2 << endl;
    return 0;
}
```

Ausgabe:

```
Vor dem Tausch:
Wert von zahl1: -876.9
Wert von zahl2: 45.7
Nach dem Tausch:
Wert von zahl1: 45.7
Wert von zahl2: -876.9
```

Wie an der Ausgabe zu sehen ist, wurden die Variablen `zahl1` und `zahl2` von der Funktion `tauschk()` verändert. Die gezeigte Art der Parameterübergabe wird *call by reference* genannt.

### Hinweis

Zu *call by value* siehe Kapitel 20, Abschnitt 20.3.1 »Wertübergabe«.

### Hinweis

Es sei daran erinnert, dass wir in Kapitel 20 im Abschnitt 20.3.2 »Arrays an Funktionen übergeben« und in Kapitel 21 »Eine Funktionsbibliothek« bereits Zeiger als Funktionsparameter verwendet haben, um Arrays in Funktionen bekannt zu machen.

## 28.4 Zeiger als Klassenelemente

Mithilfe von Zeigerattributen ist es möglich, mehrere Klassenobjekte miteinander zu verketteten:

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

class Rechteck
{
public:
    int Laenge;
    int Breite;
    Rechteck *RePointer;
    Rechteck()
    {RePointer = NULL;}
    int getFlaeche()
    {return Laenge * Breite;}
};

int main(void)
{
    Rechteck Erstes, Zweites, Drittes;
    Rechteck *ReAnfang = &Erstes;
    Drittes.Laenge = 20;
    Drittes.Breite = 15;
    Erstes.RePointer = &Zweites;
    Zweites.RePointer = &Drittes;
    return 0;
}
```

Mit den Anweisungen

```
Erstes.RePointer = &Zweites;
Zweites.RePointer = &Drittes;
```

wurden die drei Rechtecke des Programms praktisch aneinander gehängt: Das erste Rechteck – bzw. dessen Attribut `RePointer` – zeigt auf das zweite und das zweite in gleicher Weise auf das dritte.

### Hinweis

Wenn Klassenobjekte in dieser Form miteinander verkettet sind, spricht man von linearen Listen.

Nach

```
Rechteck *ReAnfang = &Erstes;
```

steht mit ReAnfang zudem ein Zeiger auf den Anfang der Liste zur Verfügung. Mit diesem Anfangszeiger können nun alle in der Liste vorkommenden Elemente angesprochen werden. So lässt sich etwa der Zugriff auf die Methode getFlaeche() des dritten Elements mit dem Zeiger ReAnfang per

```
ReAnfang->RePointer->RePointer->getFlaeche();
```

durchführen.

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

class Rechteck
{
public:
    int Laenge;
    int Breite;
    Rechteck *RePointer;
    Rechteck()
    {RePointer = NULL;}
    int getFlaeche()
    {return Laenge * Breite;}
};

int main(void)
{
    Rechteck Erstes, Zweites, Drittes;
    Rechteck *ReAnfang = &Erstes;
    Drittes.Laenge = 20;
    Drittes.Breite = 15;
    Erstes.RePointer = &Zweites;
    Zweites.RePointer = &Drittes;
    cout << "Fl\x84" << "che des dritten Rechtecks: "
         << ReAnfang->RePointer->RePointer->getFlaeche();
    return 0;
}
```

Ausgabe:

```
Fläche des dritten Rechtecks: 300
```

Da ReAnfang auf das erste Rechteck zeigt, bedeutet

```
ReAnfang->RePointer
```

den Zugriff auf dessen Zeigerattribut und

```
ReAnfang->RePointer->RePointer
```

den Zugriff auf das Zeigerattribut des zweiten Rechtecks. Dieses dereferenziert zeigt auf das dritte Rechteck-Objekt mit Auswahl des Elements `getFlaeche()`.

```
ReAnfang->RePointer->RePointer->getFlaeche()
```

So viel nur zur Demonstration. Natürlich ist der Zugriff im Beispiel mit

```
Drittes.getFlaeche()
```

nahe liegender. Lineare Listen spielen vor allem dann eine Rolle, wenn es darum geht, eine Vielzahl von Klassenobjekten, die mit dem `new`-Operator zur Laufzeit instanziiert werden, zu verwalten (zur dynamischen Speicherreservierung siehe Abschnitt 28.6 »Speicherplatz dynamisch anfordern«) – etwa bei einem Kundenverwaltungsprogramm, wenn man dem Benutzer die Entscheidung überlässt, einen neuen Kunden anzulegen.

### Hinweis

In diesem Fall wird das neu instanziierte Objekt im Programm an den Anfang der linearen Liste gehängt.

## 28.5 Der `this`-Zeiger von Methoden

Jede nicht statische Methode verfügt über einen zusätzlichen Parameter, der ihr beim Aufruf automatisch übergeben wird, den `this`-Zeiger. Dabei handelt es sich um einen konstanten Zeiger mit diesem Namen, der auf das Objekt zeigt, für das die Methode aufgerufen wird.

Hierzu ein Beispiel. In einer Klasse `Mitarbeiter` soll eine Methode definiert werden, die das Gehalt eines Mitarbeiters um 10 Prozent erhöht:

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

class Mitarbeiter
{
public:
    double gehalt;
    // ...
```

```
Mitarbeiter()
{gehalt = 2500.0;}
void mehrGehalt()
{gehalt += gehalt * 0.1;}
// ...
};

int main(void)
{
    Mitarbeiter Duck;
    cout << "Gehalt von Duck: " << Duck.gehalt << endl;
    Duck.mehrGehalt();
    cout << "Gehalt von Duck: " << Duck.gehalt << endl;
    return 0;
}
```

Ausgabe:

```
Gehalt von Duck: 2500
Gehalt von Duck: 2750
```

Um nun wiederholte Aufrufe der Methode `mehrGehalt()` in der Form

```
Duck.mehrGehalt()->mehrGehalt()->mehrGehalt()
```

zu ermöglichen, kann die Methode so eingerichtet werden, dass diese einen Zeiger auf das aktuelle Objekt zurückliefert. Dies lässt sich erreichen, indem man aus der Methode einfach den `this`-Zeiger zurückgibt:

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

class Mitarbeiter
{
public:
    double gehalt;
    // ...
    Mitarbeiter()
    {gehalt = 2500.0;}
    Mitarbeiter* mehrGehalt()
    {
        gehalt += gehalt * 0.1;
        return this;
    }
    // ...
};
```

```
int main(void)
{
    Mitarbeiter Duck;
    cout << "Gehalt von Duck: " << Duck.gehalt << endl;
    Duck.mehrGehalt()->mehrGehalt()->mehrGehalt();
    cout << "Gehalt von Duck: " << Duck.gehalt << endl;
    return 0;
}
```

Ausgabe:

```
Gehalt von Duck: 2500
Gehalt von Duck: 3327.5
```

### Hinweis

Will man beispielsweise innerhalb einer Methode das aktuelle Objekt als Ganzes ansprechen, so dereferenziert man den `this`-Zeiger einfach:

```
*this
```

## 28.6 Speicherplatz dynamisch anfordern

Wie Ihnen bereits bekannt ist, werden Datenobjekte in verschiedenen Bereichen des Arbeitsspeichers verwaltet: nicht statische lokale Variablen auf dem Stack und Variablen der Speicherklasse `static` in einem globalen Namensbereich (Kapitel 20, Abschnitt 20.8 »Globale Variablen«). Ein weiterer Bereich, der so genannte »Heap«, steht für die Reservierung von Speicher zur Verfügung, der von Programmen zur Laufzeit angefordert wird.

Zur dynamischen Speicherverwaltung können im Code die Operatoren `new` und `delete` verwendet werden.

### Hinweis

Die Anforderung von Speicher im Programm wird auch »Allokation« genannt.

Mit `new` alloziert man Speicher für ein Datenobjekt, mit `delete` gibt man diesen wieder frei. Beispielsweise fordert man mit

```
new int;
```

Platz für eine Integervariable an. Es ist allerdings nicht empfehlenswert, dies in dieser Form zu tun, da der mit obiger Anweisung reservierte Speicherbereich im weiteren Code



nicht mehr angesprochen werden kann. Tatsächlich sind Objekte auf dem Heap namenlos. Ihre Verwaltung erfolgt ausschließlich über Adressen. Nun ist es so, dass der `new`-Operator die Adresse des angelegten Datenobjekts zurückgibt. Um dieses im weiteren Verlauf ansprechen zu können, ist es also erforderlich, den Rückgabewert von `new` in einem Zeiger festzuhalten:

```
int *intPointer;  
intPointer = new int;
```

oder in einer Anweisung:

```
int *intPointer = new int;
```

Im Weiteren kann nun die Variable auf dem Heap mithilfe des – dereferenzierten – Zeigers `intPointer` angesprochen werden:

```
#include <iostream>  
using namespace std;  
  
int main(void)  
{  
    int *intPointer = new int;  
    *intPointer = 40;  
    cout << *intPointer;  
    return 0;  
}
```

Ausgabe: 40

Will man den Zeiger wieder löschen, also den reservierten Speicherbereich wieder freigeben, so wendet man den Operator `delete` auf den Zeiger an:

```
#include <iostream>  
using namespace std;  
  
int main(void)  
{  
    int *intPointer = new int;  
    *intPointer = 40;  
    cout << *intPointer;  
    delete intPointer;  
    intPointer = NULL;  
    return 0;  
}
```

Ausgabe: 40

Beachten Sie, dass mit der Anweisung

```
delete intPointer;
```

allein der zuvor allozierte Speicher, nicht jedoch der Zeiger `intPointer` gelöscht wird. Dieser besitzt nach wie vor Gültigkeit.

### Hinweis

Die Zeigervariable `intPointer` selbst wird ja auf dem Stack verwaltet, besitzt also die Speicherklasse `auto`, während die namenlose, mit `new` angeforderte Integervariable, auf die der Zeiger nach der Zuweisung

```
int *intPointer = new int;
```

verweist, auf dem Heap angelegt wird.

Die Anweisung

```
intPointer = NULL;
```

dient dazu, dem Zeiger einen wohldefinierten Wert zu geben. Ansonsten würde es sich um einen wilden Zeiger handeln (Abschnitt 28.2.1 »Wilde Zeiger«).

Es sei ausdrücklich darauf hingewiesen, dass bei dynamischer Allokation von Speicher der Programmierer die Verantwortung dafür trägt, dass dieser auch wieder freigegeben wird. Lässt man also im obigen Beispiel die `delete`-Anweisung weg, so bleibt der zuvor mit

```
new int
```

angeforderte Bereich auch nach dem Ende der Programmausführung belegt, sodass er von anderen Programmen nicht mehr genutzt werden kann (freigegeben wird er dann erst bei einem Neustart des Computers).

Falls auf dem Heap nicht genügend Speicher zur Verfügung steht, schlägt die dynamische Speichieranforderung fehl. In diesem Fall gibt der `new`-Operator den Wert `NULL` zurück. Um also im Programm festzustellen, ob die Allokation erfolgreich war, kann der entsprechende Zeiger auf diesen Wert geprüft werden:

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

int main(void)
{
    double *dp = new double;
    if (dp == NULL)
        return -1;
```

```

else
{
    cin >> *dp;
    cout << *dp;
    delete dp;
    dp = NULL;
}
return 0;
}

```

## 28.6.1 Arrays dynamisch allozieren

Wenn man auf dem Heap Platz für ein Array anfordert, geschieht dies in der entsprechenden Weise, nur dass wie gewöhnlich die Elementzahl des Arrays anzugeben ist:

```

int *ip;
ip = new int[7];

```

Mit obiger Anweisung wird ein Array aus sieben Elementen erzeugt, dessen Adresse dem Zeiger `ip` zugewiesen wird. Mit diesem kann nun der Zugriff auf die einzelnen Arrayelemente erfolgen:

```

#include <iostream>
using namespace std;

int main(void)
{
    int *ip = new int[7];
    int i;
    for (i = 0; i < 7; i++)
        ip[i] = i * i;
    for (i = 0; i < 7; i++)
        cout << ip[i] << ' ';
    return 0;
}

```

Ausgabe: 0 1 4 9 16 25 36

Beachten Sie, dass der Operator `[]` – ähnlich dem Operator `->` bezüglich der Elemente von Klassenobjekten – Elementauswahl und Dereferenzierung in einem darstellt. So bedeutet, bezogen auf obiges Beispiel, ein Ausdruck wie

```
ip[5]
```

»Gehe an die Speicherstelle `ip`, dann  $5 * 4$  Bytes weiter und greife auf den dortigen Inhalt zu!«

Um den für ein Array dynamisch reservierten Speicherbereich wieder freizugeben, ist wiederum der `delete`-Operator zu verwenden – mit Angabe von `[]`, um deutlich zu machen, dass es sich um ein Array handelt:

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

int main(void)
{
    int *ip = new int[7];
    int i;
    for (i = 0; i < 7; i++)
        ip[i] = i * i;
    for (i = 0; i < 7; i++)
        cout << ip[i] << ' ';
    delete[] ip;
    ip = NULL;
    return 0;
}
```

Ausgabe: 0 1 4 9 16 25 36

Anders als bei Arrays, die auf dem Stack verwaltet werden, muss es sich bei der in eckigen Klammern anzugebenden Elementzahl von dynamisch angeforderten Arrays nicht um einen konstanten Wert handeln:

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

class Kunde
{
public:
    string vorname;
    string name;
    int kundenr;
    // ...
};

int main(void)
{
    int anzahl;
    cout << "Wie viele Kunden wollen Sie anlegen? ";
    cin >> anzahl;
    Kunde *MaPointer = new Kunde[anzahl];
    // ...
    delete[] MaPointer;
    MaPointer = NULL;
    return 0;
}
```

Im obigen Beispiel wird zur Angabe der Elementzahl des mit `new` erzeugten Arrays die Variable `anzahl` verwendet.

### Hinweis

Es sei daran erinnert, dass nach

```
int anzahl;
```

der Versuch, ein Array `arr` mit

```
int arr[anzahl]; // FEHLER
```

zu vereinbaren, eine Fehlermeldung zur Folge haben würde (Kapitel 18, Abschnitt 18.1 »Deklaration von Arrays«).

## 28.6.2 Destruktoren sorgen für die Aufräumarbeiten

Wie an anderer Stelle bereits angedeutet, ist es die vornehmliche Aufgabe eines Destruktors, dynamisch angeforderten Speicher wieder freizugeben, wenn ein Klassenobjekt seine Gültigkeit verliert (Kapitel 24, Abschnitt 24.3 »Konstruktoren und Destruktoren«).

Als Beispiel soll uns die Bank-Klasse aus Kapitel 26 im Programm des Abschnitts 26.2.1 »Programm zum Verwalten von (Entenhausener) Bankkunden« dienen, in der ein Array von zehn Bankkunden angelegt wird:

```
class Bankkunde
{
    // ...
};

class Bank
{
public:
    Bankkunde Kunden[10];
    // ...
};
```

Die Bank-Klasse soll nun so geändert werden, dass diese lediglich einen Zeiger auf Bankkunde enthält. Außerdem wird ein Konstruktor definiert, in dem die eigentlichen Bankkunde-Objekte dynamisch alloziert werden:

```
class Bankkunde
{
    // ...
};
```

```
class Bank
{
public:
    Bankkunde *Kunden;
    Bank()
    {
        Kunden = new Bankkunde[10];
    }
    // ...
};
```

Instanziert man nun ein Bank-Objekt wie folgt

```
Bank Entenhausener;
```

wird ebenfalls ein Array aus zehn Elementen des Typs Bankkunde erzeugt, allerdings auf dem Heap. Das bedeutet, dass der mit

```
new Bankkunde[10]
```

allozierte Speicherbereich nicht automatisch gelöscht wird, wenn das Objekt Entenhausener seine Gültigkeit verliert, wohl aber der Zeiger Kunden. Dieser wurde ja auf dem Stack angelegt.

Hier ist nun die Destruktormethode gefragt. Da diese für jedes Klassenobjekt bei dessen Erlöschen automatisch ausgeführt wird, bietet es sich an, die notwendigen Aufräumarbeiten dort zu erledigen:

```
class Bankkunde
{
    // ...
};

class Bank
{
public:
    Bankkunde *Kunden;
    Bank()
    {
        Kunden = new Bankkunde[10];
    }
    ~Bank()
    {
        delete[] Kunden;
    }
    // ...
};
```

## Referenz

Zur Definition von Destruktoren siehe Kapitel 24, Abschnitt 24.3 »Konstruktoren und Destruktoren«.

Wenn nun ein Bank-Objekt seine Gültigkeit verliert, sorgt der Destruktor dafür, dass der Speicherbereich, auf den der Zeiger Kunden verweist, freigegeben wird. Dies geschieht gerade noch rechtzeitig, da Kunden unmittelbar danach ja nicht mehr zur Verfügung steht.

## Hinweis

Versäumt man es, dynamisch angeforderten Speicher freizugeben, solange dieser noch ansprechbar ist, so entstehen so genannte Speicherleichen. Womit ein Speicherbereich gemeint ist, der nicht mehr freigegeben werden kann, da eben kein Zeiger darauf verweist.

Bezüglich des oben genannten Programms aus Kapitel 26 (Abschnitt 26.2.1) lässt sich nun der Umstand ausnutzen, dass das Instanzieren der Bankkunden dynamisch erfolgt. So kann die Anzahl der Kunden vom Programm eingelesen werden, indem man einen entsprechenden Wert in die zugehörige Textdatei schreibt (siehe Abbildung 28.1, der erste Wert gibt Auskunft über die Anzahl der Bankkunden).

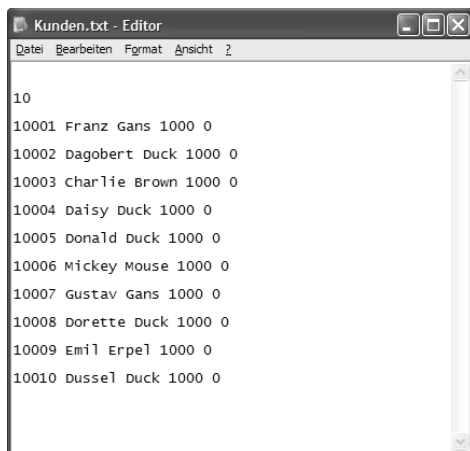


Abbildung 28.1: Textdatei Kunden.txt

Für den Fall, dass also Kunden hinzukommen oder wegfallen, muss in der Textdatei nur die erste Zahl von Hand geändert werden, um dies im Programm zu berücksichtigen.

## Hinweis

Außerdem müssen natürlich die Anfangsdaten des neuen Kunden editiert bzw. die Daten eines nicht mehr vorhandenen Kunden gelöscht werden.

Um das Programm dementsprechend einzurichten, ist es sinnvoll, das Einlesen der Kunden im Konstruktor der Bank-Klasse durchzuführen. Dies lässt sich einfach dadurch erreichen, dass man die Methode `KundenEinlesen()` dort aufruft, wobei das Instanzieren der Bankkunde-Objekte in dieser Methode stattfinden soll. Damit die Anzahl der Kunden auch den Methoden `KundenInDatei()` und `KundeSuchen()` zur Verfügung steht, soll die Klasse mit einem entsprechenden Attribut versehen werden. Dieses wird für den Zugriff von außerhalb der Klasse nicht gebraucht und kann daher ohne zusätzlichen Aufwand als `private` spezifiziert werden. Außerdem muss in der Methode `KundenInDatei()` dafür gesorgt werden, dass der Wert für die Kundenzahl für den nächsten Programmlauf wieder in die Textdatei geschrieben wird:

```
class Bank
{
private:
    int Anzahl;
public:
    Bankkunde *Kunden;
    Bank() {KundenEinlesen();}
    ~Bank() {delete[] Kunden;}
    int KundeSuchen(int nr);
    // ...
    void KundenEinlesen();
    void KundenInDatei();
};

int Bank::KundeSuchen(int nr)
// Rueckgabewert ist -1, falls Konto nicht vorhanden,
// ansonsten der Index des Kunden
{
    for (int i = 0; i < Anzahl; i++)
    {
        if (Kunden[i].getKontonr() == nr)
            return i;
    }
    return -1;
}

// ...

void Bank::KundenEinlesen()
{
    int hilf;
    string name;
    double help;
    ifstream lesen("Kunden.txt");
    lesen >> Anzahl;
    Kunden = new Bankkunde[Anzahl];
}
```



```

    for (int i = 0; i < Anzahl; i++)
    {
        lesen >> hilf; Kunden[i].setKontonr(hilf);
        lesen >> name; Kunden[i].setVorname(name);
        lesen >> name; Kunden[i].setNachname(name);
        lesen >> help; Kunden[i].setLimit(help);
        lesen >> help; Kunden[i].setKontostand(help);
    }
    lesen.close();
}

void Bank::KundenInDatei()
{
    int hilf;
    string name;
    double help;
    ofstream schreiben("Kunden.txt");
    schreiben << endl << endl << Anzahl;
    for (int i = 0; i < Anzahl; i++)
    {
        hilf = Kunden[i].getKontonr();
        schreiben << endl << endl << hilf << ' ';
        name = Kunden[i].getVorname();
        schreiben << name << ' ';
        name = Kunden[i].getNachname();
        schreiben << name << ' ';
        help = Kunden[i].getLimit();
        schreiben << help << ' ';
        help = Kunden[i].getKontostand();
        schreiben << help;
    }
    schreiben.close();
}

```

Hier das vollständige Listing:

```

#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
using namespace std;

class Bankkunde
{
private:
    string vorname;
    string nachname;

```

```

    int kto_nr;
    double kto_stand;
    double limit;
public:
    Bankkunde() {}
    void setVorname(string v)
    {vorname = v;}
    string getVorname()
    {return vorname;}
    void setNachname(string n)
    {nachname = n;}
    string getNachname()
    {return nachname;}
    void setKontonr(int nr)
    {kto_nr = nr;}
    int getKontonr()
    {return kto_nr;}
    void setKontostand(double betrag)
    {kto_stand = betrag;}
    double getKontostand()
    {return kto_stand;}
    void setLimit(double l);
    double getLimit()
    {return limit;}
    void einzahlen(double betrag);
    int auszahlen(double betrag);
}; // end class Bankkunde

void Bankkunde::setLimit(double l)
{
    if (l >= 0)
        limit = l;
}

void Bankkunde::einzahlen(double betrag)
{
    if (betrag > 0)
        kto_stand += betrag;
}

int Bankkunde::auszahlen(double betrag)
{
    if (betrag <= 0)
        return 2; // negativer Auszahlungsbetrag
    if (kto_stand -betrag >= -limit)
    {
        kto_stand -= betrag;
    }
}

```

```

        return 0; // Auszahlung ist erfolgt
    }
    else
        return 1; // Auszahlung nicht moeglich
}

class Bank
{
private:
    int Anzahl;
public:
    Bankkunde *Kunden;
    Bank() {KundenEinlesen();}
    ~Bank() {delete[] Kunden;}
    int KundeSuchen(int nr);
    int ueberweisen(int auftr, int empf, double betrag);
    void KundenEinlesen();
    void KundenInDatei();
}; // end class Bank

int Bank::KundeSuchen(int nr)
// Rueckgabewert ist -1, falls Konto nicht vorhanden,
// ansonsten der Index des Kunden
{
    for (int i = 0; i < Anzahl; i++)
    {
        if (Kunden[i].getKontonr() == nr)
            return i;
    }
    return -1;
}

int Bank::ueberweisen(int auftr, int empf, double betrag)
{
    if (betrag <= 0)
        return -1; // Programmierfehler
    if (Kunden[auftr].auszahlen(betrag) == 0) {
        Kunden[empf].einzahlen(betrag);
        return 0; // Ueberweisung ausgefuehrt
    }
    else
        return 1; // Konto nicht gedeckt
}

void Bank::KundenEinlesen()
{
    int hilf;

```

```

string name;
double help;
ifstream lesen("Kunden.txt");
lesen >> Anzahl;
Kunden = new Bankkunde[Anzahl];
for (int i = 0; i < Anzahl; i++)
{
    lesen >> hilf; Kunden[i].setKontonr(hilf);
    lesen >> name; Kunden[i].setVorname(name);
    lesen >> name; Kunden[i].setNachname(name);
    lesen >> help; Kunden[i].setLimit(help);
    lesen >> help; Kunden[i].setKontostand(help);
}
lesen.close();
}

void Bank::KundenInDatei()
{
    int hilf;
    string name;
    double help;
    ofstream schreiben("Kunden.txt");
    schreiben << endl << endl << Anzahl;
    for (int i = 0; i < Anzahl; i++)
    {
        hilf = Kunden[i].getKontonr();
        schreiben << endl << endl << hilf << ' ';
        name = Kunden[i].getVorname();
        schreiben << name << ' ';
        name = Kunden[i].getNachname();
        schreiben << name << ' ';
        help = Kunden[i].getLimit();
        schreiben << help << ' ';
        help = Kunden[i].getKontostand();
        schreiben << help;
    }
    schreiben.close();
}

int main(void)
{
    Bank Entenhausener;
    int index, index_empf, ktonr, ktonr_empf;
    double betrag;
    char wahl;
    cout << "Deine Kontonummer: ";

```

```

cin >> ktonr;
index = Entenhausener.KundeSuchen(ktonr);
if (index == -1) {
    cout << "Falsche Kontonummer" << endl;
    return 0;
}
cout << fixed << setprecision(2);
cout << "Hallo "
    << Entenhausener.Kunden[index].getVorname() << ' '
    << Entenhausener.Kunden[index].getNachname()
    << endl;
do {
    cout << endl;
    cout << "(K)ontostandsanzeige" << endl;
    cout << "(E)inzahlung" << endl;
    cout << "(A)uszahlung" << endl;
    cout << "(U)eberweisung" << endl;
    cout << "E(N)DE" << endl;
    cin >> wahl;
    switch (wahl)
    {
        case 'k':
        case 'K':
            cout << "Dein Kontostand: "
                << Entenhausener.Kunden[index].getKontostand()
                << " Euro" << endl;
            break;
        case 'e':
        case 'E':
            cout << "Betrag: ";
            do {
                cin >> betrag;
            } while (betrag < 0);
            Entenhausener.Kunden[index].einzahlen(betrag);
            break;
        case 'a':
        case 'A':
            cout << "Betrag: ";
            do {
                cin >> betrag;
            } while (betrag < 0);
            if (Entenhausener.Kunden[index].auszahlen(betrag) == 1)
                cout << "Nicht liquide!" << endl;
            else
                cout << "Sollst Du haben!" << endl;
            break;
    }
}

```

```

case 'u':
case 'U':
    cout << "Kontonummer des Empf\x84ngers: ";
    cin >> ktonr_empf;
    index_empf = Entenhausener.KundeSuchen(ktonr_empf);
    if (index_empf == -1) {
        cout << "Falsche Kontonummer" << endl;
        break;
    }
    cout << "An "
    << Entenhausener.Kunden[index_empf].getVorname()
    << ' ' <<
    Entenhausener.Kunden[index_empf].getNachname();
    cout << '!' << endl << "Betrag: ";
    do {
        cin >> betrag;
    } while (betrag < 0);
    if (Entenhausener.ueberweisen
        (index, index_empf, betrag) != 0)
        cout << '\x9A' << "berweisung konnte nicht"
        << " ausgef\x81hrt werden" << endl;
    break;
case 'n':
case 'N':
    cout << "Tsch\x81ss" << endl;
    break;
default:
    cout << "Falsche Eingabe" << endl;
} // end switch
} while (wahl != 'n' && wahl != 'N');
Entenhausener.KundenInDatei();
return 0;
} // end main()

```

Ausgabe:

```

Deine Kontonummer: 10004
Hallo Daisy Duck

(K)ontostandsanzeige
(E)inzahlung
(A)uszahlung
(U)eberweisung
E(N)DE
e
Betrag: 2500

```

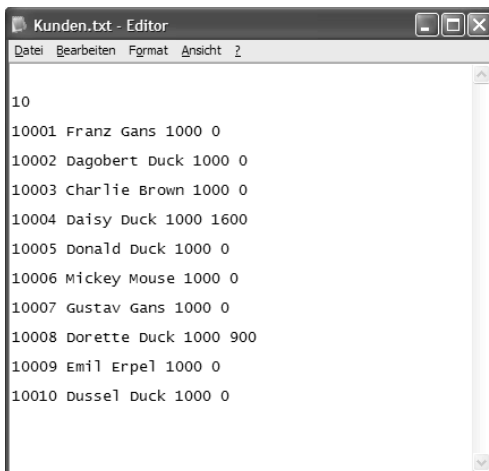
```
(K)ontostandsanzeige
(E)inzahlung
(A)uszahlung
(U)eberweisung
E(N)DE
u
Kontonummer des Empfängers: 10008
An Dorette Duck!
Betrag: 900
```

```
(K)ontostandsanzeige
(E)inzahlung
(A)uszahlung
(U)eberweisung
E(N)DE
n
Tschüss
```

## CD-ROM

Textdatei, .exe-Datei und Quellcodedatei des Beispiels finden Sie gepackt zu *Bank\_K28.zip* auf der Buch-CD im Ordner *Beispiele/K28*.

Abbildung 28.2 zeigt die nach der beschriebenen Programmausführung veränderte Textdatei *Kunden.txt*.



**Abbildung 28.2:** Die Datei *Kunden.txt* nach dem oben beschriebenen Programmablauf

Beachten Sie, dass im eigentlichen Programm, also in der Funktion `main()`, sowie in der *Bankkunde*-Klasse gegenüber der Version aus Kapitel 26, Abschnitt 26.2.1 – abgesehen

vom Wegfall des Methodenaufrufs `Entenhausener.KundenEinlesen()` in `main()` – praktisch nichts geändert werden musste, sondern allein in der Implementation der `Bank`-Klasse. Dies zeigt, dass eine objektorientierte Implementierung sehr flexibel ist.

Erwähnung verdient in diesem Zusammenhang auch die Tatsache, dass es für den Zugriff auf die einzelnen Arrayelemente keinen Unterschied macht, ob dieses mit

```
Kunden = new Bankkunde[Anzahl];
```

oder

```
Bankkunde Kunden[10];
```

erzeugt wurde. Auch im letzteren Fall handelt es sich ja um einen – konstanten – Zeiger, da `Kunden` für die Anfangsadresse des Arrays steht (Kapitel 19, Abschnitt 19.2 »Arrays und Adressen«). Ein Zugriff wie

```
Kunden[x]
```

bedeutet also in beiden Fällen »gehe an die Stelle `Kunden` plus Offset und greife auf den dortigen Inhalt zu« (wobei es sich mit `x` um einen gültigen Arrayindex handeln soll).



Während es sich bei Zeigern um ein Erbe aus der Programmiersprache C handelt, sind die Referenzen in C++ neu hinzugekommen. Dies aus gutem Grund, da sie eine allzu intensive und damit fehlerträchtige Verwendung von Zeigern einzuschränken helfen. Mit anderen Worten: An vielen Stellen im Code, an denen ansonsten Zeigervariablen eingesetzt werden müssten, kann ohne Nachteile auf die einfacher zu handhabenden Referenzen zurückgegriffen werden. Dies gilt vor allem in Bezug auf die Parameter von Funktionen und Methoden.

### Hinweis

Was die dynamische Allokation von Speicher angeht, ist die Verwendung von Zeigern jedoch unumgänglich.

## 29.1 Was sind Referenzen?

Eine Referenz ist ein alternativer Name für ein bestehendes Datenobjekt. Sie stellt gewissermaßen einen Verweis auf dieses dar. Um die Funktionalität, die dahinter steht, braucht sich der Programmierer jedoch nicht selbst zu kümmern. Diese wird praktisch vom Compiler zur Verfügung gestellt.

Deklariert wird eine Referenz, indem man das Zeichen & einem frei wählbaren Namen für die Referenz voranstellt. Außerdem muss eine Referenz initialisiert werden. Das heißt, es muss von Anfang an feststehen, auf welches Datenobjekt sie verweist. Hat man z.B. im Code eine Variable `zahl` mit

```
int zahl;
```

deklariert, so erzeugt man mit

```
int &rz = zahl;
```

eine Referenz auf diese Variable. Das bedeutet, dass `zahl` nun entweder mit `zahl` oder eben mit `rz` angesprochen werden kann:

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

```
int main(void)
{
    int zahl;
    int &rz = zahl;
    rz = 17;
    cout << zahl << endl;
    cout << rz << endl;
    return 0;
}
```

Ausgabe:

```
17
17
```

Das Zeichen & ist, ähnlich wie das bei der Deklaration von Zeigern zu verwendende Zeichen \*, im weiteren Code nicht mehr anzugeben, wenn man auf die Referenz rz bzw. auf die Variable zahl, für die sie steht, zugreifen will. Vielmehr würde ein Ausdruck

```
&rz
```

die Adresse von zahl liefern.

### Hinweis

Bei der Definition von Referenzen wird zwar dasselbe Zeichen wie beim Adressoperator – nämlich & – verwendet. Dennoch handelt es sich hierbei um zwei völlig verschiedene Dinge. Was jeweils gemeint ist, erkennt der Compiler am Kontext.

Anders als das bei Zeigern der Fall ist, kann eine Referenz ihr Verweisziel nicht ändern. Demzufolge würde, bezogen auf das Codestück

```
int a = 3, b = 56;
int &ra = a;
```

eine Anweisung wie

```
ra = b;
```

nicht etwa die Verbindung der Referenz ra mit der Variablen a lösen, sondern dieser den Wert von b, also 56, zuweisen. Auf der anderen Seite wären Anweisungen wie

```
ra = &b; // FEHLER
```

oder

```
&ra = b; // FEHLER
```

fehlerhaft.

Bei der Vereinbarung von mehreren Referenzen in einer Anweisung ist zu beachten, dass das Zeichen & – analog dem Zeichen \* bei der Deklaration von Zeigern – für jede Referenz anzugeben ist. Bezogen auf die Anweisung

```
double x = 99.5, y = 32.9;
```

erzeugt man mit

```
double &rx = x, &ry = y;
```

je eine Referenz auf die Variable x und eine auf die Variable y, mit

```
double &rx = x, ry = y;
```

jedoch eine Referenz auf x sowie eine Variable ry vom Typ double, die mit dem Wert von y initialisiert wird.

### Hinweis

Im Übrigen ist es auch möglich, mehrere Referenzen, die auf ein und dieselbe Variable verweisen, zu vereinbaren:

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

int main(void)
{
    double zahl = 45.11;
    double &rzahl = zahl, &rz = zahl;
    cout << rzahl << endl;
    cout << rz << endl;
    return 0;
}
```

Ausgabe:

```
45.11
45.11
```

Beachten Sie, dass eine Referenz, anders als eine Variable, keinen Speicher beansprucht, also tatsächlich kein Datenobjekt ist. Referenzen auf Referenzen bzw. Zeiger auf Referenzen sind daher nicht möglich, da eine Referenz ja keine Adresse hat.

## 29.2 Referenzen als Parameter von Funktionen

Als Funktionsparameter leisten Referenzen dasselbe wie Zeiger, lassen sich allerdings entsprechend einfacher verwenden. Hat man etwa eine Funktion `dreifach()` wie folgt eingerichtet:

```
void dreifach(int &z)
{
    z = 3 * z;
}
```

so verändert diese den Parameter des Aufrufers entsprechend, da sie eben einen Verweis auf diesen erhält.

### Hinweis

Was hier und im Weiteren über Referenzen in Verbindung mit Funktionen gesagt wird, gilt natürlich in gleicher Weise auch für Methoden.

Beachten Sie, dass `dreifach()` beim Aufruf der Verweis auf eine Integervariable zu übergeben ist, und zwar nicht etwa mit dem Adressoperator

```
dreifach(&Bezeichner) // FEHLER
```

sondern in der Form

```
dreifach(Bezeichner)
```

Wie gesagt, sorgt der Compiler dafür, dass die Funktion beim Aufruf tatsächlich einen Verweis und nicht den bloßen Wert der Variablen erhält.

### Hinweis

Es sei darauf hingewiesen, dass beim Aufruf

```
dreifach(Bezeichner)
```

eine zu dieser Funktion lokale Variable `z` angelegt wird, was somit der Anweisung

```
int &z = Bezeichner;
```

entspricht (wobei es sich mit *Bezeichner* natürlich um den Namen einer Integervariablen handelt).

**Hinweis**

Aus den genannten Gründen ist ein Aufruf wie

```
dreifach(15) // FEHLER
```

natürlich fehlerhaft.

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

void dreifach(int &z)
{
    z = 3 * z;
}

int main(void)
{
    int x = 6;
    dreifach(x);
    cout << x;
    return 0;
}
```

Ausgabe: 18

Dem Gesagten zufolge lässt sich die Funktion `tausch()` des Beispiels aus Kapitel 28, Abschnitt 28.3 »Zeiger als Parameter von Funktionen«, etwas bequemer mit Referenzen als Parameter einrichten:

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

void tausch(double &ra, double &rb)
{
    double tmp;
    tmp = ra;
    ra = rb;
    rb = tmp;
}

int main(void)
{
    double zahl1 = 15.1, zahl2 = 75.4;
    cout << "Vor dem Tausch:" << endl;
    cout << "Wert von zahl1: " << zahl1 << endl;
    cout << "Wert von zahl2: " << zahl2 << endl;
}
```

```

tausch(zahl1, zahl2);
cout << "Nach dem Tausch:" << endl;
cout << "Wert von zahl1: " << zahl1 << endl;
cout << "Wert von zahl2: " << zahl2 << endl;
return 0;
}

```

Ausgabe:

```

Vor dem Tausch:
Wert von zahl1: 15.1
Wert von zahl2: 75.4
Nach dem Tausch:
Wert von zahl1: 75.4
Wert von zahl2: 15.1

```

## 29.3 Referenzen als Rückgabewerte von Funktionen

Im Beispiel des Abschnitts 28.5 »Der this-Zeiger von Methoden« im letzten Kapitel haben wir die Methode `mehrGehalt()` der Klasse `Mitarbeiter` so eingerichtet, dass sie in der Form

```
Objektname.mehrGehalt()->mehrGehalt()->mehrGehalt()
```

für ein Objekt mehrmals aufgerufen werden kann. Erreicht wird dies, indem aus der Methode der `this`-Zeiger, also ein Zeiger auf das aktuelle Objekt, zurückgegeben wird:

```

Mitarbeiter* mehrGehalt()
{
    gehalt += gehalt * 0.1;
    return this;
}

```

Lässt man die Methode nun eine Referenz auf das aktuelle Objekt zurückgeben, so kann sie noch bequemer in der Form

```
Objektname.mehrGehalt().mehrGehalt().mehrGehalt()
```

aufgerufen werden. In diesem Fall steht ein Ausdruck wie

```
Objektname.mehrGehalt()
```

oder auch

```
Objektname.mehrGehalt().mehrGehalt()
```

für das – modifizierte – Objekt:

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

class Mitarbeiter
{
public:
    double gehalt;
    // ...
    Mitarbeiter()
    {gehalt = 2500.0;}
    Mitarbeiter& mehrGehalt()
    {
        gehalt += gehalt * 0.1;
        return *this;
    }
    // ...
};

int main(void)
{
    Mitarbeiter Donald;
    cout << "Gehalt von Donald: " << Donald.gehalt << endl;
    Donald.mehrGehalt().mehrGehalt().mehrGehalt();
    cout << "Gehalt von Donald: " << Donald.gehalt << endl;
    return 0;
}
```

Ausgabe:

```
Gehalt von Donald: 2500
Gehalt von Donald: 3327.5
```

Beachten Sie, dass der dereferenzierte this-Zeiger

```
*this
```

das Objekt selbst darstellt.





# 30 Vererbung

In C++ besteht die Möglichkeit, eine Klasse von einer oder mehreren anderen Klassen erben zu lassen, wobei man die Klasse, die erbt, als »abgeleitete Klasse«, und diejenigen Klassen, von denen diese Klasse erbt, als »Basisklassen« bezeichnet. Wenn eine Klasse von einer anderen erbt, bedeutet dies, dass sie die Elemente der Basisklasse erhält, ohne dass diese in der neuen Klasse definiert werden müssten. Über den Zugriff auf die geerbten Elemente in der abgeleiteten Klasse sowie außerhalb derselben entscheidet jedoch die Art der Vererbungsbeziehung. Dabei unterscheidet man zwischen `public`-, `protected`- und `private`-Vererbung.

Eine Vererbungsbeziehung zwischen Klassen wird man vor allem dann einrichten, wenn es sich in der Realität so verhält, dass ein Vertreter einer Gruppe gleichzeitig ein Vertreter einer anderen ist: Ein Tier ist ein Lebewesen, ein Fahrzeug ist ein Auto usw. Demgemäß würde man eine Klasse `Tier` von einer Klasse `Lebewesen` und eine Klasse `Auto` von einer Klasse `Fahrzeug` ableiten. Nicht etwa umgekehrt, da schließlich ein Fahrzeug nicht zwangsläufig ein Auto und nicht jedes Lebewesen ein Tier ist.

## 30.1 Klassen von Basisklassen ableiten

Angenommen, Sie hätten zwei Klassen `A_Klasse` und `B_Klasse`, die wie folgt definiert sind:

```
class A_Klasse
{
    ...
};

class B_Klasse
{
    ...
};
```

so könnten Sie von einer der beiden eine dritte Klasse ableiten, indem Sie nach dem Bezeichner der Klasse, getrennt durch einen Doppelpunkt, den Namen der Basisklasse angeben.

Um also beispielsweise von der Klasse `A_Klasse` eine Klasse `C_Klasse` abzuleiten, definieren Sie diese wie folgt:

```
class C_Klasse : A_Klasse
{
    ...
};
```

### Achtung

Eine Basisklasse muss im Code oberhalb einer von ihr abgeleiteten Klasse stehen, da Erstere bei der Definition der Letzteren ja bekannt sein muss.

Die Klasse `C_Klasse` besitzt somit die Elemente, die explizit innerhalb der Klasse definiert bzw. deklariert sind, sowie alle Elemente – Attribute wie Methoden – der `A_Klasse`. Letztere stellen gewissermaßen das Erbe der `A_Klasse` dar. Will man die `C_Klasse` nicht von der `A_Klasse`, sondern von der `B_Klasse` ableiten, so schreibt man dementsprechend:

```
class C_Klasse : B_Klasse
{
    ...
};
```

Im Gegensatz zu vielen anderen Programmiersprachen ist in C++ auch eine Mehrfachvererbung möglich. Das heißt, man kann eine Klasse nicht nur von einer, sondern von mehreren Basisklassen erben lassen. In diesem Fall gibt man nach dem Doppelpunkt eine durch Komma getrennte Liste mit den Namen der Basisklassen an. Um die Klasse `C_Klasse` sowohl von der `A_Klasse` als auch von der `B_Klasse` abzuleiten, definiert man sie also wie folgt:

```
class C_Klasse : B_Klasse, A_Klasse
{
    ...
};
```

Die Reihenfolge, in der die Basisklassen genannt sind, hat dabei keine Bedeutung. Mit dieser Definition besitzt die `C_Klasse` nun die Eigenschaften und Methoden, die innerhalb ihres Blocks definiert bzw. deklariert sind, sowie alle Elemente<sup>1</sup> der `A_Klasse` und der `B_Klasse`.

---

1 Von der Vererbung ausgenommen sind allerdings Konstruktoren und Destruktoren.

## 30.2 Zugriff auf die geerbten Elemente einer Basisklasse

Es verhält sich so, dass eine abgeleitete Klasse bezüglich ihrer geerbten Elemente nicht automatisch die Zugriffsrechte von der Basisklasse übernimmt. Ein z.B. in der Basisklasse als `public` spezifiziertes Element ist demnach nicht ohne weiteres in der abgeleiteten Klasse ebenfalls öffentlich zugänglich. Mit anderen Worten: Es steht nicht von vornherein fest, dass dieses Element von Instanzen der abgeleiteten Klasse auf direktem Wege angesprochen werden kann. Dies hängt von der Art der Vererbungsbeziehung ab. Diese kann `private`, `protected` oder `public` sein. Festgelegt wird dies durch einen entsprechenden Spezifizierer, der in der Definition der abgeleiteten Klasse vor dem Namen der Basisklasse anzugeben ist:

```
class C_Klasse : public B_Klasse, protected A_Klasse
{
    ...
};
```

Hier handelt es sich im Verhältnis von `C_Klasse` zur Basisklasse `B_Klasse` um eine `public`-Vererbung und im Verhältnis zur Basisklasse `A_Klasse` um eine `protected`-Vererbung (auf die Folgen werden wir gleich zu sprechen kommen).

Per Default ist die Vererbungsbeziehung zu Basisklassen bei Klassen, die mit dem Schlüsselwort `struct` definiert sind, `public` und bei Klassen, die mit dem Schlüsselwort `class` definiert sind, `private`. Bei den im letzten Abschnitt genannten Ableitungen handelt es sich somit durchweg um `private`-Vererbungen.

Will man bezüglich der Klasse `C_Klasse` zur Basisklasse `B_Klasse` eine `public`- und im Verhältnis zur `A_Klasse` eine `private`-Vererbung einrichten, so schreibt man demnach

```
class C_Klasse : public B_Klasse, A_Klasse
{
    ...
};
```

oder explizit

```
class C_Klasse : public B_Klasse, private A_Klasse
{
    ...
};
```

### 30.2.1 private-Vererbung

Was die verschiedenen Vererbungsarten angeht, so gilt die Regel, dass die Zugriffsbeschränkung von geerbten Elementen in der abgeleiteten Klasse nie geringer ist als der Ableitungsmodus bezüglich der Basisklasse. Da die Zugriffsbeschränkung von `public`

über `protected` bis `private` immer strenger wird bedeutet dies für eine `private`-Ableitung, dass geerbte Elemente, die in der Basisklasse als `public` oder `protected` deklariert sind, in der abgeleiteten Klasse als `private` gelten.

### Der Zugriffsspezifizierer »protected«

Man kann Elemente einer Klasse neben `public` und `private` auch als `protected` deklarieren. Was den Zugriff bezüglich dieser Klasse angeht, verhalten sich `protected`-Elemente wie `private`-Elemente. Sie können innerhalb der Klasse – also von deren Methoden – verwendet werden, jedoch nicht von Objekten der Klasse. Von daher macht es hinsichtlich der Programmierung mit den Objekten keinen Unterschied, ob ein Klassenelement als `private` oder `protected` definiert ist, wohl aber im Hinblick auf eine Ableitung von dieser Klasse, wie wir im Folgenden sehen werden.

Ein `public`- oder `protected`-Element der Basisklasse verhält sich also in einer `private`-abgeleiteten Klasse so, als ob es in dieser als `private` spezifiziert wäre:

```
class Fahrzeug
{
protected:
    int fahrGeschw;
    // ...
public:
    void beschleunigen(){fahrGeschw += 10;}
    void geschwVermindern(){fahrGeschw -= 10;}
    int getFahrGeschw(){return fahrGeschw;}
    void setFahrGeschw(int fg){fahrGeschw = fg;}
    // ...
};

class Auto : private Fahrzeug
{
private:
    int raeder;
    // ...
public:
    void bremsen()
    {fahrGeschw = 0;}; // OK
    // ...
};
```

Da z.B. das Attribut `fahrGeschw` in der Basisklasse `Fahrzeug` als `protected` spezifiziert ist, besitzt es in der von `Fahrzeug` per `private` abgeleiteten Klasse `Auto` den Status `private`. Daher kann es von der Methode `bremsen()`, die innerhalb der Klasse `Auto` definiert ist, verwendet werden. Da diese Methode wiederum `public` ist, kann auf sie mit einer Instanz der Klasse `Auto` zugegriffen werden.

```
Auto meinAuto;  
meinAuto.bremsen(); // OK
```

Allerdings ist der Zugriff auf die Methoden `beschleunigen()`, `geschwVermindern()`, `getFahrGeschw()`, `setFahrGeschw()` mit einem Objekt der Klasse `Auto` nicht möglich, da diese Methoden in dieser Klasse allesamt `private` sind (das Gleiche gilt natürlich für die Eigenschaft `fahrGeschw`).

```
meinAuto.beschleunigen(); // FEHLER, da  
// beschleunigen() private in Auto
```

Selbstverständlich ist es möglich, in der Klasse `Auto` zusätzlich entsprechende `public`-Methoden zu definieren, die gegebenenfalls auf die genannten Methoden zugreifen:

```
class Fahrzeug  
{  
protected:  
    int fahrGeschw;  
    // ...  
public:  
    void beschleunigen(){fahrGeschw += 10;}  
    // ...  
};  
  
class Auto : private Fahrzeug  
{  
    // ...  
public:  
    void machSchneller()  
    {beschleunigen();}  
    // ...  
};
```

Da die Methode `machSchneller()` der `Auto`-Klasse `public` ist, kann sie von einer Instanz dieser Klasse verwendet werden:

```
meinAuto.machSchneller();
```

### Hinweis

Natürlich hätte man die Methode auch wie folgt einrichten können:

```
void machSchneller()  
{fahrGeschw += 10;}
```

Für den Fall, dass in der Klasse Fahrzeug ein Konstruktor definiert ist, der das Attribut `fahrGeschw` initialisiert, wirkt sich dies nicht auf die abgeleitete Klasse `Auto` aus, da Konstruktoren – wie Destruktoren – nicht mitvererbt werden. Es lässt sich aber für die Klasse `Auto` ein eigener Konstruktor definieren, der für die Initialisierung von `fahrGeschw` sorgt. Schließlich hat dieser, wie alle Methoden der Klasse `Auto`, Zugriff auf das geerbte Element `fahrGeschw`:

```
class Fahrzeug
{
protected:
    int fahrGeschw;
    // ...
public:
    void beschleunigen(){fahrGeschw += 10;}
    void geschwVermindern(){fahrGeschw -= 10;}
    int getFahrGeschw(){return fahrGeschw;}
    void setFahrGeschw(int fg){fahrGeschw = fg;}
    // ...
};

class Auto : private Fahrzeug
{
private:
    int raeder;
    // ...
public:
    Auto() {fahrGeschw = 0;}
    void bremsen()
    {fahrGeschw = 0;}
    // ...
};
```

Nach der eingangs genannten Regel könnte man zu dem Schluss kommen, dass sich auch `private`-Elemente der Basisklasse in einer abgeleiteten Klasse ebenfalls in der gewohnten Weise verhalten. Dies trifft jedoch nicht zu. Solche Elemente gelten zwar in der abgeleiteten Klasse ebenfalls als `private`. Tatsächlich ist es jedoch so, dass auf geerbte Elemente, die in der Basisklasse den Status `private` besitzen, in der abgeleiteten Klasse nicht zugegriffen werden kann, weder von außerhalb noch von Methoden der Klasse.

Definiert man also das Element `fahrGeschw` in der Klasse `Fahrzeug` als `private`, so sind die Zugriffe auf dieses innerhalb der Methode `bremsen()` sowie im Konstruktor der Klasse `Auto` fehlerhaft:

```
class Fahrzeug
{
private:
    int fahrGeschw;
    // ...
```

```
public:
    void beschleunigen(){fahrGeschw += 10;}
    // ...
};

class Auto : private Fahrzeug
{
private:
    int raeder;
    // ...
public:
    Auto() {fahrGeschw = 0;} // FEHLER
    void bremsen()
    {fahrGeschw = 0;} // FEHLER
    // ...
};
```

Im Übrigen gilt dies unabhängig von der Ableitungsart. Auf geerbte private-Elemente – gemeint sind solche, die in der Basisklasse als private spezifiziert sind – kann in einer abgeleiteten Klasse grundsätzlich nicht zugegriffen werden.

### Hinweis

Es sei darauf hingewiesen, dass diese Tatsache den Programmierer in keiner Weise einschränkt. Schließlich steht es ihm ja frei, das entsprechende Element in der Basis-klassse als protected zu definieren.

Dass geerbte private-Elemente dennoch Bestandteil einer abgeleiteten Klasse sind, lässt sich schon daran erkennen, dass der Zugriff auf geerbte public- oder protected-Metho-den auch dann erlaubt ist, wenn diese im Methodenrumpf auf private-Elemente der Basisklasse zugreifen:

```
class Fahrzeug
{
private:
    int fahrGeschw;
    // ...
public:
    void beschleunigen(){fahrGeschw += 10;}
    void geschwVermindern(){fahrGeschw -= 10;}
    int getFahrGeschw(){return fahrGeschw;}
    void setFahrGeschw(int fg){fahrGeschw = fg;}
    // ...
};
```

```

class Auto : private Fahrzeug
{
private:
    int raeder;
    // ...
public:
    Auto() {setFahrGeschw(0);} // OK
    void bremsen()
    {setFahrGeschw(0);} // OK
    // ...
};

```

### 30.2.2 protected-Vererbung

Gemäß der weiter oben genannten Regel, nach der die Form der Ableitung den Status der geerbten Elemente in der abgeleiteten Klasse nach oben hin einschränkt, werden im Zuge einer protected-Vererbung public-Elemente der Basisklasse in der abgeleiteten Klasse zu protected. Dagegen behalten protected- und private-Elemente ihren Status bei. Für Letztere gilt allerdings die oben genannte Zugriffsbeschränkung:

```

#include <iostream>
using namespace std;

class Fahrzeug
{
private:
    int hoechstGeschw;
protected:
    int fahrGeschw;
    // ...
public:
    void beschleunigen()
    {
        if (fahrGeschw + 10 <= hoechstGeschw)
            fahrGeschw += 10;
    }
    void geschwVermindern(){fahrGeschw -= 10;}
    int getFahrGeschw(){return fahrGeschw;}
    void setFahrGeschw(int fg)
    {
        if (fg <= hoechstGeschw)
            fahrGeschw = fg;
    }
    int getHoechstGeschw(){return hoechstGeschw;}
    void setHoechstGeschw(int hg){hoechstGeschw = hg;}
    // ...
};

```



```
class Auto : protected Fahrzeug
{
private:
    int raeder;
    // ...
public:
    Auto(){fahrGeschw = 0; setHoechstGeschw(200);}
    void bremsen()
    {fahrGeschw = 0;}
    void machSchneller()
    {beschleunigen();}
    void zeigGeschw()
    {cout << fahrGeschw << " km/h" << endl;}
    // ...
};

int main(void)
{
    Auto myCar;
    myCar.zeigGeschw();
    myCar.machSchneller();
    myCar.zeigGeschw();
    myCar.machSchneller();
    myCar.zeigGeschw();
    myCar.bremsen();
    myCar.zeigGeschw();
    return 0;
}
```

Ausgabe:

```
0 km/h
10 km/h
20 km/h
0 km/h
```

Bezogen auf obiges Beispiel besitzt die Klasse Auto, die von der Klasse Fahrzeug per `protected` abgeleitet ist, folgende Elemente:

Attribute mit dem Zugriffsstatus `private` sind `raeder` und `hoechstGeschw`. Letzteres ist von der Klasse Fahrzeug geerbt. Da es in dieser den Status `private` besitzt, kann es weder von Objekten noch von Methoden der Klasse Auto unmittelbar angesprochen werden. Allerdings wird es innerhalb der geerbten Methoden `beschleunigen()`, `setFahrGeschw()`, `getHoechstGeschw()` und `setHoechstGeschw()` verwendet. Da diese Methoden den Zugriffsstatus `protected` besitzen, dürfen sie innerhalb der Klasse Auto verwendet werden.

### Hinweis

Gemeint sind damit natürlich die – von der Klasse Fahrzeug geerbten – Methoden der Klasse Auto. Die Klasse Fahrzeug besitzt ja gleichnamige Methoden mit dem Zugriffsstatus public.

So erfolgt z.B. über die Konstruktormethode, in der `setHoechstGeschw()` aufgerufen wird, gewissermaßen ein indirekter Zugriff auf das Element `hoechstGeschw` der Klasse Auto:

```
class Fahrzeug
{
private:
    int hoechstGeschw;
    // ...
public:
    void setHoechstGeschw(int hg){hoechstGeschw = hg;}
    // ...
};

class Auto : protected Fahrzeug
{
    // ...
public:
    Auto(){fahrGeschw = 0; setHoechstGeschw(200);}
    // ...
};

int main(void)
{
    Auto myCar; // ruft Konstruktor auf
    // ...
    return 0;
}
```

### Hinweis

Man beachte, dass sowohl die Klasse Fahrzeug als auch die Klasse Auto je ein Attribut `hoechstGeschw` besitzt. Nur dass das zur Auto-Klasse gehörige allein über den gezeigten Umweg angesprochen werden kann.

Weitere Bestandteile der Klasse Auto sind außer dem Konstruktor die public-Methoden `bremsen()`, `machSchneller()` und `zeigGeschw()`, die in der Klasse definiert sind, sowie die geerbten Methoden `beschleunigen()`, `geschwVermindern()`, `getFahrGeschw()`, `setFahrGeschw()`, `getHoechstGeschw()` und `setHoechstGeschw()`. Letztere besitzen den Zugriffsstatus

protected, was auch für das geerbte Datenelement `fahrGeschw` gilt. Folglich würde man eine entsprechende Klasse ohne Vererbungsbeziehung etwa wie folgt definieren:

```
class Auto
{
private:
    int hoechstGeschw;
    int raeder;
    // ...
protected:
    int fahrGeschw;
    void beschleunigen()
    {
        if (fahrGeschw + 10 <= hoechstGeschw)
            fahrGeschw += 10;
    }
    void geschwVermindern(){fahrGeschw -= 10;}
    int getFahrGeschw(){return fahrGeschw;}
    void setFahrGeschw(int fg)
    {
        if (fg <= hoechstGeschw)
            fahrGeschw = fg;
    }
    int getHoechstGeschw(){return hoechstGeschw;}
    void setHoechstGeschw(int hg){hoechstGeschw = hg;}
public:
    Auto(){fahrGeschw = 0; setHoechstGeschw(200);}
    void bremsen()
    {fahrGeschw = 0;}
    void machSchneller()
    {beschleunigen();}
    void zeigGeschw()
    {cout << fahrGeschw << " km/h" << endl;}
    // ...
};
```

### Hinweis

Wobei diese Definition die in der vorherigen Version bestehenden Einschränkungen bezüglich des Datenelements `hoechstGeschw` allerdings nicht wiedergibt.

### 30.2.3 public-Vererbung

Wenn eine Klasse von einer anderen public-abgeleitet ist, dann entspricht der Zugriffsstatus der geerbten Elemente dem der Basisklasse, wobei für private-Elemente die genannten Abstriche zu machen sind<sup>1</sup>:

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

class Fahrzeug
{
private:
    int hoechstGeschw;
protected:
    int fahrGeschw;
    // ...
public:
    void beschleunigen()
    {
        if (fahrGeschw + 10 <= hoechstGeschw)
            fahrGeschw += 10;
    }
    void geschwVermindern(){fahrGeschw -= 10;}
    int getFahrGeschw(){return fahrGeschw;}
    void setFahrGeschw(int fg)
    {
        if (fg <= hoechstGeschw)
            fahrGeschw = fg;
    }
    int getHoechstGeschw(){return hoechstGeschw;}
    void setHoechstGeschw(int hg){hoechstGeschw = hg;}
    // ...
};

class Auto : public Fahrzeug
{
private:
    int raeder;
    // ...
public:
    Auto(){fahrGeschw = 0; setHoechstGeschw(200);}
    void bremsen()
    {fahrGeschw = 0;}
};
```

<sup>1</sup> Auf geerbte Elemente, die in der Basisklasse als private deklariert sind, kann die abgeleitete Klasse nicht direkt, sondern nur über geerbte public- oder protected-Methoden zugreifen.

```
void zeigGeschw()
{cout << fahrGeschw << " km/h" << endl;}
// ...
};

int main(void)
{
    Auto myCar;
    myCar.zeigGeschw();
    myCar.beschleunigen();
    myCar.zeigGeschw();
    myCar.beschleunigen();
    myCar.zeigGeschw();
    myCar.beschleunigen();
    myCar.zeigGeschw();
    return 0;
}
```

Ausgabe:

```
0 km/h
10 km/h
20 km/h
30 km/h
```

Die von der Klasse Fahrzeug übernommenen public-Methoden können nun von Instanzen der Klasse Auto verwendet werden:

```
Auto BMW;
BMW.setHoechstGeschw(250);
```

Die public-Methode

```
void machSchneller()
{beschleunigen();}
```

kann somit entfallen, da nunmehr die Methode beschleunigen() für den direkten Zugriff von außen zur Verfügung steht.

## 30.3 Überschreiben von Methoden

Falls man in einer abgeleiteten Klasse eine Methode definiert, deren Namen und Signatur einer von der Basisklasse geerbten Methode gleicht, so spricht man vom »Überschreiben« dieser Methode. Sie wird dadurch überdeckt. Das heißt, die – überschriebene – geerbte Methode kann innerhalb der abgeleiteten Klasse oder von Objekten dieser Klasse nur mit Bereichsauflösung angesprochen werden.

## Referenz

Zur Signatur einer Funktion bzw. Methode siehe Kapitel 20, Abschnitt 20.5 »Überladen von Funktionen«.

Im folgenden Beispiel ist für die Klasse `Auto` eine zusätzliche Methode `beschleunigen()` definiert. Somit besitzt diese Klasse zwei gleichnamige Methoden, die zudem in ihrer Signatur übereinstimmen. Das heißt, die in der Klasse `Auto` deklarierte Methode verdeckt die von der Klasse `Fahrzeug` geerbte. Dies bedeutet wiederum, dass der Zugriff auf Letztere mit dem qualifizierten Namen erfolgen muss:

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

class Fahrzeug
{
private:
    int hoechstGeschw;
protected:
    int fahrGeschw;
    // ...
public:
    void beschleunigen()
    {
        if (fahrGeschw + 10 <= hoechstGeschw)
            fahrGeschw += 10;
    }
    void geschwVermindern(){fahrGeschw -= 10;}
    int getFahrGeschw(){return fahrGeschw;}
    void setFahrGeschw(int fg)
    {
        if (fg <= hoechstGeschw)
            fahrGeschw = fg;
    }
    int getHoechstGeschw(){return hoechstGeschw;}
    void setHoechstGeschw(int hg){hoechstGeschw = hg;}
    // ...
};

class Auto : public Fahrzeug
{
private:
    int raeder;
    // ...
public:
    Auto(){fahrGeschw = 0; setHoechstGeschw(200);}
```

```

    void bremsen()
    {fahrGeschw = 0;}
    void zeigGeschw()
    {cout << fahrGeschw << " km/h" << endl;}
    void beschleunigen();
    // ...
};

void Auto::beschleunigen()
{
    if (fahrGeschw + 50 <= getHoechstGeschw())
        fahrGeschw += 50;
    else
        fahrGeschw = getHoechstGeschw();
}

int main(void)
{
    Auto einBMW;
    einBMW.zeigGeschw();
    einBMW.beschleunigen();
    einBMW.zeigGeschw();
    einBMW.Fahrzeug::beschleunigen();
    einBMW.zeigGeschw();
    return 0;
}

```

Ausgabe:

```

0 km/h
50 km/h
60 km/h

```

Im Beispiel bewirkt die Anweisung

```
einBMW.beschleunigen();
```

den Zugriff der Instanz einBMW auf deren innerhalb der Auto-Klasse deklarierte Methode, während mit

```
einBMW.Fahrzeug::beschleunigen();
```

die gleichnamige geerbte Methode aufgerufen wird. Erstere beschleunigt um 50 km/h, Letztere um 10 km/h, wie die Ausgabe zeigt.

Allerdings verhält es sich nicht viel anders, wenn man in der abgeleiteten Klasse gleichnamige Methoden definiert, die sich in ihrer Parameterliste gegenüber den von der Basisklasse geerbten unterscheiden (nur dass man in diesem Fall nicht vom Überschrei-

ben spricht). Die geerbten Methoden werden dadurch nicht etwa überladen, sondern in der gleichen Weise wie überschriebene Methoden überdeckt. Das heißt im Ergebnis: Der unqualifizierte Zugriff auf eine geerbte Methode, die den gleichen Namen trägt wie eine in der abgeleiteten Klasse definierte bzw. deklarierte, ist grundsätzlich nicht möglich. Ist z.B. eine Methode `xyz()` in der Klasse A zehnfach überladen und man definiert eine gleichnamige Methode in einer Klasse B, die von A erbt, so verdeckt man dadurch alle zehn geerbten Methoden.

Zur Demonstration soll die Methode `beschleunigen()` für die Klasse `Auto` mit einem Parameter definiert werden:

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

class Fahrzeug
{
private:
    int hoechstGeschw;
protected:
    int fahrGeschw;
    // ...
public:
    void beschleunigen()
    {
        if (fahrGeschw + 10 <= hoechstGeschw)
            fahrGeschw += 10;
    }
    void geschwVermindern(){fahrGeschw -= 10;}
    int getFahrGeschw(){return fahrGeschw;}
    void setFahrGeschw(int fg)
    {
        if (fg <= hoechstGeschw)
            fahrGeschw = fg;
    }
    int getHoechstGeschw(){return hoechstGeschw;}
    void setHoechstGeschw(int hg){hoechstGeschw = hg;}
    // ...
};

class Auto : public Fahrzeug
{
private:
    int raeder;
    // ...
public:
    Auto(){fahrGeschw = 0; setHoechstGeschw(200);}
    void bremsen()
    {fahrGeschw = 0;}
};
```



```
void zeigGeschw()
{cout << fahrGeschw << " km/h" << endl;}
void beschleunigen(int b);
// ...
};

void Auto::beschleunigen(int b)
{
    if (fahrGeschw + b <= getHoechstGeschw())
        fahrGeschw += b;
    else
        fahrGeschw = getHoechstGeschw();
}

int main(void)
{
    Auto einBMW;
    einBMW.zeigGeschw();
    einBMW.beschleunigen(70);
    einBMW.zeigGeschw();
    einBMW.Fahrzeug::beschleunigen();
    einBMW.zeigGeschw();
    return 0;
}
```

Ausgabe:

```
0 km/h
70 km/h
80 km/h
```

Ein Aufruf

```
einBMW.beschleunigen(); // FEHLER
```

würde nicht etwa den Zugriff auf die geerbte parameterlose Variante der Methode bedeuten, sondern eine Fehlermeldung des Compilers nach sich ziehen.



# 31

## Überladen von Operatoren

Wie Ihnen bekannt ist, können Operationen auf den Datenelementen eines Objekts durchgeführt werden, grundsätzlich aber nicht auf den Objekten als solchen. Angenommen, es existieren zwei Objekte A und B einer Klasse Punkt, die die Datenelemente x und y besitzen. Dann bedeutet dies, dass zwar eine Addition der Koordinaten wie folgt möglich ist:

```
A.x + B.x  
A.y + B.y
```

(sofern x und y den Zugriffsstatus `public` innehaben), nicht aber die entsprechende Operation mit den beiden Objekten als solchen:

```
A + B // FEHLER
```

da der Zugriff auf ein Objekt grundsätzlich elementweise zu erfolgen hat. Einzige Ausnahme ist die direkte Zuweisung

```
A = B;
```

bzw.

```
B = A;
```

mit der die Werte aller Datenelemente der einen Instanz den Datenelementen der anderen Instanz zugewiesen werden.

Nun ist es möglich, einige der Ihnen bekannten Operatoren zu überladen. Das heißt, man kann für eine Klasse Methoden – »Operatorfunktionen« oder Operatormethoden« genannt – definieren, die die entsprechenden Operationen in der Form

```
A + B  
A - B  
A * B  
...
```

oder eben

```
B + A  
B - A  
B * A
```

usw. erlauben.

**Hinweis**

Die Bezeichnung »Operatorfunktion« lehnt sich an den Ausdruck »Elementfunktion« an, der ja ebenfalls für Methoden einer Klasse gebräuchlich ist.

## 31.1 Welche Operatoren lassen sich überladen?

In Tabelle 31.1 sehen Sie eine Zusammenstellung der Ihnen bekannten Operatoren, die in der beschriebenen Weise überladen werden können:

+	-	*	/	%	++
--	=	+=	-=	*=	/=
%=	<	<=	>	>=	==
!=	!	&&		>>	<<
()	[]	->	&	->*	,

**Tabelle 31.1:** Überladbare Operatoren

Wie gesagt, ist für den Zuweisungsoperator (=) eine entsprechende Operatormethode für jede Klasse bereits vordefiniert. Die Operatoren `new` und `delete` lassen sich zwar ebenfalls überladen, nehmen aber eine gewisse Sonderstellung ein, worauf hier und im Folgenden nicht näher eingegangen werden soll, zumal deren Überladung in der Praxis keine allzu bedeutende Rolle spielt. Nicht überladen werden kann der Bereichsauflösungsoperator (::), der Operator für die Elementauswahl, genannt Punktoperator (.), die Elementauswahl mit .\*, der `sizeof`-Operator und der Bedingungsoperator (?:).

Bezüglich überladener Operatoren gilt:

- Sie beziehen sich stets auf Objekte einer bestimmten Klasse. Das bedeutet, dass eine Operatormethode grundsätzlich nicht statisch sein darf.
- Priorität und Assoziativität eines überladenen Operators bleiben erhalten.
- Ebenso unverändert bleibt die Zahl seiner Operanden. Ein unärer Operator bleibt also ein unärer Operator. Das Gleiche gilt für einen binären Operator.

## 31.2 Definition von Operatormethoden

Gegeben sei die folgende Klasse `Punkt`. Die Klasse enthält einen Standardkonstruktor, der die Koordinaten eines Punktes mit den Werten 0, 0 initialisiert, sowie einen Konstruktor, für den entsprechende Parameter definiert sind. Für die privaten Datenelemente sind `get`- und `set`-Methoden definiert. Außerdem enthält die Klasse eine Methode, die die Koordinaten eines Punktes ausgibt:

```

class Punkt
{
private:
    int x;
    int y;
public:
    Punkt() {x = 0; y = 0;}
    Punkt(int xko, int yko)
        {x = xko; y = yko;}
    void setX(int xko) {x = xko;}
    int getX() {return x;}
    void setY(int yko) {y = yko;}
    int getY() {return y;}
    void zeigePunkt();
};

void Punkt::zeigePunkt()
{
    cout << "x-Koordinate: " << x << endl;
    cout << "y-Koordinate: " << y << endl;
}

```

### Hinweis

Es sei darauf hingewiesen, dass man statt der beiden Konstruktoren auch einen einzigen definieren und für dessen Parameter entsprechende Vorgabeargumente setzen könnte:

```

Punkt(int xko = 0, int yko = 0)
{
    x = xko;
    y = yko;
}

```

In diesem Fall kann der Konstruktor beim Instanzieren eines Punktes mit einem, zwei oder gar keinem Argument aufgerufen werden. Dabei stellt sich allein die Frage, ob man es Programmierern, die die Klasse Punkt nutzen, erlauben möchte, an den Konstruktor der Klasse nur einen einzigen Wert zu übergeben.

## 31.2.1 Überladen von binären Operatoren

Für die Klasse Punkt soll nun der binäre »+«-Operator überladen werden, sodass eine Operation in der Form

```
PunktEins + PunktZwei
```

bzw.

```
PunktZwei + PunktEins
```

möglich ist, wobei es sich bei `PunktEins` und `PunktZwei` um `Punkt`-Objekte handeln soll.

Die Definition einer Operatormethode unterscheidet sich an und für sich nicht von der einer gewöhnlichen nicht statischen Methode, abgesehen davon, dass einige weitere Vorgaben zu beachten sind:

- Die Anzahl der Parameter ist vorgeschrieben. Für eine Operatormethode, die einen binären Operator überlädt, gilt: Sie muss mit genau einem formalen Parameter definiert sein.
- Der Name einer Operatormethode setzt sich zusammen aus dem Schlüsselwort `operator` und dem Symbol für den zu überladenden Operator.

Der Name einer Operatormethode, die den `»+«`-Operator überlädt, ist also

```
operator+
```

Rückgabewert unserer Operatormethode soll ein Objekt der Klasse `Punkt` sein. Wir wollen den binären `»+«`-Operator in der Bedeutung überladen, dass mit Ausführung der entsprechenden Operatormethode die Koordinatenwerte des aktuellen `Punkt`-Objekts und diejenigen eines anderen addiert werden. Mit aktuellem `Punkt`-Objekt ist natürlich diejenige Instanz der Klasse gemeint, für die die Operatormethode jeweils aufgerufen wird. Einziger Parameter ist daher ebenfalls ein Objekt des Typs `Punkt`. Als Grundgerüst einer entsprechenden Methode zur Überladung des binären `»+«`-Operators für die Klasse `Punkt` ergibt sich damit:

```
Punkt operator+(Punkt Anderer)
{
}
```

Im Rumpf der Methode muss die Addition der Koordinaten des übergebenen Punktes mit den Datenelementen `x` und `y` des aktuellen Punktes stattfinden. Das Ergebnis wird einer lokalen Instanz der Klasse `Punkt` zugewiesen, die den Rückgabewert der Operatormethode bildet:

```
Punkt operator+(Punkt Anderer)
{
    Punkt SumKo;
    SumKo.x = x + Anderer.x;
    SumKo.y = y + Anderer.y;
    return SumKo;
}
```

Damit ergibt sich für die Klasse Punkt folgendes Aussehen:

```
class Punkt
{
private:
    int x;
    int y;
public:
    Punkt() {x = 0; y = 0;}
    Punkt(int xko, int yko)
    {x = xko; y = yko;}
    void setX(int xko) {x = xko;}
    int getX() {return x;}
    void setY(int yko) {y = yko;}
    int getY() {return y;}
    void zeigePunkt();

    Punkt operator+(Punkt Anderer)
    {
        Punkt SumKo;
        SumKo.x = x + Anderer.x;
        SumKo.y = y + Anderer.y;
        return SumKo;
    }
};

void Punkt::zeigePunkt()
{
    cout << "x-Koordinate: " << x << endl;
    cout << "y-Koordinate: " << y << endl;
}
```

bzw. – falls man die Operatormethode operator+() außerhalb der Klasse definiert:

```
class Punkt
{
private:
    int x;
    int y;
public:
    Punkt() {x = 0; y = 0;}
    Punkt(int xko, int yko)
    {x = xko; y = yko;}
    void setX(int xko) {x = xko;}
    int getX() {return x;}
    void setY(int yko) {y = yko;}
    int getY() {return y;}
```

```

    void zeigePunkt();
    Punkt operator+(Punkt Anderer);
};

void Punkt::zeigePunkt()
{
    cout << "x-Koordinate: " << x << endl;
    cout << "y-Koordinate: " << y << endl;
}

Punkt Punkt::operator+(Punkt Anderer)
{
    Punkt SumKo;
    SumKo.x = x + Anderer.x;
    SumKo.y = y + Anderer.y;
    return SumKo;
}

```

Die oben genannten Operationen sind nun mit Instanzen der Punkt-Klasse durchführbar, wie das folgende Beispiel zeigt:

```

#include <iostream>
using namespace std;

class Punkt
{
private:
    int x;
    int y;
public:
    Punkt() {x = 0; y = 0;}
    Punkt(int xko, int yko)
    {x = xko; y = yko;}
    void setX(int xko) {x = xko;}
    int getX() {return x;}
    void setY(int yko) {y = yko;}
    int getY() {return y;}
    void zeigePunkt();
    Punkt operator+(Punkt Anderer);
};

void Punkt::zeigePunkt()
{
    cout << "x-Koordinate: " << x << endl;
    cout << "y-Koordinate: " << y << endl;
}

```



```
Punkt Punkt::operator+(Punkt Anderer)
{
    Punkt SumKo;
    SumKo.x = x + Anderer.x;
    SumKo.y = y + Anderer.y;
    return SumKo;
}

int main(void)
{
    Punkt A(2, 3), B(10, 7), C;
    C = A + B;
    cout << "Punkt A: " << endl;
    A.zeigePunkt();
    cout << "Punkt B: " << endl;
    B.zeigePunkt();
    cout << "Punkt C: " << endl;
    C.zeigePunkt();
    return 0;
}
```

Ausgabe:

```
Punkt A:
x-Koordinate: 2
y-Koordinate: 3
Punkt B:
x-Koordinate: 10
y-Koordinate: 7
Punkt C:
x-Koordinate: 12
y-Koordinate: 10
```

Beachten Sie, dass die Operatormethode das Objekt, für das sie aufgerufen wird – hier A –, nicht verändert. Damit kann der »+«-Operator für Punkt-Objekte in ähnlicher Bedeutung verwendet werden, wie man es von den Standarddatentypen her gewohnt ist.

Es sei darauf hingewiesen, dass ein Ausdruck

```
Objekt_1 + Objekt_2
```

schließlich eine Kurzform für den expliziten Methodenaufruf

```
Objekt_1.operator+(Objekt_2)
```

darstellt.

Im Beispiel hätte man also statt

```
A + B
```

auch

```
A.operator+(B)
```

schreiben können:

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

class Punkt
{
private:
    int x;
    int y;
public:
    Punkt() {x = 0; y = 0;}
    Punkt(int xko, int yko)
    {x = xko; y = yko;}
    void setX(int xko) {x = xko;}
    int getX() {return x;}
    void setY(int yko) {y = yko;}
    int getY() {return y;}
    void zeigePunkt();
    Punkt operator+(Punkt Anderer);
};

void Punkt::zeigePunkt()
{
    cout << "x-Koordinate: " << x << endl;
    cout << "y-Koordinate: " << y << endl;
}

Punkt Punkt::operator+(Punkt Anderer)
{
    Punkt SumKo;
    SumKo.x = x + Anderer.x;
    SumKo.y = y + Anderer.y;
    return SumKo;
}
```

```
int main(void)
{
    Punkt A(2, 3), B(10, 7), C;
    C = A.operator+(B);
    cout << "Punkt A: " << endl;
    A.zeigePunkt();
    cout << "Punkt B: " << endl;
    B.zeigePunkt();
    cout << "Punkt C: " << endl;
    C.zeigePunkt();
    return 0;
}
```

Ausgabe:

```
Punkt A:
x-Koordinate: 2
y-Koordinate: 3
Punkt B:
x-Koordinate: 10
y-Koordinate: 7
Punkt C:
x-Koordinate: 12
y-Koordinate: 10
```

Dies nur zum besseren Verständnis. Sinn und Zweck der Operatorüberladung ist es natürlich, zu ermöglichen, dass Operatoren für Objekte einer bestimmten Klasse in der gewohnten Weise verwendet werden können. In der Regel wird man daher der Kurzschreibweise den Vorzug geben. Wichtig für Sie zu wissen ist aber, dass es sich – bezogen auf obiges Beispiel – bei einem Ausdruck wie

```
A + B
```

tatsächlich um einen Aufruf der Operatormethode `operator+()` für das Objekt A mit dem Argument B handelt.

Beachten Sie, dass eine Definition der obigen Operatormethode `operator+()` mit je einem Parameter für die x- sowie für die y-Koordinate als Syntaxfehler interpretiert wird:

```
Punkt operator+(int xko, int yko) // FEHLER
{
    Punkt SumKo;
    SumKo.x = x + xko;
    SumKo.y = y + yko;
    return SumKo;
}
```

Eine Methode zum Überladen eines binären Operators darf deshalb nicht mehr – und auch nicht weniger – als einen Parameter besitzen, da ein binärer Operator eben genau einen rechten Operanden hat und das Argument einer aufgerufenen Operatormethode – bezogen auf die Kurzschreibweise – ja diesen rechten Operanden darstellt.

Während jedoch der linke Operand eines in Form einer Klassenmethode überladenen Operators immer die Instanz ist, für die die entsprechende Operatormethode aufgerufen wird, muss es sich bei dem rechten Operanden nicht notwendig um ein Objekt derselben Klasse handeln. So ließe sich für die Klasse `Punkt` auch eine Operatormethode definieren, die Operationen in der Form

```
einPunkt + 7
```

gestattet. Wobei `einPunkt` natürlich ein `Punkt`-Objekt sein muss, der rechte Operand aber ein beliebiger Integerwert sein kann. Im Ergebnis soll eine Addition der Koordinatenwerte eines Punktes mit einem Integerwert stattfinden, sodass z.B. der obige Ausdruck ein `Punkt`-Objekt mit den Koordinaten 8,9 liefert, falls `einPunkt` die Koordinaten 1,2 besitzt:

```
Punkt operator+(int koo)
{
    Punkt SumKo;
    SumKo.x = x + koo;
    SumKo.y = y + koo;
    return SumKo;
}
```

Auch diese Methode führt keine Veränderungen am aktuellen Objekt durch. Dessen Datenelemente `x` und `y` erscheinen allein jeweils auf der rechten Seite beider Zuweisungen. Für den passenden Rückgabewert sorgt wiederum das lokale Objekt `SumKo`:

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

class Punkt
{
private:
    int x;
    int y;
public:
    Punkt() {x = 0; y = 0;}
    Punkt(int xko, int yko)
    {x = xko; y = yko;}
    void setX(int xko) {x = xko;}
    int getX() {return x;}
    void setY(int yko) {y = yko;}
    int getY() {return y;}
}
```

```
void zeigePunkt();
Punkt operator+(Punkt Anderer);
Punkt operator+(int koo);
};

void Punkt::zeigePunkt()
{
    cout << "x-Koordinate: " << x << endl;
    cout << "y-Koordinate: " << y << endl;
}

Punkt Punkt::operator+(Punkt Anderer)
{
    Punkt SumKo;
    SumKo.x = x + Anderer.x;
    SumKo.y = y + Anderer.y;
    return SumKo;
}

Punkt Punkt::operator+(int koo)
{
    Punkt SumKo;
    SumKo.x = x + koo;
    SumKo.y = y + koo;
    return SumKo;
}

int main(void)
{
    Punkt A(25, 5), B;
    B = A + 5;
    cout << "Punkt A: " << endl;
    A.zeigePunkt();
    cout << "Punkt B: " << endl;
    B.zeigePunkt();
    return 0;
}
```

Ausgabe:

```
Punkt A:
x-Koordinate: 25
y-Koordinate: 5
Punkt B:
x-Koordinate: 30
y-Koordinate: 10
```

Nun kann es sich ein Programmierer, der die Klasse `Punkt` in seinen Programmen verwendet, aussuchen, ob er einen `Punkt` oder einen einfachen Integerwert als rechten Operanden der für die Klasse verfügbaren »+«-Operation einsetzen möchte. Wie gesagt, ist die Bedeutung dieser Operation mit den Definitionen der beiden Operatormethoden `operator+` festgelegt.

### Hinweis

Lassen Sie sich durch den Sprachgebrauch nicht irritieren. Der binäre »+«-Operator wurde für die Klasse `Punkt` überladen und die Operatormethode `operator+` ist ebenfalls überladen. Genau genommen hat das eine nichts mit dem anderen zu tun. Erstes bedeutet, dass für Objekte der Klasse `Punkt` nunmehr eine »+«-Operation durchgeführt werden kann, Letzteres, dass in der Klasse – also in ein und demselben Gültigkeitsbereich – zwei Methoden gleichen Namens definiert sind (zum Überladen von Funktionen siehe Kapitel 20, Abschnitt 20.5 »Überladen von Funktionen«).

Das folgende Beispiel, in dem zwei »+«-Operationen miteinander verkettet werden, verdeutlicht, dass die Linksassoziativität des binären »+«-Operators nach dessen Überladung erhalten bleibt:

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

class Punkt
{
private:
    int x;
    int y;
public:
    Punkt() {x = 0; y = 0;}
    Punkt(int xko, int yko)
        {x = xko; y = yko;}
    void setX(int xko) {x = xko;}
    int getX() {return x;}
    void setY(int yko) {y = yko;}
    int getY() {return y;}
    void zeigePunkt();
    Punkt operator+(Punkt Anderer);
    Punkt operator+(int koo);
};

void Punkt::zeigePunkt()
{
    cout << "x-Koordinate: " << x << endl;
    cout << "y-Koordinate: " << y << endl;
}
```

```
Punkt Punkt::operator+(Punkt Anderer)
{
    Punkt SumKo;
    SumKo.x = x + Anderer.x;
    SumKo.y = y + Anderer.y;
    return SumKo;
}

Punkt Punkt::operator+(int koo)
{
    Punkt SumKo;
    SumKo.x = x + koo;
    SumKo.y = y + koo;
    return SumKo;
}

int main(void)
{
    Punkt Erster(11, 4), Zweiter(2, 3);
    cout << "Punkt Zweiter vor der Zuweisung: " << endl;
    Zweiter.zeigePunkt();
    Zweiter = Zweiter + 30 + Erster;
    cout << "Punkt Zweiter nach der Zuweisung: " << endl;
    Zweiter.zeigePunkt();
    return 0;
}
```

Ausgabe:

```
Punkt Zweiter vor der Zuweisung:
x-Koordinate: 2
y-Koordinate: 3
Punkt Zweiter nach der Zuweisung:
x-Koordinate: 43
y-Koordinate: 37
```

Mit der Zuweisung

```
Zweiter = Zweiter + 30 + Erster;
```

werden zu den bestehenden Koordinatenwerten von Zweiter jeweils der Wert 30 sowie die Koordinatenwerte des Punktes Erster hinzuaddiert. Dabei entspricht die Ausführung der Additionsoperationen der Klammerung

```
Zweiter = (Zweiter + 30) + Erster;
```

wobei der Ausdruck

```
Zweiter + 30
```

ein Punkt-Objekt zurückgibt und daher die folgende Addition zweier Punkte problemlos durchgeführt werden kann. Beachten Sie, dass die nicht geklammerte Anweisung des Beispiels fehlerhaft wäre, falls es sich mit + um einen rechtsassoziativen Operator handeln würde, da eine Operation

```
30 + Erster
```

eben nicht durchführbar ist.

### 31.2.2 Überladen von unären Operatoren

Als Beispiel für die Überladung eines unären Operators soll uns der Dekrementoperator (--) dienen. Er soll für die Klasse Punkt in der Bedeutung überladen werden, dass die damit verbundene Operation den Wert beider Koordinaten um den Betrag von eins vermindert. Eine Operatormethode, die einen unären Operator überlädt, ist grundsätzlich parameterlos, da ein unärer Operator ja nur einen einzigen Operanden besitzt. Rückgabewert der Operatormethode ist wiederum ein Objekt der Klasse Punkt.

```
Punkt operator--()
{
}
```

Im Unterschied zu den Operatormethoden des vorigen Abschnitts soll nun das Objekt, für das die Methode aufgerufen wird, tatsächlich in der beschriebenen Weise verändert werden. Zudem soll ein Ausdruck wie

```
--einPunkt
```

natürlich gleichzeitig den Wert des modifizierten Objekts zurückgeben. Um aus der Methode die Werte des aktuellen Objekts herauszugeben, liegt es daher nahe, statt einer lokalen Kopie für den Rückgabewert den dereferenzierten this-Zeiger zu verwenden:

```
Punkt operator--()
{
    x--;
    y--;
    return *this;
}
```



Fügt man diese Methode der Klasse Punkt hinzu, so können Objekte dieser Klasse in der beschriebenen Weise dekrementiert werden:

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

class Punkt
{
private:
    int x;
    int y;
public:
    Punkt() {x = 0; y = 0;}
    Punkt(int xko, int yko)
    {x = xko; y = yko;}
    void setX(int xko) {x = xko;}
    int getX() {return x;}
    void setY(int yko) {y = yko;}
    int getY() {return y;}
    void zeigePunkt();
    Punkt operator+(Punkt Anderer);
    Punkt operator+(int koo);
    Punkt operator--();
};

void Punkt::zeigePunkt()
{
    cout << "x-Koordinate: " << x << endl;
    cout << "y-Koordinate: " << y << endl;
}

Punkt Punkt::operator+(Punkt Anderer)
{
    Punkt SumKo;
    SumKo.x = x + Anderer.x;
    SumKo.y = y + Anderer.y;
    return SumKo;
}

Punkt Punkt::operator+(int koo)
{
    Punkt SumKo;
    SumKo.x = x + koo;
    SumKo.y = y + koo;
    return SumKo;
}
```

```

Punkt Punkt::operator--()
{
    x--;
    y--;
    return *this;
}

int main(void)
{
    Punkt Eins(7, 6), Zwei;
    Zwei = --Eins;
    cout << "Punkt Eins: " << endl;
    Eins.zeigePunkt();
    cout << "Punkt Zwei: " << endl;
    Zwei.zeigePunkt();
    return 0;
}

```

Ausgabe:

```

Punkt Eins:
x-Koordinate: 6
y-Koordinate: 5
Punkt Zwei:
x-Koordinate: 6
y-Koordinate: 5

```

Wie am Beispiel zu sehen ist, erfolgt die Inkrementierung in der Bedeutung einer Präfixnotation, da in der Anweisung

```
Zwei = --Eins;
```

das Objekt `Zwei` die Werte des dekrementierten Objekts `Eins` erhält. Schließlich ist die Methode `operator--()` ja entsprechend eingerichtet. Daher würde sich an diesem Ergebnis auch nichts ändern, falls man den Operator `--` in der Postfixnotation verwendet:

```
Zwei = Eins--;
```

Allerdings erhielten Sie vermutlich einen Hinweis Ihres Compilers, dass er auf die Präfixvariante zurückgreift, da keine Postfixversion definiert ist.

Tatsächlich ist für die obige Definition der Operatormethode `operator--()` die Präfixschreibweise vorgesehen. Das heißt, korrekt kann dieser Operator nur in der Form

```
--einPunkt
```

verwendet werden.

**Hinweis**

Wobei es sich natürlich um zwei verschiedene Dinge handelt, wie die Operation im Code zu notieren ist und was die Operation bewirkt. Letzteres entscheidet allein die Operatormethode, die dahinter steht, bzw. deren Implementation. So ließe sich die obige Methode `operator--()` natürlich auch dementsprechend einrichten, dass die Dekrementierung gegebenenfalls erst im Anschluss an eine Zuweisung erfolgt. Womit die Präfixvariante des Dekrementoperators für die Klasse `Punkt` in einer Bedeutung vorläge, die der von den elementaren Datentypen her gewohnten genau entgegensteht (was natürlich nicht als Empfehlung verstanden werden soll).

Um eine entsprechende Operatormethode in der Postfixversion zu definieren, muss die Methode mit einem zusätzlichen Integerparameter versehen werden. Dies sowie das bezüglich Präfix-/Postfixnotation Gesagte gilt natürlich in gleicher Weise für den Inkrementoperator. Insofern bilden beide Operatoren eine Ausnahme von der oben genannten Regel, dass Operatormethoden für unäre Operatoren keine Parameter besitzen.

Allerdings wird dieser Parameter nicht weiterverwendet. Er dient lediglich in Verbindung mit dem Überladen von Operatoren dazu – indem er angegeben oder eben nicht angegeben wird –, den Compiler zwischen Präfix- und Postfixvariante unterscheiden zu lassen. Demgemäß wird Ihr Compiler die Definition einer Methode `operator++()` mit einem `int`-Parameter erwarten, falls er im Code einen Ausdruck wie z.B.

```
einPunkt++
```

antrifft.

Als Grundgerüst für die Postfixversion der Methode `operator--()` für die Klasse `Punkt` ergibt sich damit:

```
Punkt operator--(int)
{
}
```

**Hinweis**

Es ist nicht erforderlich, einen Bezeichner für den Parameter anzugeben, da dieser innerhalb der Methode nicht verwendet wird.

Der Rückgabewert ist wie bei der Präfixmethode vom Typ `Punkt`. Allerdings werden wir die Operation in der gewohnten Bedeutung einrichten, das heißt, ein Ausdruck wie

```
einPunkt--
```

soll den Wert des Punkt-Objekts vor der Dekrementierung zurückgeben. Dazu werden die Koordinaten vor der Dekrementierung in einem lokalen Punkt-Objekt gespeichert und dieses anschließend aus der Methode herausgegeben:

```
Punkt operator--(int)
{
    Punkt KopieAlt = *this;
    x--;
    y--;
    return KopieAlt;
}
```

Fügt man diese Methode der Punkt-Klasse hinzu, so kann die Postfixnotation in der bekannten Form für Instanzen dieser Klasse eingesetzt werden:

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

class Punkt
{
private:
    int x;
    int y;
public:
    Punkt() {x = 0; y = 0;}
    Punkt(int xko, int yko)
    {x = xko; y = yko;}
    void setX(int xko) {x = xko;}
    int getX() {return x;}
    void setY(int yko) {y = yko;}
    int getY() {return y;}
    void zeigePunkt();
    Punkt operator+(Punkt Anderer);
    Punkt operator+(int koo);
    Punkt operator--();
    Punkt operator--(int);
};

void Punkt::zeigePunkt()
{
    cout << "x-Koordinate: " << x << endl;
    cout << "y-Koordinate: " << y << endl;
}

Punkt Punkt::operator+(Punkt Anderer)
{
    Punkt SumKo;
```

```
SumKo.x = x + Anderer.x;
SumKo.y = y + Anderer.y;
return SumKo;
}

Punkt Punkt::operator+(int koo)
{
    Punkt SumKo;
    SumKo.x = x + koo;
    SumKo.y = y + koo;
    return SumKo;
}

Punkt Punkt::operator--()
{
    x--;
    y--;
    return *this;
}

Punkt Punkt::operator--(int)
{
    Punkt KopieAlt = *this;
    x--;
    y--;
    return KopieAlt;
}

int main(void)
{
    Punkt Eins(7, 6), Zwei;
    Zwei = Eins--;
    cout << "Punkt Eins: " << endl;
    Eins.zeigePunkt();
    cout << "Punkt Zwei: " << endl;
    Zwei.zeigePunkt();
    return 0;
}
```

Ausgabe:

```
Punkt Eins:
x-Koordinate: 6
y-Koordinate: 5
Punkt Zwei:
x-Koordinate: 7
y-Koordinate: 6
```

Was den Ausdruck

```
Eins--
```

angeht, so kümmert sich Ihr Compiler um den Übergabewert an die Postfixoperatormethode. Anders verhält es sich, wenn man diese in der Form

```
Eins.operator--(5)
```

aufruft. Wobei der tatsächliche Wert des Arguments natürlich keine Bedeutung hat.

### 31.2.3 Überladen eines Vergleichsoperators

Zum Abschluss und um zu demonstrieren, dass eine Operatormethode nicht notwendig ein Objekt ihrer Klasse zurückgeben muss, soll die Punkt-Klasse noch um eine Überladung des binären Vergleichsoperators `==` erweitert werden. Dies soll in der Bedeutung geschehen, dass ein Ausdruck

```
Punkt_1 == Punkt_2
```

den Wert `true` liefert, falls `x`- und `y`-Koordinaten beider Punkte übereinstimmen, andernfalls `false`. Demnach muss die Operatormethode `operator==(Punkt Anderer)` einen booleschen Wert zurückgeben. Rechter Operand ist ein `Punkt`-Objekt, womit der Typ des formalen Parameters ebenfalls feststeht. Im Rumpf der Methode sind die entsprechenden Koordinatenvergleiche durchzuführen:

```
bool operator==(Punkt Anderer)
{
    if (x == Anderer.x && y == Anderer.y)
        return true;
    else
        return false;
}
```

Definiert man die Methode außerhalb der Klasse, ergibt sich für diese nunmehr folgendes Bild:

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

class Punkt
{
private:
    int x;
    int y;
```

```
public:
    Punkt() {x = 0; y = 0;}
    Punkt(int xko, int yko)
        {x = xko; y = yko;}
    void setX(int xko) {x = xko;}
    int getX() {return x;}
    void setY(int yko) {y = yko;}
    int getY() {return y;}
    void zeigePunkt();
    Punkt operator+(Punkt Anderer);
    Punkt operator+(int koo);
    Punkt operator--();
    Punkt operator--(int);
    bool operator==(Punkt Anderer);
};

void Punkt::zeigePunkt()
{
    cout << "x-Koordinate: " << x << endl;
    cout << "y-Koordinate: " << y << endl;
}

Punkt Punkt::operator+(Punkt Anderer)
{
    Punkt SumKo;
    SumKo.x = x + Anderer.x;
    SumKo.y = y + Anderer.y;
    return SumKo;
}

Punkt Punkt::operator+(int koo)
{
    Punkt SumKo;
    SumKo.x = x + koo;
    SumKo.y = y + koo;
    return SumKo;
}

Punkt Punkt::operator--()
{
    x--;
    y--;
    return *this;
}
```

```
Punkt Punkt::operator--(int)
{
    Punkt KopieAlt = *this;
    x--;
    y--;
    return KopieAlt;
}

bool Punkt::operator==(Punkt Anderer)
{
    if (x == Anderer.x && y == Anderer.y)
        return true;
    else
        return false;
}

int main(void)
{
    Punkt einPunkt(5, 15), nochEinPunkt(5, 15);
    if (einPunkt == nochEinPunkt)
    {
        cout << "Beide Punkte haben die gleichen"
              << " Koordinaten" << endl;
    }
    else
    {
        cout << "Die Koordinaten beider Punkte sind"
              << " verschieden" << endl;
    }
    return 0;
}
```

Ausgabe:

```
Beide Punkte haben die gleichen Koordinaten
```



# 32

## Ausnahmebehandlung

Während des Programmlaufs auftretende Laufzeitfehler werden in C++ »Ausnahmen« (Englisch »exceptions«) genannt. Der ANSI/ISO-C++-Standard sieht insgesamt acht Ausnahmesituationen vor. Darüber hinaus steht es dem Programmierer frei, im Code Ausnahmesituationen nach seiner Wahl selbst festzulegen. Ebenso ist es möglich, Ausnahmen im Code individuell zu behandeln.

In diesem Zusammenhang bedient man sich einer speziellen Ausdrucksweise: Wenn während des Programmlaufs eine Fehlersituation auftritt, sagt man, es wird eine Ausnahme »geworfen«. Dementsprechend spricht man vom Auffangen dieser Ausnahme, falls im Code auf die spezielle Fehlersituation reagiert wird. Dies geschieht, wie wir gleich sehen werden, in einem separaten Anweisungsblock, dem so genannten catch-Handler (»catch«, zu Deutsch »fangen«).

### 32.1 try - catch

Nehmen wir an, in einem Programm wird der Benutzer aufgefordert, zwei Zahlen einzugeben, die im Anschluss daran die Operanden einer Division bilden:

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

int main(void)
{
    double z1, z2, ergebnis;
    cin >> z1 >> z2;
    ergebnis = z1 / z2;
    cout << "Ergebnis: " << ergebnis;
    // ...
    return 0;
}
```

Gibt der Benutzer nun als zweite Zahl eine Null ein, so ist das Ergebnis der Operation

$z1 / z2$

nicht vorhersehbar.

**Hinweis**

Einige Compiler behandeln Divisionen durch Null in mehr oder weniger unbefriedigender Weise automatisch, andere gar nicht. Im ersteren Fall hat die oben genannte Operation nur undefinierte Berechnungsergebnisse zur Folge, im letzteren führt sie zu einem Programmabsturz.

Will man für die beschriebene Situation eine Ausnahmebehandlung durchführen, so fasst man den sensiblen Code zunächst in einen Block, der mit dem Schlüsselwort `try` eingeleitet wird:

```
int main(void)
{
    double z1, z2, ergebnis;
    try
    {
        cin >> z1 >> z2;
        ergebnis = z1 / z2;
        cout << "Ergebnis: " << ergebnis;
    }
    // ...
    return 0;
}
```

**Hinweis**

Es sei darauf hingewiesen, dass sich jedem `try`-Block zumindest ein `catch`-Handler anschließen muss. Obige `main()`-Funktion ist in diesem Sinne (noch) fehlerhaft.

Um innerhalb des `try`-Blocks eine Ausnahmesituation festzulegen, verwendet man den `throw`-Operator, dessen rechter Operand ein beliebiger Ausdruck ist (»throw«, zu Deutsch »werfen«). Beispielsweise wird mit

```
throw 5;
```

eine Ausnahme mit dem Integerwert 5 geworfen, falls die Anweisung während des Programmlaufs zum Zuge kommt. Zur Verdeutlichung empfiehlt es sich, den Datentypbezeichner mit anzugeben:

```
throw int (5);
```

Wobei es grundsätzlich dem Programmierer überlassen bleibt, für welchen Datentyp bzw. für welchen Wert er sich entscheidet.

## Hinweis

Der Ausdruck

```
int (5)
```

erzeugt eine temporäre, namenlose Integervariable und initialisiert diese mit dem Wert 5. Die Schreibweise lässt erkennen, dass auch elementare Datentypen über eine Art Konstruktor verfügen. So lässt sich eine Integervariable bei der Deklaration nicht nur mit

```
int zahl = 10;
```

sondern auch wie folgt initialisieren:

```
int zahl(10);
```

In unserem Beispiel soll, falls der Benutzer beim Programmlauf als zweite Zahl eine Null eingibt, eine Ausnahme des Typs `string` geworfen werden (wobei der spezielle Wert vorerst keine Rolle spielen soll). Wie gesagt, könnte es sich genauso gut um jeden anderen Datentyp handeln.

```
int main(void)
{
    double z1, z2, ergebnis;
    try
    {
        cin >> z1 >> z2;
        if (z2 == 0)
            throw string("");
        ergebnis = z1 / z2;
        cout << "Ergebnis: " << ergebnis;
    }
    // ...
    return 0;
}
```

Wie oben angedeutet, muss einem `try`-Block unmittelbar ein `catch`-Handler folgen. Dies ist ein mit diesem Schlüsselwort eingeleiteter Block, der für die Ausnahmebehandlung vorgesehen ist. Nach dem Schlüsselwort `catch` ist in Klammern genau ein formaler Parameter anzugeben. Falls es nicht auf den speziellen Wert des Parameters ankommt, kann auf die Angabe eines Bezeichners verzichtet werden:

```
int main(void)
{
    double z1, z2, ergebnis;
    try
    {
        cin >> z1 >> z2;
```

```

    if (z2 == 0)
        throw string("");
    ergebnis = z1 / z2;
    cout << "Ergebnis: " << ergebnis;
}
catch (string)
{
}
// ...
return 0;
}

```

Nun soll im Block des catch-Handlers auf die genannte Ausnahme in der Weise reagiert werden, dass das Programm mit einer passenden Fehlermeldung kontrolliert beendet wird:

```

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

int main(void)
{
    double z1, z2, ergebnis;
    try
    {
        cin >> z1 >> z2;
        if (z2 == 0)
            throw string("");
        ergebnis = z1 / z2;
        cout << "Ergebnis: " << ergebnis;
    }
    catch (string)
    {
        cerr << "Fehler: Division durch Null" << endl;
        cerr << "Programm wird beendet" << endl;
        return -1;
    }
    // ...
    return 0;
}

```

Ausgabe:

```

17.3  0
Fehler: Division durch Null
Programm wird beendet

```

## Referenz

Zum Objekt `cerr` siehe Kapitel 26, Abschnitt 26.2 »Aus Textdateien lesen«.

### Die Funktion `exit()`

Die `return`-Anweisung beendet, wie Ihnen bekannt ist, nur die aktuelle Funktion – was natürlich gleichbedeutend mit dem Programmende ist, falls sich die Programmausführung wie hier in `main()` befindet. Um ein Programm von einer anderen Funktion aus unverzüglich zu beenden, können Sie die Funktion `exit()` verwenden. Dabei ist zu beachten, dass `exit()` immer mit einem Integerwert aufzurufen ist, auch wenn der Rückgabebetyp der aktuellen Funktion `void` ist:

```
...
void function()
{
    try
    {
        ...
    }
    catch(string)
    {
        ...
        exit(-1);
    }
}
int main()
{
    ...
    function();
    ...
}
```

Falls die Programmausführung im `catch`-Handler der Funktion `function()` landet, beendet dort die Anweisung

```
exit(-1);
```

das Programm mit dem Rückgabewert `-1`.

Wie bei den Parametern einer Funktion kann der Übergabewert an den `catch`-Handler von diesem im Block auch weiterverarbeitet werden. In diesem Fall ist der Bezeichner des formalen Parameters anzugeben. Schließlich handelt es sich auch hier um die Deklaration einer Variablen:

```

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

int main(void)
{
    double z1, z2, ergebnis;
    try
    {
        cin >> z1 >> z2;
        if (z2 == 0)
            throw string("Fehler: Division durch Null");
        ergebnis = z1 / z2;
        cout << "Ergebnis: " << ergebnis;
    }
    catch (string s)
    {
        cerr << s << endl;
        cerr << "Programm wird beendet" << endl;
        return -1;
    }
    // ...
    return 0;
}

```

Ausgabe:

```

19 0
Fehler: Division durch Null
Programm wird beendet

```

### 32.1.1 Benutzerdefinierte Klassen für die Fehlerbehandlung

Manchmal ist es nötig, für die Ausnahmebehandlung eigene Datentypen zu definieren. In diesem Sinne bietet es sich bezüglich des obigen Beispiels an, für die Ausnahme »Division durch Null« eine Klasse `DivDurchNull` zu definieren:

```

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

class DivDurchNull
{
public:
    string meldung;
    DivDurchNull() {}
}

```

```
    DivDurchNull(string m)
    {meldung = m;}
};

int main(void)
{
    double z1, z2, ergebnis;
    try
    {
        cin >> z1 >> z2;
        if (z2 == 0)
            throw DivDurchNull ("Fehler: Division durch Null");
        ergebnis = z1 / z2;
        cout << "Ergebnis: " << ergebnis;
    }
    catch (DivDurchNull F)
    {
        cerr << F.meldung << endl;
        cerr << "Programm wird beendet" << endl;
        return -1;
    }
    // ...
    return 0;
}
```

Ausgabe:

```
56 0
Fehler: Division durch Null
Programm wird beendet
```

Falls man für Ausnahmen des Typs `DivDurchNull` in seinen Programmen grundsätzlich keine Übergabewerte an `catch`-Handler vorsieht, ließe sich die Klassendefinition auch auf ein Minimum reduzieren:

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

struct DivDurchNull {};

int main(void)
{
    double z1, z2, ergebnis;
    try
    {
        cin >> z1 >> z2;
```

```

    if (z2 == 0)
        throw DivDurchNull();
    ergebnis = z1 / z2;
    cout << "Ergebnis: " << ergebnis;
}
catch (DivDurchNull)
{
    cerr << "Fehler: Division durch Null" << endl;
    cerr << "Programm wird beendet" << endl;
    return -1;
}
// ...
return 0;
}

```

Ausgabe:

```

7.5  0
Fehler: Division durch Null
Programm wird beendet

```

### Hinweis

Der throw-Ausdruck

`DivDurchNull()`

erzeugt ein temporäres `DivDurchNull`-Objekt unter Verwendung eines Standardkonstruktors, der der Klasse vom Compiler automatisch hinzugefügt wird (Kapitel 24, Abschnitt 24.3.2 »Ersatzkonstruktor«).

## 32.1.2 Mehrere catch-Handler

Grundsätzlich können sich einem try-Block beliebig viele catch-Handler anschließen:

```

try
{
    ...
}
catch(DivDurchNull)
{
    ...
}
catch(FileOpenError dat)
{
    cerr << "Datei " << dat

```



```
    << "konnte nicht geoeffnet werden";  
    ...  
}
```

Wird im try-Block eine Ausnahme geworfen, so landet die Programmausführung im ersten vom Typ her passenden catch-Handler (wobei die angegebenen catch-Handler von oben nach unten betrachtet werden). Sobald ein catch-Handler zur Ausführung gelangt, fährt die Programmausführung mit der ersten Anweisung nach dem letzten catch-Handler fort – es sei denn, das Programm bzw. die Funktion wird im Block des ausgeführten catch-Handlers beendet.

Wird eine Ausnahme nicht behandelt, ist also kein zum Typ der Ausnahme passender catch-Handler definiert, so führt dies zu einem impliziten Aufruf einer Funktion `terminate()`, die für den Abbruch des Programms mit der Meldung »abnormal program termination« sorgt. Um dies zu vermeiden, lässt sich ein catch-Handler definieren, der jede Ausnahme fängt. Dazu sind in den Klammern drei Punkte (...) anzugeben:

```
try  
{  
    ...  
}  
catch(DivDurchNull)  
{  
    ...  
}  
catch(FileOpenError dat)  
{  
    cerr << "Datei " << dat  
        << "konnte nicht geoeffnet werden";  
    ...  
}  
catch(...)  
{  
    ...  
}
```

Beachten Sie, dass der allgemeine catch-Handler von der Logik her der letzte sein muss.

## 32.2 Auffangen von Ausnahmen im Aufrufer

Die Philosophie der Ausnahmebehandlung von C++ besteht darin, Fehlerursache und Fehlerbehandlung voneinander zu trennen. Dies ist vor allem dann bedeutsam, wenn eine Funktion die andere aufruft, wobei Ausnahmen in der aufgerufenen Funktion im Aufrufer behandelt werden. Hat man etwa eine Funktion `div()`, die Ausnahmen des Typs `DivDurchNull` wirft, aber nicht selbst darauf reagiert, so können diese in der aufrufenden Umgebung aufgefangen werden:

```

#include <iostream>
using namespace std;

struct DivDurchNull {};

double div(double a, double b)
{
    if (b == 0)
        throw DivDurchNull();
    return a / b;
}

int main(void)
{
    try
    {
        double zahl1, zahl2, erg;
        cout << "Gib mir zwei Zahlen: ";
        cin >> zahl1 >> zahl2;
        erg = div(zahl1, zahl2);
        cout << "Ergebnis: " << erg;
        // ...
        return 0;
    }
    catch (DivDurchNull)
    {
        cerr << "Fehler: Division durch Null" << endl;
        cerr << "Programm wird beendet" << endl;
        return -1;
    }
    catch(...)
    {
        cerr << "Unerwartete Ausnahme" << endl;
        cerr << "Programm wird beendet" << endl;
        return -2;
    }
}

```

Ausgabe:

```

Gib mir zwei Zahlen: 85 0
Fehler: Division durch Null
Programm wird beendet

```

Falls in div() mit

```
throw DivDurchNull();
```

eine Ausnahme geworfen wird, erfolgt ein unverzüglicher Rücksprung nach `main()`. Dort wird dann der passende `catch`-Handler ausgeführt. Nach Abarbeitung eines `catch`-Handlers verbleibt die Programmausführung im Aufrufer, es erfolgt also nicht etwa ein automatischer Rücksprung in die Funktion, in der der Fehler aufgetreten ist.

Sobald also eine Funktion eine Ausnahme wirft und diese nicht im Block der Funktion aufgefangen wird, wird der dazu passende `catch`-Handler in der aufrufenden Umgebung gesucht. Dieser Mechanismus funktioniert auch über mehrere Aufrufebenen hinweg:

```
void func1()
{
    ...
    // func1() wirft Ausnahme vom Typ ErrorXY
    throw ErrorXY();
    ...
}

void func2()
{
    ...
    func1();
    ...
}

int main()
{
    try
    {
        ...
        func2();
        ...
    }
    catch(ErrorXY)
    {
        // Hier werden Ausnahmen vom Typ ErrorXY behandelt
    }
}
```

Hier wird im `try`-Block von `main()` die Funktion `func2()` aufgerufen, in dieser wiederum die Funktion `func1()`, die Ausnahmen des Typs `ErrorXY` wirft. Falls diese Ausnahmesituation während der Programmausführung in `func1()` auftritt, es also zur Ausführung der Anweisung

```
throw ErrorXY();
```

kommt, erfolgt der unverzügliche Sprung nach `main()`, da sich dort der nächste passende `catch`-Handler befindet.



# Bonusteil

## **Managed Code; GUI-Programmierung**

In diesem zusätzlichen Teil erfahren Sie zunächst einiges Wissenswertes über das .NET von Microsoft. Außerdem – gewissermaßen als Belohnung für Ihr Durchhaltevermögen – machen Sie Bekanntschaft mit der Programmierung von grafischen Benutzeroberflächen. Vermutlich werden Sie angenehm überrascht sein, wie einfach es ist – das inzwischen erworbene Programmier-Know-how vorausgesetzt –, Programme mit GUI-Elementen wie Fenstern, Schaltflächen, Menüs etc. zu versehen.



# 33

## Microsoft und das .NET

In Kapitel 1 wurde beschrieben, wie der Quellcode vom Compiler mehr oder weniger direkt in Maschinencode übersetzt wird. Danach liegt ein Programm in seiner ausführbaren Form vor. Jedes Betriebssystem besitzt jedoch seine Eigenheiten. Daher muss damit gerechnet werden, dass die fertig übersetzten Programme nur auf Betriebssystemen laufen, auf denen sie kompiliert wurden. Eine Antwort darauf gibt das .NET Framework oder kurz .NET (gesprochen »dotnet«), eine plattformübergreifende Initiative von Microsoft, die – zumindest vom Prinzip her – als ein Gegenstück zur Java Virtual Machine angesehen werden kann. Allerdings übersteigt die Funktionalität des .NET die der JVM bei weitem. Vermutlich haben Sie von Begriffen wie .NET Framework, Common Language Runtime (CLR) oder Managed Code schon gelesen oder gehört. In diesem Kapitel erfahren Sie, was es damit auf sich hat.

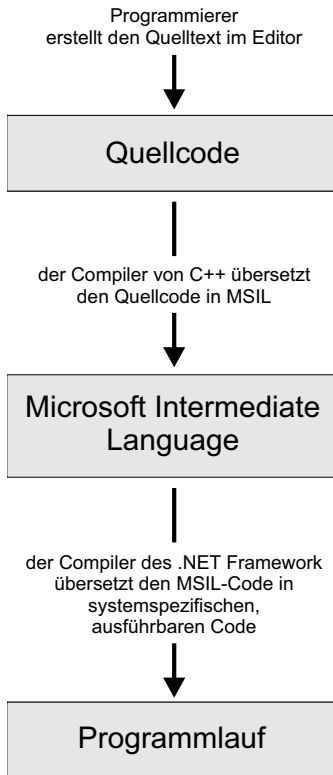
### 33.1 Microsoft Intermediate Language

Das .NET Framework fungiert praktisch als Mittler zwischen Programm und Betriebssystem. Der Quellcode wird vom .NET-eigenen C++-Compiler zunächst in eine Zwischenform umgewandelt (was natürlich auch für andere .NET-Compiler zutrifft, z.B. den von C# oder Visual Basic). Die aktuelle Bezeichnung für diesen Zwischencode ist Microsoft Intermediate Language (MSIL) oder auch nur Intermediate Language. Der MSIL-Code wird bei der Ausführung vom Compiler des .NET Framework in die Form gebracht, welche vom jeweiligen Betriebssystem verstanden wird. Es sind also insgesamt zwei Compiler am Werk, einmal derjenige der Programmiersprache und bei der Programmausführung der des .NET Framework. Natürlich setzt die Ausführung eines für das .NET kompilierten Programms voraus, dass eben dieses .NET Framework auf dem Computer installiert ist. Umgekehrt kann auf jedem Computer, auf dem das .NET Framework installiert ist – unabhängig vom verwendeten Betriebssystem – jede .NET-Anwendung ausgeführt werden.

#### Hinweis

Wie oben angedeutet, ist ein Vorläufer dieses Konzepts die Programmiersprache Java. Auch der Compiler von Java erzeugt zunächst so genannte *class*-Dateien, welche bei der Ausführung dann die Java Virtual Machine benötigen. Die Funktionsweise der JVM für Java-Programme entspricht in dieser Hinsicht der des .NET Framework – genauer der CLR, welche Bestandteil des .NET ist (dazu gleich mehr weiter unten) – für Anwendungen, die mit einer ans .NET angepassten Sprache geschrieben sind. Auf neueren Computern mit Windows-Betriebssystem ist das .NET Framework in der Regel vorinstalliert.

Gegenüber der Darstellung im ersten Kapitel (Abbildung 1.1 in Abschnitt 1.2.2 »Kompilierung«) ergibt sich damit folgendes Bild.



**Abbildung 33.1:** Vom Quellcode bis zur Programmausführung im .NET

Beachten Sie, dass diese Darstellung zu der aus Kapitel 1 praktisch nicht im Widerspruch steht. Nach der Übersetzung der Quellcodedatei durch den spracheigenen – fürs .NET angepassten – Compiler liegt ja ebenfalls ein fertiges Endprodukt vor. Nur benötigt dieses sozusagen ausführbare Programm eine besondere Laufzeitumgebung, eben das .NET Framework.

### Laufzeitumgebung

Benötigt ein Programm eine bestimmte Software, damit es ausgeführt werden kann, spricht man von einer Laufzeitumgebung (Engl. Runtime Environment). Im übertragenen Sinne stellt das Betriebssystem selbst natürlich ebenfalls eine Laufzeitumgebung für Programme dar.



## 33.2 Das .NET Framework

Wie gesagt, stellt das .NET Framework die Laufzeitumgebung für .NET-Programme dar, ähnlich der Java Virtual Machine für Java-Anwendungen. Während die JVM jedoch nur Java-Anwendungen unterstützt, dient das .NET Framework als Laufzeitumgebung für mehrere Programmiersprachen, die im Kontext des .NET Framework auch miteinander interagieren können. Weitere Programmiersprachen können für eine Unterstützung an das .NET Framework angepasst werden. Somit ist das .NET Framework nicht nur unabhängig vom Betriebssystem, sondern in gewisser Weise auch von der verwendeten Programmiersprache. Darüber hinaus stellt das .NET Framework eine Reihe von Hilfsmitteln zur Verfügung, welche die Programmentwicklung erleichtern und eine Verbesserung des Laufzeitverhaltens der Programme mit sich bringen.

### Hinweis

Die ursprünglich zuge dachte Bezeichnung für das .NET Framework war NGWS, was für »Next Generation Windows Services« steht und so viel heißt wie »Windows-Dienste der nächsten Generation«. Dieser Name (Next Generation Windows Services) macht zwei wesentliche Aspekte besonders deutlich: zum einen die Bereitstellung von zusätzlichen Diensten und zum anderen die enge Verbindung des .NET Framework mit dem Betriebssystem.

Ganz allgemein lässt sich feststellen, dass das .NET Framework nicht nur als Laufzeitumgebung dient, sondern zusätzlich den Programmcode verwaltet. Zum Beispiel übersetzt der Compiler des .NET Framework nicht auf ein Mal den ganzen Zwischencode und legt ihn in Maschinensprache ab, sondern er kompiliert die einzelnen Befehle erst dann, wenn sie gebraucht werden. Aus diesem Grund bezeichnet man den Compiler des .NET Framework auch als »Just-in-Time-Compiler« (oder kurz JITter). Der JITter geht dabei sehr effizient vor. Nehmen wir an, dass die Entwicklung einer Anwendung abgeschlossen ist. Das heißt, der MSIL-Code liegt vor und soll nun ausgeführt werden. Nehmen wir weiter an, es handle sich um eine Anwendung, die dem Benutzer mehrere mathematische Berechnungsfunktionen zur Verfügung stellt, zwischen denen er wählen kann. In diesem Fall könnte sich in etwa die in Abbildung 33.2 dargestellte Konstellation ergeben.

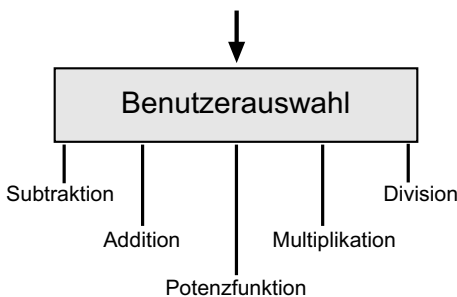


Abbildung 33.2: Verschiedene Programmzweige

Die Befehle an das Computerprogramm unterscheiden sich also je nach Benutzerauswahl. Wenn der Benutzer die Potenzfunktion des Programms wählt, kommen natürlich andere Anweisungen zur Ausführung, als wenn er sich etwa für die Addition entschieden hätte. Tatsächlich reagiert der Compiler des .NET Framework darauf, indem er hier nur die Befehle des gewählten Programmzweigs in Maschinencode übersetzt – im vorliegenden Fall also diejenigen für das Potenzieren. Alle anderen Anweisungen bleiben außen vor – zumindest so lange, bis derselbe Benutzer sich anderer Auswahloptionen bedient, wenn der Programmlauf erneut diese Stelle erreicht. Sollte nun während desselben Programmlaufs die Potenzfunktion durch den Anwender erneut aufgerufen werden, liegen die entsprechenden Befehle bereits in Maschinencode vor und der JITter des .NET Framework wird den entsprechenden MSIL-Code nicht erneut übersetzen. Er wird stattdessen gleich auf den für diesen Programmzweig bereits vorhandenen Maschinencode zurückgreifen.

### Hinweis

Wir sprechen oben vom JITter bzw. vom Compiler des .NET Framework. Dies ist eine Vereinfachung, welche der Anschaulichkeit dienen soll, und entspricht nicht ganz den Tatsachen. Denn eigentlich handelt es sich nicht nur um einen, sondern um mehrere Compiler, die zum .NET Framework gehören. Zum Beispiel wird das .NET Framework für Windows-Betriebssysteme mit drei verschiedenen Just-in-Time-Compilern ausgeliefert. Es existiert auch ein Programm namens JIT Compiler Manager, mit dem sich bestimmte Einstellungen bezüglich dieser Compiler vornehmen lassen. Unter anderem kann man mit dem JIT Compiler Manager auf Laufzeitverhalten und Speicherplatzbedarf von Programmen Einfluss nehmen. Dies sei nur ergänzend erwähnt; im Allgemeinen besteht keine Notwendigkeit, an den Standardeinstellungen Änderungen vorzunehmen.

## 33.3 Common Language Runtime

Das angeführte Beispiel ist natürlich nur eines von vielen für die Verwaltung des Codes im .NET. Genaugenommen ist es die Common Language Runtime (CLR), die den Microsoft Intermediate Language Code ausführt. Die Laufzeitumgebung – die CLR – ist also wie andere Komponenten, auf die wir hier nicht eingehen, in dem umfassenden Framework – dem .NET – enthalten.

Was das .NET gegenüber anderen auf Zwischencode basierenden Systemen vor allem unterscheidet, ist die Bereitstellung einer gemeinsamen Klassenbibliothek. In verschiedenen Sprachen geschriebene Programmteile können damit auf gemeinsame Ressourcen zugreifen. Eine Klasse, die Sie im Managed Code verwenden, ist also nicht sprachspezifisch, sondern Teil der auch für andere Sprachen verfügbaren .NET-Klassenbibliothek, der so genannten Framework Class Library (FCL).

**Hinweis**

In der Sprache des .NET bezeichnet man Programme, die innerhalb der CLR laufen, also für das .NET geschrieben wurden, als *managed*, herkömmliche Programme als *unmanaged*.

Zusammenfassend lässt sich feststellen, dass das .NET neben einer virtuellen Laufzeitumgebung – der Common Language Runtime – ein Rahmenwerk (Framework) von Klassenbibliotheken und Diensten bereitstellt (was natürlich angesichts seiner Komplexität eine grobe Vereinfachung darstellt). Im übertragenen Sinne handelt es sich also um eine übergreifende Programmierumgebung.

### 33.3.1 Garbage-Collection

Eine weitere Besonderheit des .NET ist die Garbage-Collection. Die CLR hat die vollständige Kontrolle über die Speicherverwaltung. Das heißt, sie regelt Anforderung und Freigabe von Speicher und anderen Ressourcen. Nicht mehr benötigte Objekte auf dem Heap werden automatisch entfernt. Allerdings erfolgt die Freigabe nicht sofort. Über den genauen Zeitpunkt entscheidet der Garbage-Collector der CLR. Die CLR überwacht auch Zugriffe auf Dienste, auf das Dateisystem oder Geräte und lehnt diese ab, wenn sie gegen Sicherheitsrichtlinien verstoßen.

### 33.3.2 Programmieren für .NET

Obwohl C++-Anwendungen traditionell direkt auf Betriebssystemebene ausgeführt werden und weder das .NET Framework noch eine spezielle Laufzeitumgebung benötigen, ist es für C++-Programmierer durchaus attraktiv, für .NET zu programmieren. Nicht nur, weil sie die schwierige und fehlerträchtige dynamische Speicherverwaltung in die vertrauenswürdigen Hände des Garbage-Collector legen können, nicht nur wegen des Austauschs von Programmmodulen über Sprachgrenzen hinweg, sondern auch wegen der Vielzahl erstklassiger Klassenbibliotheken, die das .NET Framework zur Verfügung stellt (beispielsweise für die GUI-Programmierung, mit der wir uns im nachfolgenden Kapitel beschäftigen).

Doch die Annehmlichkeiten des .NET Framework haben ihren Preis. Sie stehen nur zur Verfügung, wenn der C++-Programmierer verwalteten (*managed*) Code schreibt. Das heißt, der Programmierer muss die Hoheit über die dynamische Sprachverwaltung an den Garbage-Collector abgeben, was viele Programmierer wohl eher mit einem lachenden als einem weinenden Auge tun werden, und – was gravierender ist – der resultierende C++-Code ist nicht mehr ANSI-kompatibel, denn *Managed C++*-Code kann nur mithilfe des erweiterten Sprachdialekts C++/CLI erstellt werden.

**Hinweis**

C++/CLI ist der verbesserte Nachfolger der »Managed Extensions for C++«. Mehr zu C++/CLI erfahren Sie in Kapitel 34.



# 34

## Windows-Anwendungen

Zum Abschluss entwickeln wir nun ein kleines Programm mit grafischer Benutzeroberfläche. Thema ist wiederum das Ihnen aus Kapitel 22 bekannte Lottospiel. Allerdings werden wir uns damit begnügen, Zahlen auszulosen. Das entspricht, abgesehen von der Zusatzzahl, in etwa dem Algorithmus des letzten Beispiels von Abschnitt 22.1 »Im Programm Lottozahlen auslosen« aus Kapitel 22. Der praktische Nutzen der fertigen Anwendung besteht dann in einer Entscheidungshilfe beim Ausfüllen von Lottoscheinen. In erster Linie geht es jedoch darum, Ihnen exemplarisch die Vorgehensweise beim Erstellen von grafischen Benutzeroberflächen in modernen Entwicklungsumgebungen näher zu bringen.

Den folgenden Ausführungen liegt die Professional Edition des Visual Studio 2005 von Microsoft zugrunde. Die hier beschriebene Vorgehensweise sollte sich aber von anderen Versionen – und auch von anderen IDEs – im Wesentlichen kaum unterscheiden. Falls Sie noch kein Geld für eine Entwicklungsumgebung ausgeben möchten, empfehlen wir Ihnen, auf die kostenfrei erhältliche Visual C++ 2005 Express Edition von Microsoft zurückzugreifen (siehe Kasten »Visual C++ 2005 Express Edition« im folgenden Abschnitt).

### 34.1 Windows Forms

Im Visual Studio bilden Formulare die Basis jeder GUI-Anwendung – daher auch der Name Forms (Engl. »form« = »Formular«). Formulare sind nichts anderes als Fenster, was auch Zusatzfenster wie Dialogfelder einschließt. Auf einem Formular platzieren Sie die benötigten Steuerelemente (Textfelder, Schaltflächen, Dialogfelder usw.), welche in ihrer Gesamtheit schließlich das Aussehen Ihrer Anwendung bestimmen. Zu diesem Zweck stehen Ihnen eine Vielzahl von Hilfsmitteln zur Verfügung, unter anderem eine Palette von fertigen Steuerelementen, mit denen Sie auch bei der Entwicklung professioneller Anwendungen auskommen sollten.

#### Hinweis

Grundsätzlich stehen Ihnen in Visual Studio zwei Möglichkeiten offen: Entweder entwickeln Sie herkömmliche Windows-Anwendungen oder solche für das .NET Framework. Für die Ersteren sind die älteren MFC-Klassen, für die Letzteren die Windows Forms zuständig, die auch wir hier verwenden. Dabei handelt es sich um die Klassen, die im Namensraum `System.Windows.Forms` definiert sind.

Hinter jedem Steuerelement steht eine Klasse des .NET Framework. Sobald Sie Ihrem Formular ein Steuerelement hinzufügen, generiert Visual Studio im Hintergrund den

benötigten Code (z.B. Instanzieren eines Objekts der betreffenden Klasse). Die vordefinierte Funktionalität der eingefügten Steuerelemente führt in Verbindung mit dem von Ihnen zu entwickelnden Code in der Regel verhältnismäßig schnell zu den gewünschten Lösungen.

### Visual C++ 2005 Express Edition

Bei der Visual C++ 2005 Express Edition – oder kürzer: Visual C++ Express – handelt es sich um eine vollwertige Teilmenge des Visual Studio 2005 und nicht etwa um eine funktional eingeschränkte oder gar zeitlich befristete Demoversion. Was hier über Visual Studio 2005 gesagt wird, gilt deshalb im Wesentlichen auch für diese IDE. Falls Sie sich irgendwann die »große« Version von Visual Studio zulegen sollten, werden Sie durch Ihre Erfahrungen mit Visual C++ Express bestens vorbereitet sein. Im Übrigen eignet sich die Visual C++ 2005 Express Edition für den Hobbyprogrammierer ebenso wie für den professionellen Softwareentwickler. Einzige Einschränkung: Die MFC-Klassen stehen nicht zur Verfügung. Das heißt, Sie können mit Visual C++ Express ausschließlich Anwendungen für das .NET Framework erstellen – was aber im Hinblick auf die aktuelle Windows-Programmierung kein allzu großer Nachteil sein dürfte. Die Software ist konzipiert für Windows 2000 (ab Service Pack 4), Windows XP (ab Service Pack 2), Windows Server 2003 (ab Service Pack 1) und Windows Vista. Außerdem sollten Sie über mindestens 256 MB RAM und einen Prozessor mit wenigstens 1 Gigahertz verfügen (angegebene Mindestwerte: 192 MB und 600 Megahertz). Sie können die Visual C++ 2005 Express Edition unter <http://www.getexpress.de> kostenlos herunterladen (die URL ist ein Shortcut, Sie werden anschließend auf die Microsoft-Seite <http://www.microsoft.com/germany/msdn/vstudio/products/express/download.msp> umgeleitet). Im Installationsumfang sind die MSDN-Library und Microsoft SQL Server Express enthalten. Das .NET Framework wird – sofern noch nicht vorhanden – automatisch mitinstalliert. Die Installation selbst erweist sich in der Regel als unkompliziert.

### Tipp

Microsoft bietet auf der Internetseite <http://www.learnvisualstudio.net> eine Reihe von erstklassigen Videoanimationen zu Visual Studio, den Express-Versionen und dem .NET Framework an. Falls Sie sich dort registrieren, erhalten Sie in regelmäßigen Abständen Videos zum kostenlosen Download.

## 34.1.1 Ein neues Projekt beginnen

Zunächst legen wir in Visual Studio ein passendes Projekt an. Rufen Sie dazu den Menübefehl DATEI/NEU/PROJEKT... auf. Wählen Sie unter CLR die Projektvorlage WINDOWS FORMS-ANWENDUNG und geben Sie als Projektname *Lottospiel* ein (Abbildung 34.1).

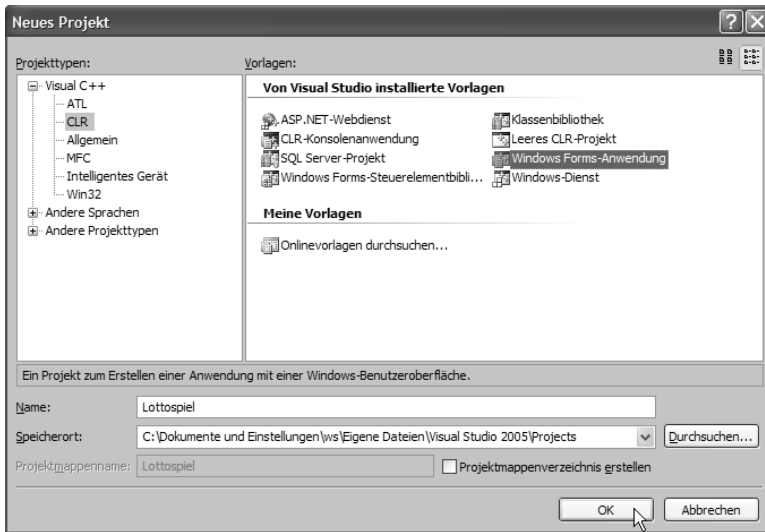


Abbildung 34.1: Auswahl der Projektvorlage »Windows Forms-Anwendung«

Visual C++ erzeugt daraufhin unter anderem den Code für das Hauptformular. Die Oberfläche der IDE sollte sich nun in etwa wie in Abbildung 34.2 präsentieren.

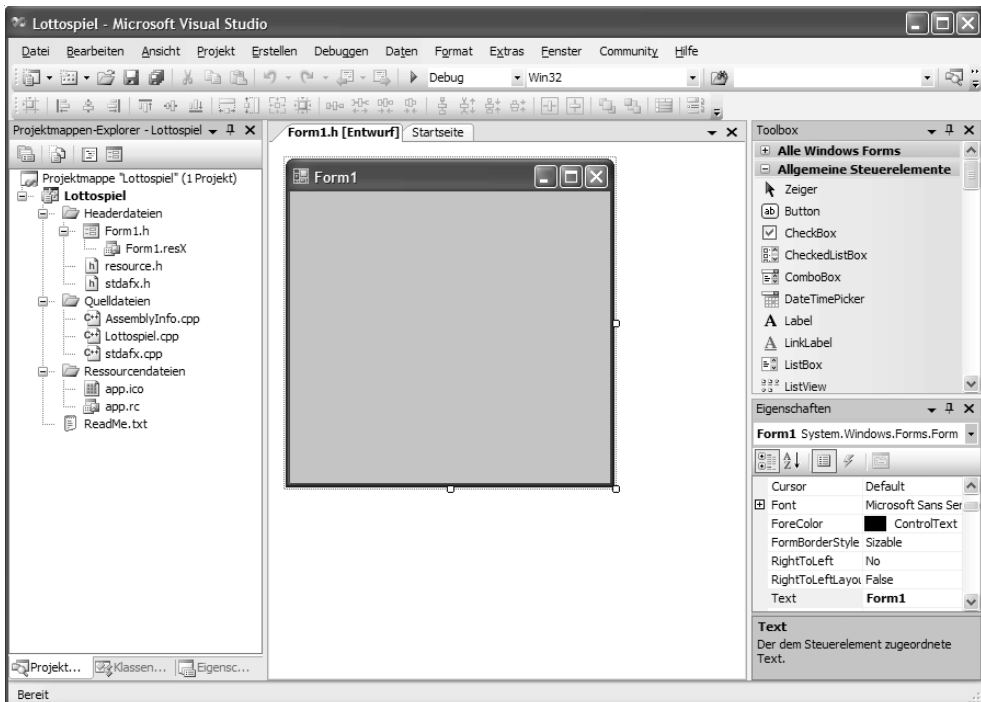


Abbildung 34.2: Projektmappe-Explorer, Windows Forms-Designer, Toolbox und Eigenschaftsfenster

Die Toolbox (rechts im Bild) und das Eigenschaftfenster (rechts, unterhalb der Toolbox) blenden Sie gegebenenfalls im Menü **ANSICHT/TOOLBOX** bzw. **ANSICHT/WEITERE FENSTER/EIGENSCHAFTENFENSTER** (oder Klick auf die entsprechenden Symbole in der Symbolleiste rechts oben) ein.

Bei der Windows Form, wie sie in der Entwurfsansicht zu sehen ist, handelt es sich um eine visuelle Darstellung des Fensters, wie es beim Starten der Anwendung erscheint. Der entsprechende Code, den Visual C++ generiert hat, befindet sich in den Dateien *Lottospiel.cpp* und *Form1.h* – wobei *Lottospiel.cpp* den Code zum Anzeigen des Formulars und *Form1.h* die Implementierung der Formalklasse enthält.

### Hinweis

Um gegebenenfalls von der Entwurfsansicht des Formulars in die Codeansicht zu wechseln, klicken Sie im Projektmappen-Explorer (links im Bild) mit der rechten Maustaste auf den Eintrag *Form1.h* und wählen im Kontextmenü den Befehl **CODE ANZEIGEN**. Die Entwurfsansicht können Sie jederzeit durch Doppelklick auf die Header-Datei des Formulars laden.

In der Toolbox besorgen wir uns nun die passenden Steuerelemente. Für unser Lottoprogramm benötigen wir insgesamt sechs Textfelder zum Anzeigen der Zahlen und eine Schaltfläche, mit der die Auslosung in Gang gesetzt werden soll. Die passenden Steuerelemente heißen **TEXTBOX** und **BUTTON**. Vorerst fügen wir unserem Formular neben der Schaltfläche nur ein einziges Textfeld hinzu. Nachdem wir dieses nach unseren Wünschen angepasst haben, werden wir es fünfmal kopieren. Wir sparen uns damit den Aufwand, die restlichen fünf Textboxen ebenfalls anpassen zu müssen.

### Tipp

Erweitern Sie in der Toolbox gegebenenfalls die Kategorie **ALLE WINDOWS FORMS**, um alle Steuerelemente auf einen Blick sehen zu können.

Um einem Formular ein Steuerelement hinzuzufügen, klicken Sie in der Toolbox auf den entsprechenden Eintrag und danach auf die gewünschte Stelle im Formular. Daraufhin wird das betreffende Steuerelement auf dem Formular abgelegt. Die Position kann durch Ziehen des Steuerelements nachträglich noch geändert werden.

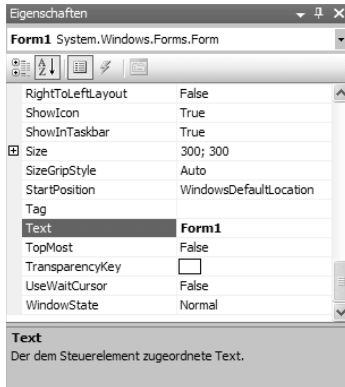
### Hinweis

Alternativ können Sie ein Steuerelement bei gedrückter Maustaste von der Toolbox auf das Formular ziehen.



### 34.1.2 Das Eigenschaftfenster im Visual Studio

Jedes Steuerelement – auch das Formular selbst – besitzt bestimmte Eigenschaften. Um diese zu bearbeiten, wählen Sie das Steuerelement mit einem Mausklick aus und begeben Sie anschließend in das Eigenschaftfenster. Dort können Sie zwischen der Eigenschaftenansicht und der Ereignisansicht wählen.



**Abbildung 34.3:** Eigenschaftenansicht des Eigenschaftfensters

Klicken Sie dazu auf das entsprechende Symbol in der oberen Symbolleiste des Eigenschaftfensters. In der Eigenschaftenansicht sehen Sie in der linken Spalte den Namen der jeweiligen Eigenschaft und in der rechten den dazugehörigen aktuellen Wert, den Sie dort auch neu setzen können. Der untere Teil des Eigenschaftfensters zeigt eine Beschreibung der jeweils ausgewählten Eigenschaft.

#### Hinweis

Denken Sie daran, dass jedes Steuerelement eine Instanz der betreffenden Steuerelement-Klasse ist. Wenn Sie bestimmte Eigenschaften (das Wort wird hier im üblichen Sinne verwendet) variieren, ändern Sie letztlich die entsprechenden Attributwerte (mit Attribut ist hier die Eigenschaft im programmiersprachlichen Sinne gemeint). Das heißt, Visual C++ erzeugt im Hintergrund den entsprechenden Code.

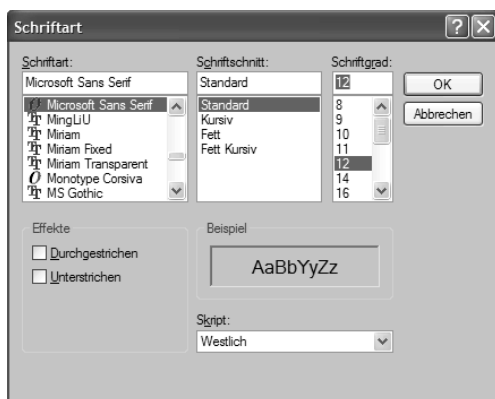
Als Erstes ändern wir den Wert für die Eigenschaft Text des Formulars in »Ihre Lottozahlen«. Klicken Sie dazu im Windows-Forms-Designer Ihr Formular an, lassen Sie sich im Eigenschaftfenster die Eigenschaftenansicht anzeigen und setzen Sie die Einfügemarke in die rechte Spalte neben Text. Dort löschen Sie den vorhandenen Eintrag »Form1« und tippen »Ihre Lottozahlen« ein. Damit legen Sie den Text für die Titelleiste des Formulars fest. Der neue Text wird dort angezeigt, sobald Sie mit dem Cursor das Feld verlassen haben.

In gleicher Weise ändern wir die Eigenschaften der TextBox wie folgt: Wir setzen die Breite des Textfelds auf 35 Pixel. Erweitern Sie dazu im Eigenschaftfenster die Eigenschaft Size, nachdem Sie das Textfeld in der Entwurfsansicht ausgewählt haben. Belassen Sie es für Height bei dem Wert 20 und geben Sie für Width den Wert 35 an.

Das Attribut Size der Font-Eigenschaft setzen wir auf den Wert 12. Damit legen wir für die Darstellung der Lottozahlen eine Schriftgröße von 12 pt fest. Die Eigenschaft Font umfasst wie die Eigenschaft Size mehrere Attribute. Zusammengesetzte Eigenschaften werden im Eigenschaftsfenster in einer Strukturansicht angezeigt, erkennbar an dem vorangestellten Plus-Zeichen (+). Um die Werte einzelner Attribute zu ändern, erweitern Sie den Eintrag für die Eigenschaft, indem Sie darauf doppelklicken. Anschließend verfahren Sie wie gerade bei dem Width-Attribut von Size.

## Tipp

Bei vielen Eigenschaften, so auch bei der Font-Eigenschaft, kann das Setzen der Attributwerte alternativ über einen Dialog erfolgen, was sehr praktisch ist, um mehrere Attribute in einem Durchgang zu ändern. In diesem Fall wird in der rechten Spalte eine mit Auslassungspunkten beschriftete Schaltfläche (»...«) angeboten. Klicken Sie diese entsprechend an.



**Abbildung 34.4:** Der Dialog zum Ändern der Eigenschaft »Font«. Er entspricht dem Dialog aus Textverarbeitungsprogrammen zur Anbringung zeichenorientierter Formatierungen.

Die Anchor-Eigenschaft der TextBox setzen wir auf None. Das ist notwendig, damit die Textfelder später beim Vergrößern des Anwendungsfensters zentriert bleiben. Klicken Sie im Eigenschaftsfenster in der Spalte neben Anchor auf das Symbol rechts und dann jeweils einmal auf die grauen Balken, bis kein grauer Balken mehr vorhanden ist.

Außerdem setzen wir die TextAlign-Eigenschaft auf Center, damit die Zahl später im Feld zentriert dargestellt wird.

Nun, da unser Textfeld das gewünschte Aussehen hat, müssen wir es nur noch fünfmal kopieren. Dazu wählen Sie es aus, kopieren es mit **(Strg) + [C]** in die Zwischenablage (oder wählen Sie den Menübefehl **BEARBEITEN/KOPIEREN**) und fügen es dann fünfmal in Ihr Formular ein (**(Strg) + [V]** oder **BEARBEITEN/EINFÜGEN**).

Nachdem sich nun alle benötigten Steuerelemente auf dem Formular befinden, folgt als Nächstes deren Anordnung. Zunächst sorgen wir dafür, dass die Textfelder mittig in

einer Reihe nebeneinanderliegen. Ihre Namen sollten dabei von links nach rechts `textBox1`, `textBox2`, ... `textBox6` lauten. Ziehen Sie also zuerst das `textBox1`-Textfeld nach links, platzieren Sie dann das `textBox2`-Textfeld daneben usw.

### Hinweis

Sie finden den Namen eines Steuerelements in der zweiten Spalte des Eigenschaftenfensters neben dem eingeklammerten Eintrag (Name). Außerdem wird er im oberen Listenfeld des Eigenschaftenfensters samt Klassenzugehörigkeit angezeigt. Wie Sie sich vermutlich denken können, handelt es sich bei diesem Namen um den Bezeichner für die Instanz des betreffenden Steuerelements. Über diesen (Instanz-)Namen können Sie im Code auf die verschiedenen Eigenschaften des Steuerelements zugreifen (beispielsweise die `Text`-Eigenschaft des Textfelds, die den im Textfeld angezeigten Text speichert).

Beim Ausrichten von Steuerelementen werden Sie von den Ausrichtungslinien unterstützt, den so genannten `SnapLines`. Alternativ können Sie die auszurichtenden Steuerelemente im Formular auswählen und dann die passenden Befehle im Menü `FORMAT` aufrufen. Wenn Sie mit den Textboxen so verfahren, können Sie auf bequeme Weise die horizontalen Abstände zwischen deren Rändern angleichen (`FORMAT/HORIZONTALER ABSTAND/ANGLEICHEN`) und sie auf dem Formular zentrieren (`FORMAT/AUF FORMULAR ZENTRIEREN/HORIZONTAL bzw. VERTIKAL`).

### Tipp

Um zuverlässig mehrere Steuerelemente gleichzeitig zu markieren, erweitern Sie in der Toolbox den Eintrag `ALLE WINDOWS FORMS` und klicken auf `ZEIGER`. Dann klicken Sie in der Entwurfsansicht das erste Steuerelement an und anschließend mit gedrückter `[Strg]`-Taste alle weiteren auszuwählenden Steuerelemente. Oder Sie rufen im Menü `BEARBEITEN` den Befehl `ALLE AUSWÄHLEN` auf und klicken mit gedrückter `[Strg]`-Taste auf die Steuerelemente, die nicht ausgewählt werden sollen. Des Weiteren besteht die Möglichkeit, bei gedrückter Maustaste einen Auswahlrahmen aufzuziehen, wodurch alle auf diese Weise erfassten Steuerelemente markiert werden.

Wir ändern nun noch für die Schaltfläche die `Text`-Eigenschaft zu »Lottozahlen auslosen« und setzen ihre `Anchor`-Eigenschaft auf `None`. Außerdem legen wir für den Text der Schaltfläche eine 10-Punkt-Schrift fest und passen ihre Größe an, indem wir das `Width`-Attribut der `Size`-Eigenschaft auf den Wert 160 und `Height` auf 30 setzen.

### Tipp

Um Größe bzw. Position eines Steuerelements zu ändern, können Sie auch mit gedrückter `[↕]`- bzw. `[Strg]`-Taste und den Pfeiltasten arbeiten. Ebenso kann die Größe des Steuerelements durch Ziehen der Anfasser (den Quadraten an den Begrenzungen des Steuerelements) bei gedrückter Maustaste geändert werden.

Als Letztes platzieren wir die Schaltfläche mittig über den Textfeldern. Damit ist das Design unserer Anwendung im Wesentlichen fertig. Das Ergebnis können wir uns sogleich ansehen, wenn wir die **[F5]**-Taste drücken.



Abbildung 34.5: Das fertige Formular bei gestartetem Programm

Es steht Ihnen natürlich frei, weitere Verbesserungen wie z.B. farbliche Hervorhebungen (Form-, Button- oder TextBox-Eigenschaften `ForeColor` bzw. `BackColor`) hinzuzufügen. Es sei Ihnen an dieser Stelle empfohlen, gegebenenfalls möglichst viel auszuprobieren, um Ihre IDE noch besser kennen zu lernen.

**Tipp**

Falls Ihnen die Größe des Formulars nicht zusagt, können Sie dieses analog zu den Steuerelementen durch Ziehen eines Anfassers bei gedrückter Maustaste ändern. Vergessen Sie nicht, das Formular zunächst mit einem Klick auszuwählen.

### 34.1.3 Weitere Steuerelemente

Visual Studio stellt Ihnen als Entwickler von Windows-Anwendungen eine Reihe von Steuerelementen zur Auswahl, wie Sie diese als Anwender im Umgang mit professionellen Programmen gewöhnt sind. Eine Auflistung der am häufigsten verwendeten Steuerelemente können Sie der folgenden Tabelle entnehmen.

Steuerelement	Beschreibung
Label	Beschreibungstext, in der Regel als Beschriftung für andere Steuerelemente gedacht
TextBox	Textfeld zur Eingabe von Text
RichTextBox	Textfeld mit umfangreichen Formatierungsmöglichkeiten
Button	Schaltfläche zum Anklicken
RadioButton	Optionsschaltfläche, die aktiviert oder deaktiviert werden kann

Tabelle 34.1: Steuerelemente mit Beschreibung

Steuerelement	Beschreibung
CheckBox	Kontrollkästchen, das aktiviert oder deaktiviert werden kann
MenuStrip	Zeigt eine Menüleiste in bekannter Form, auch kaskadiert, an
ToolTip	QuickInfo für andere Steuerelemente



**Tabelle 34.1:** Steuerelemente mit Beschreibung (Forts.)

RadioButton- und CheckBox-Steuerelemente besitzen eine ähnliche Funktion. Beide stellen Auswahlmöglichkeiten bereit, die ein Benutzer durch Anklicken aktivieren oder deaktivieren kann. Während sich die Optionsfelder eines RadioButton-Steuerelements jedoch gegenseitig ausschließen – das heißt von mehreren vorhandenen Optionen kann vom Benutzer jeweils nur eine ausgewählt werden –, können gleichzeitig mehrere CheckBoxen (Kontrollkästchen) aktiviert sein.

**Hinweis**

RadioButton-Steuerelemente, die zusammen in einem Formular, einem so genannten Panel oder einer GroupBox liegen (zwei weitere Steuerelemente, die als Container für andere Steuerelemente dienen), bilden eine Gruppe. Wenn ein Benutzer ein Optionsfeld innerhalb einer Gruppe aktiviert, werden die anderen Optionsfelder automatisch deaktiviert.

Mit einem MenuStrip richten Sie am oberen Rand eines Formulars ein Standardmenü ein. Zum Hinzufügen eines MenuStrip-Steuerelements in Ihr Formular gehen Sie in der oben beschriebenen Weise vor. Ziehen Sie also das Steuerelement auf das Formular oder klicken Sie je einmal in der Toolbox auf das MenuStrip-Symbol bzw. den Text MENUSTRIP und anschließend an eine beliebige Position innerhalb des Formulars. Danach erscheint im unteren Bereich des Entwurfsfensters, dem so genannten Komponentenfach, ein Eintrag für das MenuStrip-Steuerelement. (Das Komponentenfach dient zur Aufbewahrung von Steuerelementen, die zur Laufzeit entweder nicht sichtbar sind, z.B. eines Timers, oder die einen festen Platz haben, wie es bei einem Standardmenü der Fall ist; das Komponentenfach ist entsprechend nur im Entwurfsmodus, nicht aber zur Laufzeit sichtbar.)

Nach Ablegen des MenuStrip-Steuerelements auf dem Formular wird Ihnen oben in der Menüleiste für die erste Auswahlmöglichkeit ein Textfeld angeboten (klicken Sie gegebenenfalls das MenuStrip-Steuerelement im Komponentenfach an, falls diese Anzeige inzwischen nicht mehr sichtbar ist). Sie können sogleich mit der Eingabe des Textes für die erste Auswahlmöglichkeit beginnen. (Sie müssen dazu ausdrücklich nicht in die Menüleiste klicken, es genügt tatsächlich, wenn das MenuStrip-Steuerelement aktiv ist. Mit dem ersten Tastendruck wird die Einfügemarke in ein dann sichtbares Textfeld der Menüleiste gesetzt.) Beenden Sie die Eingabe mit . Danach erscheint rechts davon ein weiteres Textfeld für die nächste Auswahlmöglichkeit und unterhalb davon ein Textfeld für Auswahlmöglichkeiten der zweiten Ebene – vergleichbar z.B. mit den Menüs DATEI oder BEARBEITEN (für die erste Ebene) bzw. BEARBEITEN/KOPIEREN (zweite Ebene), wie sie in fast jedem Windows-Programm vorhanden sind. Durch Drücken von  werden Ihnen jeweils neue leere Felder zum Erstellen weiterer Auswahlmöglichkeiten angeboten.

Sobald Sie das Menü vollständig aufgefüllt haben, entziehen Sie dem MenuStrip-Steuer-element den Fokus, indem Sie mit der Maus einfach in einen anderen Bereich des Formulars klicken.

### Hinweis

Um ein MenuStrip-Steuerelement nachträglich weiterzubearbeiten, klicken Sie das Steuerelement im Komponentenfach an oder klicken in die obere Menüleiste. Um einem bestimmten Menüeintrag weitere Untereinträge hinzuzufügen, klicken Sie auf den Eintrag der übergeordneten Ebene. Mit **[Entf]** lassen sich Menüeinträge nachträglich entfernen.

Ein ToolTip-Steuerelement können Sie fast jedem beliebigen anderen Steuerelement zuordnen. Es zeigt eine QuickInfo an, wenn ein Benutzer den Mauszeiger auf das betreffende Steuerelement bewegt und ein paar Augenblicke wartet.

Wenn Sie Ihrem Formular ein ToolTip-Element aus der Toolbox hinzufügen, erscheint wie beim MenuStrip-Steuerelement ein entsprechendes Symbol im Komponentenfach. Um nun dieses ToolTip-Element als QuickInfo für ein Steuerelement des Formulars festzulegen, gehen Sie wie folgt vor:

- Wählen Sie das Steuerelement – beispielsweise die Schaltfläche Ihres Lottoprogramms – aus, welches Sie mit der QuickInfo verbinden möchten.
- Legen Sie im Eigenschaftenfenster für den Wert `ToolTip` auf `toolTip1` einen passenden Text fest. Dieser erscheint dann als QuickInfo, wenn ein Benutzer den Mauszeiger auf dem Steuerelement positioniert.
- Mit der Eigenschaft `ToolTipTitle` des ToolTip-Elements können Sie zusätzlich den Titel der QuickInfo bestimmen.

### Hinweis

Wenn Sie ein ToolTip-Element in Ihrem Formular ablegen, wird allen sich dort befindlichen Steuerelementen – auch dem Formular selbst – eine Eigenschaft `ToolTip` auf *Name\_des\_Tooltip\_Elements* hinzugefügt. Um für ein bestimmtes Steuerelement eine QuickInfo einzurichten, müssen Sie dieser Eigenschaft im Eigenschaftenfenster lediglich einen Wert zuweisen. Sie benötigen also keinesfalls für jedes Steuerelement ein eigenes ToolTip-Element.

## 34.2 Steuerelemente mit Code verbinden - Ereignisbehandlungsroutinen

Um auf unser Lottoprogramm zurückzukommen: Jedem Steuerelement sind eine Reihe von Ereignissen zugeordnet, die beim Programmlauf je nach Benutzerverhalten ausgelöst werden. Für jedes dieser Ereignisse kann eine Methode eingerichtet werden, die jedes Mal ausgeführt wird, wenn das Ereignis während der Programmausführung auftritt.

### Hinweis

Methoden, die auf ein Ereignis reagieren, werden auch Ereignis- oder Eventhandler genannt.

Klickt der Benutzer beispielsweise auf eine Schaltfläche, so löst er damit deren Click-Ereignis aus. Falls Sie für dieses Ereignis einen Ereignishandler definiert haben, wird dieser ausgeführt, ansonsten hat das Ereignis keine weitere Auswirkung.

### Hinweis

Die Ereignisse für ein bestimmtes Steuerelement können Sie sich im Eigenschaftenfenster anzeigen lassen, indem Sie dort in die Ereignisansicht wechseln (Klick auf das Blitz-Symbol). Sobald Sie die Einfügemarke auf ein bestimmtes Ereignis setzen, erhalten Sie unten im Eigenschaftenfenster eine Beschreibung, wann dieses Ereignis ausgelöst wird.

In unserem Programm sollen nun sechs Lottozahlen ausgelost und in die Textfelder geschrieben werden, wenn ein Benutzer auf die Schaltfläche klickt. Wir müssen also eine Methode, die genau dies leistet, als Ereignishandler mit dem Click-Ereignis unserer Schaltfläche verbinden. Im Fall des Click-Ereignisses für Schaltflächen müssen wir dazu nicht einmal über das Eigenschaftenfenster gehen; es reicht aus, in der Entwurfsansicht einen Doppelklick auf die Schaltfläche auszuführen. Die IDE legt daraufhin eine Methode `button1_Click()` als Ereignishandler an und lädt das zugehörige Codegerüst in den Editor.

### Hinweis

Jedes Steuerelement besitzt ein Standardereignis. Für Schaltflächen ist dies das Click-Ereignis, für Textfelder ist es z.B. das `TextChanged`-Ereignis, welches eintritt, sobald sich der Wert des Textfelds ändert. Um für Standardereignisse eine vordefinierte – zunächst leere – Ereignishandler-Methode anzulegen, genügt der gerade erwähnte Doppelklick auf das jeweilige Steuerelement. Für die anderen Ereignisse legen Sie die entsprechenden (ebenso zunächst leeren) Methoden an, indem Sie im Eigenschaftenfenster den Ereignisnamen doppelt anklicken (auch eine Methode für das Standardereignis kann so alternativ angelegt werden).

Hier der Code für die `button1_Click()`-Methode:

```
int lottozahlen[6], i;
Random^ generator = gcnew Random();

do {
    for(i=0; i<6; i++)
        lottozahlen[i] = generator->Next(1, 50);
} while (doppelte(lottozahlen, 6));

textBox1->Text = lottozahlen[0].ToString();
textBox2->Text = lottozahlen[1].ToString();
textBox3->Text = lottozahlen[2].ToString();
textBox4->Text = lottozahlen[3].ToString();
textBox5->Text = lottozahlen[4].ToString();
textBox6->Text = lottozahlen[5].ToString();
```

## CD-ROM

Das Visual-Studio-Projekt finden Sie als *Lottospiel.zip* im Ordner *Beispiele/K34* auf der Buch-CD. Die – eigenständig ausführbare – *.exe*-Datei befindet sich nach dem Entpacken von *Lottospiel.zip* im *Debug*-Verzeichnis. Laden Sie die Datei *Lottospiel.sln*, wenn Sie das Projekt in Visual Studio öffnen wollen – die Datei befindet sich im Projektverzeichnis. Wählen Sie dazu im Menü **DATEI** den Befehl **ÖFFNEN** und dann **PROJEKTMAPPE....** Wählen Sie im sich öffnenden Dialog *Lottospiel.sln* aus und klicken Sie auf **ÖFFNEN**.

Der Algorithmus dürfte Ihnen von Kapitel 22 bekannt sein. Allerdings verwendet Visual Studio 2005 für den Aufbau der Benutzeroberfläche Managed Code und C++/CLI (eine proprietäre Sprachversion von Microsoft, siehe Kapitel 33). Augenfällig ist die Syntax ^ für Zeiger auf Objekte sowie das zu verwendende Schlüsselwort `gcnew` an Stelle von `new`.

## Hinweis

Da die CLR alle Klassenobjekte auf dem Heap anlegt, muss der Zugriff auf Methoden und Eigenschaften dieser Objekte in C++/CLI mit dem Pfeiloperator in der Form `Objektname->Methode` bzw. `Objektname->Eigenschaft` erfolgen (zum Pfeiloperator siehe Kapitel 28, Abschnitt 28.2.4 »Elementzugriff über Zeiger«).

Für das Auslosen der Zahlen bedienen wir uns der `Random`-Klasse des .NET mit ihrer Methode `Next()`. Schließlich schreiben wir die aufsteigend sortierten Zahlen nacheinander in die Textfelder, indem wir sie der `Text`-Eigenschaft der `TextBox`-Instanzen zuweisen. Für den Zugriff verwenden wir den Pfeiloperator `->`. Allerdings müssen wir die Integer-Zahlen vor der Zuweisung noch mit der Methode `ToString()` in Strings umwandeln.



Die Prüfung auf doppelte Zahlen in der do-while-Schleifenbedingung erfolgt mit unserer Funktion `doppelte()` aus Kapitel 21. Diese verwendet die Funktion `sortiereArr()`, diese wiederum `ArrMinIndex()`. Das heißt, wir müssen diese drei Funktionen aus unserer Funktionsbibliothek von Kapitel 21 im Programmcode noch definieren bzw. bekannt machen. Dazu genügt es, wenn wir den Code dieser Funktionen einfach unterhalb der `button1_Click()`-Methode einfügen. Die Reihenfolge ist beliebig, da wir uns ja innerhalb der Klasse `Form1` befinden.

```
int doppelte(int *arr, int elemente)
{
    sortiereArr(arr, elemente);
    for (int i = 0; i < elemente - 1; i++)
    {
        if (arr[i] == arr[i + 1])
            return 1;
    }
    return 0;
}

int ArrMinIndex(int *arr, int ugrenze, int ogrenze)
{
    int min_index = ugrenze;
    for(int i = ugrenze + 1; i <= ogrenze; i++)
    {
        if(arr[i] < arr[min_index])
            min_index = i;
    }
    return min_index;
}

void sortiereArr(int *arr, int elemente)
{
    int ablage;
    for(int i=0; i < elemente; i++)
    {
        ablage = arr[ArrMinIndex(arr, i, elemente - 1)];
        arr[ArrMinIndex(arr, i, elemente-1)] = arr[i];
        arr[i] = ablage;
    }
}
```

Hinweis

Alternativ können Sie auf die bereits bestehenden Dateien aus Kapitel 22, *function.cpp* und *function.h*, zurückgreifen, indem Sie diese dem Projekt hinzufügen. Zunächst kopieren Sie beide Dateien am besten in das Projektverzeichnis. Klicken Sie dann im Projektmappen-Explorer auf den Eintrag **HEADERDATEIEN** bzw. **QUELLDATEIEN** und wählen Sie **HINZUFÜGEN/VORHANDENES ELEMENT...** Wählen Sie im weiteren Dialog die betreffenden Dateien und klicken Sie dann auf **HINZUFÜGEN**. Bevor Sie die in *function.cpp* definierten Funktionen im Lottospiel-Projekt verwenden können, müssen Sie noch folgende Änderungen vornehmen:

- Löschen Sie die Zeile `using namespace std;` in *function.cpp*.
- Fügen Sie dort `#include "stdafx.h"` ein.
- Binden Sie in *Form.h* mit `#include "function.h"` die Header-Datei *function.h* ein.

Nach Starten der Anwendung und Betätigen der Schaltfläche »Lottozahlen auslosen« sollten nun in den Textfeldern die ausgelosten Zahlen in aufsteigender Reihenfolge erscheinen.



Abbildung 34.6: Das Lottoprogramm in Aktion

In Tabelle 34.2 sehen Sie eine Übersicht der Standardereignisse für die weiter oben genannten Steuerelemente nebst Beschreibung.

Steuerelement	Standardereignis	Beschreibung
Label	Click	Tritt ein, wenn ein Benutzer das Label anklickt
TextBox	TextChanged	Tritt mit jeder Änderung des enthaltenen Textes ein
RichTextBox	TextChanged	
Button	Click	Tritt ein, wenn ein Benutzer die Schaltfläche anklickt
RadioButton	CheckedChanged	Tritt ein, wenn sich der Wert der Checked-Eigenschaft ändert
CheckBox	CheckedChanged	

Tabelle 34.2: Die wichtigsten Steuerelemente und die dazugehörigen Standardereignisse

Steuerelement	Standardereignis	Beschreibung
MenuStrip	ItemClicked	Tritt ein, sobald ein Benutzer auf einen Menüeintrag klickt
ToolTip	Popup	Tritt unmittelbar vor dem Anzeigen der QuickInfo ein

**Tabelle 34.2:** Die wichtigsten Steuerelemente und die dazugehörigen Standardereignisse (Forts.)

Die Checked-Eigenschaft von RadioButton- und CheckBox-Steuerelementen besitzt den Wert true, wenn das jeweilige Steuerelement aktiviert ist, andernfalls false (Anzeige True bzw. False im Eigenschaftfenster). Deren Wert ändert sich also, wenn ein Benutzer die entsprechende Option aktiviert bzw. – bei RadioButtons durch Auswahl einer anderen Option – wieder deaktiviert.



# ANHANG



# A CD-ROM zum Buch

Die CD-ROM zum Buch enthält folgende Verzeichnisse:

- **Borland-Compiler:** Hier findet sich der Borland C++-Compiler (*freecommand-Linetools.exe*). Die Installation ist in Kapitel 2 beschrieben.
- **Beispiele:** Enthält, nach Kapiteln geordnet, alle Beispiele des Buches.





# B Schlüsselwörter in C++

Schlüsselwörter			
asm	auto	bad_cast	bad_typeid
bool	break	case	catch
char	class	const	const_cast
continue	default	delete	do
double	dynamic_cast	else	enum
except	explicit	extern	false
finally	float	for	friend
goto	if	inline	int
long	mutable	namespace	new
operator	private	protected	public
register	reinterpret_cast	return	short
signed	sizeof	static	static_cast
struct	switch	template	this
throw	true	try	type_info
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	while		



## C

# Prioritätsreihenfolge der C++-Operatoren

Priorität	Operator	Bedeutung	Assoziativität
17	::	Bereichszugriff (unär)	keine
17	::	Bereichszugriff (binär)	von links nach rechts
16	(...)	Funktionsaufruf	von links nach rechts
16	[...]	Arrayindex	von links nach rechts
16	->	Elementauswahl	von links nach rechts
16	.	Elementauswahl	von links nach rechts
16	static_cast<Typ>	zur Übersetzungszeit geprüfte Konvertierung	keine
16	dynamic_cast<Typ>	zur Laufzeit geprüfte Konvertierung	keine
16	reinterpret_cast<Typ>	ungeprüfte Konvertierung	keine
16	const_cast<Typ>	const-Konvertierung	keine
15	++	Inkrement	keine
15	--	Dekrement	keine
15	!	logische Negation	keine
15	~	bitweise Negation	keine
15	-	unäres Minuszeichen	keine
15	+	unäres Pluszeichen	keine
15	&	Adresse	keine
15	*	Dereferenzierung	keine
15	sizeof	Größe in Byte	keine
15	new	Anforderung von Speicher	keine
15	delete	Freigabe von Speicher	keine
15	(Typ)	Cast (Typkonvertierung)	von rechts nach links
14	.*	Elementauswahl	von links nach rechts
14	->*	Elementauswahl	von links nach rechts
13	*	Multiplikation	von links nach rechts
13	/	Division	von links nach rechts
13	%	Modulo	von links nach rechts
12	+	Addition	von links nach rechts

Priorität	Operator	Bedeutung	Assoziativität
12	-	Subtraktion	von links nach rechts
11	<<	Linksschieben	von links nach rechts
11	>>	Rechtsschieben	von links nach rechts
10	<	kleiner als	von links nach rechts
10	<=	kleiner gleich	von links nach rechts
10	>	größer als	von links nach rechts
10	>=	größer gleich	von links nach rechts
9	==	Gleichheit	von links nach rechts
9	!=	Ungleichheit	von links nach rechts
8	&	bitweises Und	von links nach rechts
7	^	bitweises Exklusiv-Oder	von links nach rechts
6		bitweises Oder	von links nach rechts
5	&&	logisches Und	von links nach rechts
4		logisches Oder	von links nach rechts
3	? :	Bedingungsoperator	von rechts nach links
2	=	Zuweisung	von rechts nach links
2	*=	Multiplikation und Zuweisung	von rechts nach links
2	/=	Division und Zuweisung	von rechts nach links
2	%=	Modulo und Zuweisung	von rechts nach links
2	+=	Addition und Zuweisung	von rechts nach links
2	-=	Subtraktion und Zuweisung	von rechts nach links
2	<<=	Linksschieben und Zuweisung	von rechts nach links
2	>>=	Rechtsschieben und Zuweisung	von rechts nach links
2	&=	bitweises Und und Zuweisung	von rechts nach links
2	=	bitweises Oder und Zuweisung	von rechts nach links
2	^=	bitweises Exklusiv-Oder und Zuweisung	von rechts nach links
1	,	Sequenzoperator	von links nach rechts
1	throw	Ausnahme werfen	keine

# D

## ASCII-Tabelle

Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen	Dez.	Hex.	Zeichen
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	\$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL



# E

## Glossar

**Abgeleitete Klassen** ->Klassen, die von einer oder mehreren anderen Klassen erben (->Vererbung). Die Bezeichnung »abgeleitete Klasse« existiert nur in Bezug auf eine ->Basisklasse.

**Anweisungen** Im Code enthaltene Programmteile, die die Ursache dafür bilden, dass beim späteren Programmaufbau etwas geschieht. Jede Anweisung hat dabei eine ganz spezielle Wirkung und setzt sich gewöhnlich aus einer Reihe von ->Textbausteinen zusammen. Das Anweisungsende jeder C++-Anweisung bildet ein Semikolon.

**Argumente** Werte, die beim Aufruf einer ->Funktion oder ->Methode an diese übergeben werden.

**Arrays** Menge von Variablen gleichen ->Datentyps, die nebeneinander im Speicher liegen und daher unter einem einzigen Bezeichner mittels Indexangabe angesprochen werden können.

**Attribute** Datenelemente einer ->Klasse.

**Basisklassen** ->Klassen, von denen eine oder mehrere andere Klassen erben (->Vererbung). Dabei bezieht sich die Bezeichnung »Basisklasse« allein auf das Verhältnis zur jeweils abgeleiteten (erbenden) Klasse.

**Benutzerdefinierte Datentypen** ->Datentypen, die der Programmierer im Code selbst definieren kann. Sie setzen sich stets in irgendeiner Form aus elementaren Datentypen zusammen. Ein Beispiel hierfür ist die Definition einer ->Klasse.

**Block** Alle ->Anweisungen, die zwischen einer öffnenden und einer schließenden geschweiften Klammer stehen ({...}), bilden zusammen einen Block.

**Compiler** Software zum Übersetzen des Quellcodes in Maschinencode.

**Datentypen** Bestimmen die möglichen Werte, die Datenobjekte annehmen können, sowie die Operationen, die mit diesen Werten ausgeführt werden können. Jedem Datenobjekt ist genau ein Datentyp zugeordnet.

**Dekrementieren** Vermindern des Wertes einer numerischen ->Variablen um den Betrag von eins.

**Destruktor** Gegenstück zum ->Konstruktor. Besondere ->Methode, die automatisch aufgerufen wird, wenn die Lebensdauer eines Klassenobjekts erlischt, das heißt wenn der für das Objekt reservierte Speicherbereich vom Programm wieder freigegeben wird.

**Eigenschaften** Datenelemente einer ->Klasse. Im Allgemeinen sind mit dieser Bezeichnung nur die nicht statischen Datenelemente gemeint, also solche, die sich auf einzelne Klassenobjekte beziehen. Demgemäß spricht man von den Eigenschaften eines

Objekts und meint damit diejenigen Datenelemente, die nur diesem Objekt gehören (siehe auch ->statische Datenelemente).

**Elementare Datentypen** ->Standarddatentypen.

**Elemente** einer ->Klasse sind alle für diese definierten Attribute und ->Methoden, statische wie nicht statische.

**Elementfunktionen** Andere Bezeichnung für ->Methoden.

**Funktionen** Unterprogramm, das einen Namen hat und dem ->Argumente übergeben werden können. Die Funktion kann außerdem nach Beendigung einen Wert zurückgeben. Sie besteht aus einem Funktionskopf und einem Funktionsrumpf. Der Code wird in aller Regel in mehrere Funktionen aufgeteilt. In der Regel wird eine Funktion eine fest umrissene Teilaufgabe abdecken.

**Heap** Spezieller Bereich des Arbeitsspeichers, in dem von Programmen zur Laufzeit Speicher angefordert und auch wieder freigegeben werden kann.

**IDE** Abkürzung für »Integrated Development Environment« (auf Deutsch »integrierte Entwicklungsumgebung«). Software, in der alle notwendigen Programmierwerkzeuge unter einer Benutzeroberfläche zusammengefasst sind.

**Inkrementieren** Erhöhen des Wertes einer numerischen ->Variablen um den Betrag von eins.

**Instanz** Andere Bezeichnung für ->Objekt.

**Kapselung** Prinzip, den inneren Aufbau einer ->Klasse nach außen hin zu verbergen. Die Benutzung der Klasse erfolgt dabei allein über genau definierte Schnittstellen, z.B. über öffentliche ->Methoden. Solange diese Schnittstellen – also z.B. ->Signatur oder Rückgabewert von public-Methoden – nicht geändert werden, haben Änderungen innerhalb der Klasse keinerlei Auswirkungen auf Programme, die die Klasse verwenden, und müssen von diesen daher auch nicht berücksichtigt werden.

**Klassen** Eine Art Schablone, die einen eigenen ->Datentyp beschreibt. Sie ist sozusagen der Bauplan für künftige ->Variablen – »Objekte« oder »Instanzen« genannt – dieser Klasse (dieses Typs). Die Objekte vereinen Daten und ->Funktionen. Letztere werden in Verbindung mit Klassen als ->Methoden bezeichnet.

**Konstruktor** Spezielle ->Methode, die für jedes Klassenobjekt (->Klassen) bei dessen Instanzierung (der Erzeugung eines konkreten Objekts) automatisch aufgerufen wird.

**Leere Anweisung** ->Anweisung, die keine ->Textbausteine enthält, also allein aus dem Anweisungsendezeichen – einem Semikolon – besteht.

**Linker** Programm, das im Zuge des Übersetzungsvorgangs aus einer bzw. mehreren ->Objektdateien das ausführbare Programm erzeugt.

**Literal** Repräsentiert im Quelltext einen konstanten Wert und zwar in direkter, nicht in symbolischer Form.

**L-Werte** Solche Werte, die auf der linken Seite einer Zuweisung stehen können, also »Links«-Werte. Dazu zählen alle Datenobjekte, die Platz im Arbeitsspeicher beanspru-



chen, wie z.B. ->Variablen, aber gegebenenfalls auch ->Zeiger und ->Referenzen, da diese ja einen Verweis auf bzw. einen Alias für solche Datenobjekte darstellen.

**Methoden** ->Funktionen, die für eine ->Klasse definiert sind, auch »Elementfunktionen« genannt.

**Objekt** ->»Variable« vom Typ einer ->Klasse. Eine andere Bezeichnung für »Objekt« ist »Instanz«.

**Objektdatei** Vom Compiler erzeugte Datei, die bereits den übersetzten Maschinencode enthält, aber in der noch keine Funktionsaufrufe (->Funktion) aufgelöst sind. Dafür ist der ->Linker zuständig. Der Begriff »Objektdatei« hat trotz des Namens nichts mit objektorientierter Programmierung zu tun.

**OOP** Abkürzung für »objektorientierte Programmierung«.

**Operatorfunktionen** Speziell definierte ->Funktionen, mit deren Hilfe sich bestimmte Operatoren für ->Klassen ->überladen lassen. Die entsprechenden Operationen können danach in der gewohnten Form, mit Bezeichnern von Klassenobjekten als Operanden, notiert werden, so wie man es von den einfachen ->Datentypen her gewohnt ist.

**Parameter** ->Variablen einer ->Funktion oder ->Methode, die bei deren Aufruf mit den ->Argumenten initialisiert werden. Allerdings bezeichnet man auch Letztere verallgemeinernd als Parameter. Um den Unterschied deutlich zu machen, nennt man die in der Funktion bzw. Methode deklarierten Variablen »formale Parameter«, die im Funktions- bzw. Methodenaufruf angegebenen Argumente »aktuelle Parameter«.

**Präprozessor** Teil der zum Übersetzen von C++-Programmen in Maschinencode benötigten Software. Der Präprozessor führt Textersetzungen durch und entfernt Kommentare aus dem Quellcode, bevor ihn der eigentliche Compiler erhält.

**private** ->Zugriffsspezifizierer für Klassenelemente.

**protected** ->Zugriffsspezifizierer für Klassenelemente. Ein protected-Element verhält sich bezüglich seiner ->Klasse wie ein private-Element. Unterschiede bestehen nur in Bezug auf ->abgeleitete Klassen.

**public** ->Zugriffsspezifizierer für Klassenelemente.

**Referenzen** Alternative Namen für Datenobjekte mit der Funktion eines Verweises. Referenzen sind eine Neuerung von C++ gegenüber der Programmiersprache C. Sie sind in der Regel einfacher zu handhaben und weniger fehlerträchtig als ->Zeiger.

**R-Werte** Werte, die ausschließlich auf der rechten Seite (das »R« steht für »rechts«) einer Zuweisung auftreten können, nie auf der linken. Darunter fallen etwa ->Literele oder symbolische Konstanten.

**Schleifen** Besondere Konstrukte der Programmiersprache, die es erlauben, im Programm Wiederholungsabläufe einzurichten.

**Schlüsselwörter** Namen, die fest zur Programmiersprache gehören und die eine ganz bestimmte Bedeutung haben.

**Signatur einer Funktion/Methode** Die Kombination aus Bezeichner plus Parameterliste (->Parameter) einer ->Funktion bzw. einer ->Methode. Über die Signatur kann eine Funktion bzw. Methode eindeutig identifiziert werden.

**Standarddatentypen** ->Datentypen, die direkt zur Programmiersprache gehören. Das Gegenstück dazu sind ->benutzerdefinierte Datentypen.

**Standardkonstruktor** ->Konstruktor ohne ->Parameter.

**Statische Datenelemente** ->Attribute einer ->Klasse, die für alle ->Objekte dieser Klasse gleichzeitig verfügbar sind, sich also nicht auf eine bestimmte Instanz beziehen. Anders als das bei nicht statischen Klassenattributen der Fall ist, existiert ein statisches nur in einer einzigen Ausfertigung (siehe auch ->statische Methoden).

**Statische Methoden** ->Methoden, die zu einer bestimmten ->Klasse gehören, nicht aber zu einzelnen ->Objekten dieser Klasse (siehe auch ->statische Datenelemente).

**Strings** So werden Zeichenketten und auch ->Variablen vom Datentyp string bezeichnet.

**Struktur** Ursprünglich – in Verbindung mit der Programmiersprache C – eine Zusammenfassung verschiedener ->Datentypen zu einem neuen Datentyp. In C++ können auch ->Methoden Mitglied einer Struktur sein. Die Struktur ist daher in C++ ebenfalls eine ->Klasse. Im Gegensatz zu den mit dem Schlüsselwort class definierten Klassen sind die ->Elemente einer mit struct definierten Klasse – also einer Struktur – per Default public.

**Textbausteine** bilden die einzelnen Bestandteile einer ->Anweisung und werden auch als »Token« bezeichnet. Textbausteine können ->Schlüsselwörter, Bezeichner, ->Literele, Operatoren oder Zeichen mit besonderer Bedeutung sein.

**Token** ->Textbausteine.

**Überladen** Das Definieren mehrerer gleichnamiger ->Methoden, die sich in ihrer ->Signatur unterscheiden. Der Zweck ist es, dass eine Methode (oder ein Operator, wenn ein solcher überladen wird; vergleiche hierzu unter ->Operatorfunktionen) scheinbar mit verschiedenen ->Datentypen arbeiten kann und man sich nicht verschiedene Namen ausdenken muss. In Wirklichkeit existieren weiterhin mehrere, datentypabhängige Methoden (oder Operatoren), wenn diese überladen werden, dem Programmierer präsentieren sie sich aber unter einem gemeinsamen Namen.

**Variable** Benannter Ort im Arbeitsspeicher, an dem Werte abgelegt, gelesen und auch verändert werden können.

**Vererbung** Bezeichnet den Sachverhalt, dass eine ->Klasse die ->Elemente einer anderen übernimmt, ohne dass diese in der ersteren explizit definiert werden müssen.

**Zeiger** ->Variablen, die die Adressen anderer Datenobjekte aufnehmen können.

**Zugriffsspezifizierer** regeln den Zugriff auf die einzelnen ->Elemente der ->Klasse und werden in der Definition einer Klasse angegeben.

# Stichwortverzeichnis

## Symbole

! 225  
-- 142  
!= 223  
# 82  
#define 478 f., 481 f.  
#elif 479  
#else 479  
#endif 479 f.  
#if 479  
#ifdef 479, 481  
#ifndef 479, 481  
#include 38, 43, 44, 477  
%= 140  
& (Adresse) 485  
& (Referenz) 521  
&& 225  
\*= 141  
++ 142  
+= 140, 141  
.cpp-Dateien 48  
.exe-Dateien 18, 51, 52  
.NET Framework 583 ff., 589, 590  
.NET *siehe* .NET Framework  
.obj-Dateien 345  
/\* \*/ 67  
// 65  
/= 140  
::-Operator 350, 441  
< 223  
<< (Ausgabeoperator) 40, 85  
<= 223  
-= 140  
== 223  
-> 497, 600  
> 223

>= 223  
>> (Eingabeoperator) 85, 127  
?: 257, 258  
\ 106  
\" 106  
' 106  
\\ 106  
\\0 106  
\\a 106, 107  
\\b 106, 108  
\\f 106, 107  
\\n 93, 104, 106  
\\r 106, 108  
\\t 106, 107  
\\v 106, 107  
^ 600  
{ 84  
|| 225  
} 84

## A

Abgeleitete Klassen 529, 531  
Ableitung 531  
Additionsoperator 122  
Adressen von Arrays 315  
Adressoperator 485  
Aktuelle Parameter 354  
Algorithmus 156, 261, 376  
Allokation 504, 521  
American National Standards Institute 62  
Anchor-Eigenschaft 594 f.  
ANSI C 62  
ANSI/ISO-Standard 24, 36, 62  
Anweisungen 39 ff., 46, 90  
Anweisungsende 46, 77  
app-Konstante 459

Argumente 354  
Arrays 299, 300, 302  
    deklarieren 299  
    dynamisch allozieren 507  
    Elemente 299  
    mehrdimensionale 308 ff.  
ASCII-Code 98, 282, 326  
    erweiterter 100, 101, 207  
    Format 47  
Assoziativität von Operatoren 146  
atof()-Funktion 328, 330  
atoi()-Funktion 328  
Attribute 401, 406  
Aufruf von Funktionen 340, 353 f.  
Aufrufer 340  
Ausdrücke 133 ff.  
    komplexe 135 f.  
Ausführen von C++ Programmen 47, 52, 53  
Ausführungsreihenfolge von Anweisungen 39  
Ausgabeanweisungen 93  
Ausgabeeinheiten 85, 94  
    Aneinanderhängen von 89  
Ausgabeoperatoren 85, 91  
Auskomentieren 70  
Ausnahmebehandlung 569  
Ausnahmen  
    auffangen 571  
    im Aufrufer auffangen 577  
    werfen 570  
Ausrichtung 187  
Ausrichtungslinien 595  
Auswahlrahmen 595  
auto-Speicherklasse 372

## B

BackColor-Eigenschaft 596  
Basic (Programmiersprache) 78  
Basisklasse 529, 531 f., 535 f.  
bcc32.cfg-Datei 34  
bcc32.exe 32, 33  
Bedingte Kompilierung 479  
Bedingte Zuweisung 259  
Bedingungsoperator 257 ff.

Befehle 24, 39  
Befehlszeile 24 f.  
Benutzeroberflächen, grafische 25, 28, 589  
Benutzerschnittstelle 26  
Bereichsauflösungsoperator 350, 441  
Bereichszugriffsoperator 350  
Betriebssystem 18  
Bezeichner 80, 111, 114 f.  
Bezeichnerwahl 114 f.  
Bibliotheken 23, 348  
Binäre Operatoren 136, 224  
Bin-Verzeichnis 32 ff.  
Blitz-Symbol (des Eigenschaftenfensters) 599  
Blöcke 77 ff.  
Bloodshed 24  
bool-Datentyp 191 f.  
Boole, George 191  
Borland C++-Compiler 24, 26, 48  
    Download 29  
    Installation 31 f.  
break-Anweisung 285, 287  
Button-Steuerelement 592, 596, 599, 602

## C

C (Programmiersprache) 60, 78, 399  
C# (Programmiersprache) 78, 583  
C++ (Programmiersprache) 60  
C++/CLI 587, 600  
C++-Editoren 47  
C++-Programme, Aufbau 39  
C++-Standard 62  
c\_str()-Methode 334, 459  
call by reference 499  
call by value 356  
case 247  
case-sensitiv 35  
Casting 220, 222  
catch 571  
catch-Handler 569  
    mehrere verwenden 576  
CD-Befehl (DOS) 50  
cerr-Objekt 463  
character 85  
char-Datentyp 86, 87, 192

CheckBox-Steuerelement 597, 602 f.  
 CheckedChanged-Ereignis 602  
 Checked-Eigenschaft 602 f.  
 cin.get() 319, 321  
 cin-Objekt 85, 127, 128, 456  
 class 410  
 Clear Screen 52  
 Click-Ereignis 599, 602  
 close()-Methode 457  
 CLR *siehe* Common Language Runtime  
 CLS-Befehl (DOS) 52  
 cmath-Headerdatei 349  
 cmd 25  
 Cobol (Programmiersprache) 78  
 Common Language Runtime 583, 586  
 Compiler 20 ff., 27, 29, 39, 42, 44, 48, 345  
 Compiler-Dokumentation 48  
 conio.h-Headerdatei 324, 349  
 const-Konstanten 159 ff.  
 const-Schlüsselwort 160 f., 493, 495  
 continue-Anweisung 287  
 cout.put() 323  
 cout-Objekt 40, 61, 85, 87, 367, 456  
 cstdlib-Headerdatei 260, 329  
 C-String 459  
 ctime-Headerdatei 262, 349

## D

Datei, ausführbare 17, 23, 26  
 Dateierweiterung 18, 20, 26, 48  
 Dateioperationen 455  
 Datenausgabe 85  
 Dateneingabe 85  
 Datenströme 85  
 Datentypen 86, 191 f., 413  
   benutzerdefinierte 399  
   elementare 86  
 Datentypqualifizierer 192, 193  
 Debugger 73  
 Debug-Verzeichnis 460, 466  
 Deklaration  
   von Arrays 299  
   von Funktionen 344  
   von Klassen 399, 410  
   von Referenzen 521

  von Strukturen 399  
   von Variablen 111  
   von Zeigern 487  
 Dekrementoperator 142  
   Postfixnotation 143 f.  
   Präfixnotation 143 f.  
 delete  
   Operator 504, 508  
   Schlüsselwort 504, 508  
 Dereferenzierung von Zeigern 491  
 Dereferenzierungsoperator 491, 497  
 Designer 598  
 Destruktoren 431, 433, 509 f.  
 DJGPP-Compiler 30  
 do while-Schleife 275 f.  
 double-Datentyp 149 ff., 192  
 Dreieckstausch 130 f., 498

## E

E (in Zahlwerten) 153, 169  
 Editor 20, 23, 47  
 Eigenschaften 401, 404  
   Fenster 593  
   von Steuerelementen 593  
 Eigenschaften-Ansicht 593  
 Eigenschaftfenster 592 ff., 598, 599  
 Eingabeaufforderung 24 ff.  
 Eingabeoperator 85  
 Eingabepuffer 319, 321 f.  
 Eingebaute Datentypen 86  
 Elemente  
   von Arrays 299  
   von Klassen 405, 411  
   von Strukturen 399 ff., 404  
   Zugriff über Punktoperator 405, 497  
 else if 241, 244  
 emacs-Editor 48  
 endl-Manipulator 93  
 Endlosschleifen 270, 278  
 Entwicklungsumgebung 47, 73  
   *siehe auch* Integrierte Entwicklungs-  
   umgebung  
 Entwurfsansicht 592, 595  
 Entwurfsfenster 597  
 eof()-Methode 463

Ereignis-Ansicht 593, 599  
 Ereignisbehandlungsroutinen 599  
 Ereignishandler 599  
 Ereignisse 599  
 Escape-Sequenzen 95  
 Eventhandler *siehe* Ereignishandler  
 Exceptions 569  
 exit()-Funktion 573  
 Explizite Typumwandlung 203  
 Exponentialschreibweise 153, 169  
 extern-Schlüsselwort 373

## F

Fakultät 271, 282  
 false 159, 191, 223  
 FCL *siehe* Framework Class Library  
 Fehlerbehandlung 569  
 Fehlermeldung des Compilers 39 f., 52, 58 f.  
 Fehlersuche 70  
 Feldbreite 185 f., 188  
 fixed  
     Ausgabeformat 168  
     Manipulator 177, 178  
 Fließkommawerte 153  
 Fließkommazahlen 153  
 float-Datentyp 149 ff., 192  
 Font-Eigenschaft 594  
 ForeColor-Eigenschaft 596  
 Form *siehe* Formulare  
 formale Parameter 354  
 Formatierung von Ausgaben 165 f.  
     Ausrichtung 187  
     Feldbreite 185  
     Formate 168  
     Füllzeichen 186  
     Genauigkeit 165, 179  
     Standardeinstellungen 165  
 Formulare 589, 592, 597 f.  
 for-Schleife 277, 279  
 Framework Class Library 586  
 fstream-Headerdatei 455  
 Füllzeichen 186  
 Funktionen 40, 44, 78, 337, 339  
     aufrufen 340, 354

definieren 338 f.  
 Deklaration von 344 ff., 349  
 inline-Deklaration 370  
 Rückgabewerte 360 f.  
 Signatur 366  
 Überladen 366  
 vordefinierte 44  
 Vorgabeargumente 364, 367  
 Funktionskopf 41  
 Funktionsprototypen 344 ff.  
 Funktionsrumpf 41

## G

Garbage-Collector 587  
 GCC-Compiler 30, 48, 51  
 gnew-Schlüsselwort 600  
 Genauigkeit 165 f., 180  
 Gesamtausdruck, logischer 225, 227 f.  
 get()-Methode 461  
 getch()-Funktion 324  
 getchar()-Funktion 53  
 getche()-Funktion 325  
 Gleitkommatentypen 153  
 Gleitkommatypen 153  
 Gleitkommawerte 153, 169, 170, 173 f., 193  
     Datentypen 153  
     interne Speicherung 153  
     Literale 152, 154, 201  
 Gleitkommazahlen 153  
 Gleitpunktwerte 153  
 Gleitpunktzahlen 153  
 global resolution-Operator 350  
 Globale Variablen 372  
 Globaler Namensbereich 372  
 Graphical User Interface *siehe* GUI  
 GroupBox-Steuerelement 597  
 Grundtypen 192  
 GUI 25 f.  
     Anwendungen 589  
     Hilfsmittel 27  
     Gültigkeitsbereich  
         Eindeutigkeitsregel 357  
         Funktionen 356  
         von Variablen 288 ff., 373 f.

## H

Hat-Beziehung 437  
 Headerdateien 43 f., 346 f., 349  
     Mehrfacheinbindung verhindern 421  
 Heap 372, 505, 587  
 Height-Attribut 593 f.  
 Hexadezimalsystem 102, 199

## I

IDE *siehe* Integrierte Entwicklungsumgebung  
 if else 239 f.  
 if-Anweisung 231, 233  
 ifstream-Klasse 455 f., 461  
 ilink32.cfg-Datei 34  
 Implizite Typumwandlung 203  
 Include-Dateien 34  
 Initialisierung 124  
     von Arrays 305 f., 309 f.  
     Zeiger 489  
 Inkrementoperator 142  
     Postfixnotation 143 f.  
     Präfixnotation 143 f.  
 inline-Funktionen 370 f.  
 inline-Methoden 418  
 Instanz 431  
 Instanzieren von Klassenobjekten 431  
 int-Datentyp 192  
 Integerdivision 217, 221  
 Integrierte Entwicklungsumgebung 22, 23, 27, 48  
 Intermediate Language *siehe* Microsoft Intermediate Language  
 International Organization for Standardization 62  
 iomanip.h-Headerdatei 173  
 iomanip-Headerdatei 173, 179, 349  
 ios\_base::fixed 174  
 ios\_base::scientific 173  
 ios\_base-Klasse 458  
 ios-Klasse 459  
 iostream.h-Headerdatei 349  
 iostream-Headerdatei 36, 43, 44, 347, 349  
 ISO 62  
 Ist-Beziehung 437

istream-Klasse 456  
 ItemClicked-Ereignis 603  
 itoa()-Funktion 328, 330

## J

Java (Programmiersprache) 78  
 Java Virtual Machine 583, 585  
 JIT Compiler Manager 586  
 JITter 585 f.  
 Just-in-Time-Compiler 585  
 JVM *siehe* Java Virtual Machine

## K

Kapselung 428  
 kbhit()-Funktion 324  
 Kernighan, Brian W. 35, 60  
 Klassen 41, 78, 399, 401, 410, 413 f.  
     abgeleitete 529, 531  
     Basisklassen 529  
     Code verteilen 420  
     definieren 399, 410  
     Destruktoren 431, 433  
     Elemente 405  
     Konstruktoren 431, 433  
     Methoden 413  
     statische Elemente 443  
     vordefinierte 44  
     Zugriffsspezifizierer 424  
 Klassenobjekte 500  
 Kommandozeile 26  
 Kommandozeilen-Compiler 24, 48, 50  
 Kommaoperator 284  
 Kommentare 41  
     auskommentieren 70  
     Einsatz von 70  
     einzeilige 65 f.  
     mehrzeilige 67 ff.  
 Kommentarzeichen 41, 65, 67, 69  
 Kommentarzeile 42  
 Kompilieren von C++ Programmen 47, 49  
 Kompilierung 20, 22  
 Komponentenfach 597  
 Komponentenzugriffsoperator 405  
 Konkatenation 126  
 Konsole 24 ff.

Konsolenprogramme 26  
 Konstanten  
     benannte 159  
     symbolische 159, 162  
     unbenannte 159  
 Konstruktoren 431, 433, 534  
     Standardkonstruktor 434, 436  
     Überladen 431, 434  
 Konstruktormethode 433  
 Kontrollkästchen 597  
 Kontrollstrukturen 39, 192, 223, 267  
     verschachteln 235  
 Konvertierung 204

## L

Label-Steuerelement 596, 602  
 Laufvariable 278  
 Laufzeitfehler 58  
 Laufzeitumgebung 584  
 Lebensdauer von Variablen 290 f., 295  
 Leerräume 80, 82  
 Leerstellen 38  
 Leerzeichen 38, 59, 81  
     als Trennzeichen 75, 80  
     im Quellcode 38  
     innerhalb von Zeichenketten 37  
 left-Manipulator 187 f.  
 Lib-Dateien 34  
 LIFO 372  
 Lineare Listen 500  
 Linker 22 f., 34, 346  
 Linux 21, 25 f., 30, 48  
 Literale 152, 159, 197  
 logische Fehler 57, 58  
 Logische Operatoren 225  
 lokale Variablen 372  
 long double-Datentyp 192 f.  
 long int-Datentyp 194  
 long-Datentyp 192, 194, 196  
 L-Werte 135, 160

## M

Macintosh 30  
 main()-Funktion 41 ff., 363 f.  
 Managed C++ 587

Managed Code 581, 583, 586  
 Manipulatoren 85, 92, 94, 165  
 Mantisse 153  
 Maschinenbefehle 19  
 Maschinencode 19, 43, 583, 586  
 Maschinsprache 19 f., 23  
 math.h-Headerdatei 348 f.  
 Mehrfachvererbung 530  
 MenuStrip-Steuerelement 597 f., 603  
 Methoden 41, 78, 413, 415  
     außerhalb der Klasse definieren 418  
     inline-Definition 418  
     this-Zeiger 502  
     überschreiben 541  
 MFC-Klassen 589 f.  
 Microsoft Intermediate Language 583  
 Microsoft SQL Server Express 590  
 Minuszeichen 136  
 Modularisierung 346  
 Modulo-Operator 138 f.  
 MSDN-Library 590  
 MSIL *siehe* Microsoft Intermediate Language

## N

Namensbereiche 350  
 Namensgleichheit 296 ff.  
 Namenskonflikte 298  
 Namensräume 440  
 namespace 440  
 Negation, logische 225  
 new  
     Operator 504, 507  
     Schlüsselwort 504, 507  
 Newline-Zeichen 82, 93  
 new-Schlüsselwort 600  
 Next Generation Windows Services 585  
 Next()-Methode 600  
 NGWS *siehe* Next Generation Windows Services  
 Nicht objektorientierte Programmiersprachen 78  
 Normalform 153  
 Notepad 38, 47  
 NULL 491, 506



Nullbyte 107, 306, 310, 313, 317  
Nullzeichen 107, 313

## O

Objektdateien 22 f., 345  
Objekte 40, 401, 413 ff.  
    vordefinierte 44  
Objektorientierte Programmiersprachen 78  
Objektorientierte Programmierung 78  
Oder, logisches 225  
Offset 316, 520  
ofstream-Klasse 455 f., 461  
Oktalsystem 199  
open()-Methode 455  
Operatoren 80, 133 ff.  
    & (Adresse) 485  
    & (Referenz) 521  
    arithmetische 137 f.  
    binäre 136, 224, 549  
    Casting 220, 222  
    delete 504, 508  
    Dereferenzierung 491, 497  
    Elementzugriff 405, 497  
    logische 225 f.  
    new 504, 507  
    Priorität 145 ff., 229  
    Sequenzoperator 284  
    überladbare 548  
    überladen 547, 549  
    unäre 136, 560  
Operatormethoden 547 f., 550  
Optionsfelder 597  
ostream-Klasse 456

## P

Panel-Steuerelement 597  
Parameter  
    aktuelle 354  
    formale 354  
    Referenzen 524  
    von Funktionen 351, 353  
    Zeiger 498  
Parameterliste von Funktionen 354, 366  
Pascal (Programmiersprache) 78

Pfeiloperator 600  
Pointer 485  
Popup-Ereignis 603  
Portierbarkeit von Programmen 21, 27, 60  
Postfixnotation von Inkrement- und Dekrementoperatoren 143 f.  
Präfixnotation von Inkrement- und Dekrementoperatoren 143 f.  
Präprozessor 22, 38, 43, 347, 477, 479, 480  
Präprozessor-Direktiven 38, 82, 477  
Priorität  
    von logischen Operatoren 229  
    von Operatoren 145  
    von Vergleichsoperatoren 229  
private-Vererbung 532 f.  
private-Zugriffsspezifizierer 411, 425, 428  
Programm 17, 20  
Programm, ausführbares 17  
Programmdatei 17 ff.  
Programmdatei, ausführbare 20, 22  
Programmierbefehle 39  
Programmierfehler 55, 57  
Programmierstil 83  
Programmierstilregeln 42, 83 f.  
Programmverzeichnis, in Visual Studio 466  
Programmverzweigungen *siehe* Verzweigungen  
Projektmappen-Explorer 592  
Projektverzeichnis, in Visual Studio 466  
Projektvorlagen 590  
Prompt 24 f.  
protected-Vererbung 536, 539  
protected-Zugriffsspezifizierer 532  
Prototypen von Funktionen 344 ff.  
Prozessor 18  
public-Vererbung 540  
public-Zugriffsspezifizierer 411, 425, 431  
Punktoroperator 405, 497  
put()-Methode 461

## Q

Quellcode 20 f., 23, 37, 48  
Quellcodedatei 22, 48  
Quelldatei *siehe* Quellcodedatei

Quelltext *siehe* Quellcode

QuickInfo 597 f., 603

## R

RadioButton-Steuerelement 596, 602 f.

rand()-Funktion 259 f., 262, 387

RAND\_MAX 259

Random (.NET-Klasse) 600

Referenzen 521 f.

als Parameter 524

als Rückgabewerte 526

deklarieren 521

Rekursion 341, 367

resetiosflags()-Manipulator 175, 177

return-Anweisung 42, 286, 361

RichTextBox-Steuerelement 596, 602

right-Manipulator 187 f.

Ritchie, Dennis M. 35, 60

Rückgabewerte

Referenzen 526

von Funktionen 360 f.

von main() 363 f.

Runtime Environment *siehe* Laufzeit-  
umgebung

R-Werte 135, 160

## S

Schaltflächen 596, 599

Schleifen 247, 267, 269

abweisende 268

annehmende 275

Schleifenvariable 278 ff.

Schlüsselwörter 20, 75, 80, 159

Schnittstellen, grafische 25, 28

scientific

Ausgabeformat 168

Manipulator 177 f.

scope-Operator 350

Sequenzoperator 284

setf()-Methode 189

setfill()-Manipulator 186

setiosflags()-Manipulator 173 ff.

setprecision()-Manipulator 179, 184

setw()-Manipulator 185 f.

short int-Datentyp 194

short-Datentyp 192, 194 f.

Signatur, einer Funktion 366, 542

signed 195

Size-Eigenschaft 593 f.

sizeof-Operator 150 f., 307

SnapLines 595

Sonderzeichen 76

source code 20

Sprachstandard 24

srand()-Funktion 261, 388

Stack 372

Standard 26

Standardausgabegerät 40, 85

Standardbibliothek 348 f., 350, 442

Standarddatentypen 86, 192 ff.

Standardeingabe 85

Standardeingabegerät 127

Standardereignisse (von Steuerelementen)  
599, 602

Standardkonstruktor 434, 436

Standardmenü, einrichten 597

Statements 39

static

Schlüsselwort 443, 447

Speicherklasse 372, 374, 444

static-Variablen 294 f.

Statische Attribute einer Klasse 443, 447

Statische Elemente einer Klasse 443

Statische Methoden einer Klasse 446

stdafx.h-Headerdatei 602

stdio.h-Headerdatei 349

stdlib.h-Headerdatei 260, 329

std-Namensraum 44, 350, 442

Steuerelemente 589, 592 f., 595 f., 599, 602

ausrichten 595

auswählen 595

Eigenschaften 593

Farben 596

Größe 593, 595

Name 595

Position 595

Schrift 594

Text 593

Verankerung 594

Steuerelement-Klassen 593

Steuerzeichen 47, 95 ff.  
 strcmp()-Funktion 326  
 strcpy()-Funktion 328  
 Streamklassen 456  
 Streamobjekte 456  
 string-Datentyp 86, 125, 192  
 Strings 86, 125  
     Literale 86  
     Variablen vom Datentyp string 125  
 string-Variablen 331 ff.  
 Stroustrup, Bjarne 60  
 struct 399, 410  
 Strukturansicht von Eigenschaften 594  
 Strukturen 399  
     definieren 399  
     Elemente 399, 401, 404  
 switch-Anweisung 247 ff.  
 Syntaxfehler 55, 57 f., 60  
 Syntaxregeln 39, 75 ff.  
 System.Windows.Forms-Namensraum 589

## T

Tabulatorschritte 59, 81  
 Teilausdrücke, logische 225 f., 228  
 Terminal 25  
 TextAlign-Eigenschaft 594  
 Textbausteine 59, 75 ff.  
 TextBox-Steuerelement 592 ff., 596, 602  
 TextChanged-Ereignis 599, 602  
 Textdateien 455, 461, 466  
 Text-Eigenschaft 593, 595  
 Textfelder 596  
 Textverarbeitungsprogramm 47  
 this-Zeiger 502, 504  
 throw 570  
 time()-Funktion 259  
 time.h-Headerdatei 262, 349  
 Timer-Steuerelement 597  
 Token 75 f.  
 Toolbox 592, 597  
 ToolTip auf ToolTip1-Eigenschaft 598  
 ToolTip-Steuerelement 597 f., 603  
 ToolTipTitle-Eigenschaft 598  
 ToString()-Methode 600

true 159, 191, 223  
 try – catch 569  
 Tutorials 30  
 Typkonvertierung 204  
 Typmodifizierer 161  
 Typumwandlung 203 ff.  
     explizite 220 ff.  
     implizite 203 ff.

## U

Überdecken von Methoden 541  
 Überladen  
     binäre Operatoren 549  
     Konstruktoren 431  
     unäre Operatoren 560  
     von Funktionen 366  
     von Konstruktoren 434  
     von Operatoren 547  
 Überschreiben von Methoden 541  
 Unäre Operatoren 136  
 Und, logisches 225  
 Unix 21  
 unsigned 195  
 using namespace std 36, 44

## V

Variablen 111 ff.  
     Adresse 488  
     deklarieren 111  
     extern-Deklaration 373  
     globale 372  
     Gültigkeitsbereich 288  
     Initialisierung von 124  
     Lebensdauer 290, 295  
     lokale 290, 372  
     statische 295  
 Vererbung 437, 530  
     abgeleitete Klassen 529, 531  
     Basisklassen 529  
     Mehrfachvererbung 530  
     private-Elemente 534  
     Zugriffsspezifizierer 529, 531  
 Vergleichsoperatoren 223  
 Verkettung von Strings 92  
 Verschachteln von Blöcken 80

Verzweigungen 43, 223  
vi (Editor) 48  
Visual Basic (Programmiersprache) 583  
Visual C++ 24, 591  
    IDE 73  
Visual C++ 2005 Express Edition 466, 589,  
    590  
Visual Studio 27, 38, 193, 197, 270, 460, 466,  
    596  
Visual Studio 2005 589 f., 600  
Visual Studio 6 290  
void 43, 353, 361, 363  
Vorgabeargumente von Funktionen 364,  
    367  
Vorgabeformat 168

## W

Wahrheitswerte 191  
Warnungen des Compilers 152  
Wertübergabe 356  
while-Schleife 267 ff.  
Width-Attribut 593, 595  
Wiederholungsanweisungen 267  
Window-Manager 27  
Windows Forms 589  
Windows Server 2003 590  
Windows Vista 590  
Windows XP 590  
Windows-Anwendungen 589 f., 592, 596  
Windows-Betriebssystem 21, 25, 29, 48,  
    583, 586  
Windows-Editor 47  
Windows-Explorer 25, 53

Windows-Forms-Designer 593  
Windows-Programme 26, 27  
Windows-Programmierung 27

## Z

Zählschleife 277  
Zeichenketten 82, 86 f., 91, 107, 313 ff.  
    Aneinanderhängen von 92  
    Literele 86  
Zeiger 359, 485  
    Adressen zuweisen 489  
    als Attribut 500  
    als Parameter 498  
    auf Klassenelemente 495  
    Datentyp 488  
    deklarieren 487  
    dereferenzieren 491  
    initialisieren 489  
    konstante 493  
    NULL 491  
    this 502  
Zeigervariablen 359, 485  
Zeilenumbrüche 59  
Zufallsgenerator 261 f., 388  
Zufallszahlen 259, 261 f., 387  
Zugriffsspezifizierer  
    bei der Vererbung 529, 531  
    in der Klassendefinition 424  
Zuweisungen 116, 118 f.  
    zusammengesetzte 140  
Zuweisungsoperator 117  
Zwischenraumzeichen 37, 39, 59, 80 ff., 92