



dirk LOUIS



2. Auflage

C++

Das komplette Starterkit
für den einfachen Einstieg
in die Programmierung

HANSER



Inklusive Einführung in Visual Studio
(Windows) und den GNU-Compiler (Linux)

Bleiben Sie auf dem Laufenden!



Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter



www.hanser-fachbuch.de/newsletter



Hanser Update ist der IT-Blog des Hanser Verlags mit Beiträgen und Praxistipps von unseren Autoren rund um die Themen Online Marketing, Webentwicklung, Programmierung, Softwareentwicklung sowie IT- und Projektmanagement. Lesen Sie mit und abonnieren Sie unsere News unter



www.hanser-fachbuch.de/update



Dirk Louis

C++

Das komplette Starterkit
für den einfachen Einstieg
in die Programmierung

2. Auflage

HANSER

Der Autor:

Dirk Louis, Saarbrücken, autoren@carpelibrum.de

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

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

Bibliografische Information der Deutschen Nationalbibliothek:

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

Dieses Werk ist urheberrechtlich geschützt.

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

© 2018 Carl Hanser Verlag München, www.hanser-fachbuch.de

Lektorat: Brigitte Bauer-Schierek

Copy editing: Petra Kienle, Fürstenfeldbruck

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

Umschlagrealisation: Stephan Rönigk

Layout: Kösel Media GmbH, Krugzell

Druck und Bindung: Hubert & Co. GmbH & Co. KG BuchPartner, Göttingen

Printed in Germany

Print-ISBN: 978-3-446-44597-0

E-Book-ISBN: 978-3-446-45388-3

Inhalt

Vorwort	XXIII
Teil I: Grundkurs	1
1 Keine Angst vor C++!	3
1.1 Von C zu C++	4
1.1.1 Rückblick	4
1.1.2 Die strukturierte Programmierung	6
1.1.3 Chips sind billig, Programmierer teuer	8
1.1.4 Fassen wir zusammen	9
1.2 Von der Idee zum fertigen Programm	10
1.3 Näher hingeschaut: der C++-Compiler	12
1.3.1 Der Compiler ist ein strenger Lehrer	12
1.3.2 Definition und Deklaration	13
1.3.3 Das Konzept der Headerdateien	15
1.3.4 Namensräume	16
1.3.5 Der Compiler bei der Arbeit	18
1.3.6 ISO und die Compiler-Wahl	19
1.3.7 Der neue C++17-Standard	19
1.4 Übungen	20
2 Grundkurs: Das erste Programm	21
2.1 Hallo Welt! – das Programmgerüst	21
2.1.1 Typischer Programmaufbau	22
2.1.2 Die Eintrittsfunktion main()	23
2.1.3 Die Anweisungen	24
2.1.4 Headerdateien	26
2.1.5 Kommentare	27
2.2 Programmerstellung	28
2.2.1 Programmerstellung mit Visual Studio	28
2.2.2 Programmerstellung mit GNU-Compiler	35
2.2.3 Programmausführung	36

2.3	Stil	38
2.4	Übungen	39
3	Grundkurs: Daten und Variablen	41
3.1	Konstanten (Literale)	41
3.2	Variablen	44
3.2.1	Variablendefinition	44
3.2.2	Werte in Variablen speichern	47
3.2.3	Variablen bei der Definition initialisieren	48
3.2.4	Werte von Variablen abfragen	49
3.3	Konstante Variablen	50
3.4	Die Datentypen	51
3.4.1	Die Bedeutung des Datentyps	51
3.4.2	Die elementaren Datentypen	55
3.4.3	Weitere Datentypen	57
3.5	Typumwandlung	57
3.5.1	Typumwandlung bei der Ein- und Ausgabe	57
3.5.2	Automatische Typumwandlungen	60
3.5.3	Explizite Typumwandlungen	61
3.6	Übungen	62
4	Grundkurs: Operatoren und Ausdrücke	65
4.1	Rechenoperationen	65
4.1.1	Die arithmetischen Operatoren	65
4.1.2	Die mathematischen Funktionen	68
4.2	Ausdrücke	69
4.3	Die kombinierten Zuweisungen	71
4.4	Inkrement und Dekrement	71
4.5	Strings addieren	73
4.6	Weitere Operatoren	74
4.7	Übungen	74
5	Grundkurs: Kontrollstrukturen	75
5.1	Entscheidungen und Bedingungen	75
5.1.1	Bedingungen	76
5.1.2	Die Vergleichsoperatoren	77
5.1.3	Die logischen Operatoren	78
5.2	Verzweigungen	80
5.2.1	Die einfache if-Anweisung	80
5.2.2	Die if-else-Verzweigung	82
5.2.3	Die switch-Verzweigung	85

5.3	Schleifen	89
5.3.1	Die while-Schleife	89
5.3.2	Die do-while-Schleife	93
5.3.3	Die for-Schleife	95
5.3.4	Schleifen mit mehreren Schleifenvariablen	96
5.3.5	Performance-Tipps	97
5.4	Sprunganweisungen	97
5.4.1	Abbruchbefehle für Schleife	99
5.4.2	Abbruchbefehle für Funktionen	102
5.4.3	Sprünge mit goto	102
5.5	Fallstricke	102
5.5.1	Die leere Anweisung ;	102
5.5.2	Nebeneffekte in booleschen Ausdrücken	103
5.5.3	Dangling else-Problem	104
5.5.4	Endlosschleifen	105
5.6	Übungen	106
6	Grundkurs: Eigene Funktionen	109
6.1	Definition und Aufruf	110
6.1.1	Der Ort der Funktionsdefinition	111
6.1.2	Funktionsprototypen (Deklaration)	112
6.2	Rückgabewerte und Parameter	113
6.2.1	Rückgabewerte	115
6.2.2	Parameter	117
6.3	Lokale und globale Variablen	122
6.3.1	Lokale Variablen	122
6.3.2	Globale Variablen	123
6.3.3	Gültigkeitsbereiche und Verdeckung	124
6.4	Funktionen und der Stack	126
6.5	Überladung	128
6.6	Übungen	130
7	Grundkurs: Eigene Datentypen	131
7.1	Arrays	131
7.1.1	Definition	131
7.1.2	Auf Array-Elemente zugreifen	133
7.1.3	Initialisierung	133
7.1.4	Arrays in Schleifen durchlaufen	134
7.1.5	Arrays an Funktionen übergeben	137
7.1.6	Mehrdimensionale Arrays	137
7.1.7	Vor- und Nachteile der Programmierung mit Arrays	138

7.2	Aufzählungen	138
7.2.1	Definition	141
7.2.2	Variablen	141
7.2.3	Aufzählungstypen und switch-Verzweigungen	142
7.2.4	Die neuen enum class-Aufzählungen	142
7.3	Strukturen	143
7.3.1	Definition	144
7.3.2	Variablendefinition	145
7.3.3	Zugriff auf Elemente	146
7.3.4	Initialisierung	146
7.3.5	Arrays von Strukturen	146
7.4	Klassen	148
7.4.1	Definition	148
7.4.2	Variablen, Objekte und Konstruktoren	148
7.4.3	Zugriffsschutz	149
7.5	Übungen	152
8	Grundkurs: Zeiger und Referenzen	153
8.1	Zeiger	153
8.1.1	Definition	154
8.1.2	Initialisierung	154
8.1.3	Dereferenzierung	156
8.1.4	Zeigerarithmetik	158
8.2	Referenzen	159
8.3	Einsatzgebiete	159
8.3.1	call by reference	160
8.3.2	Dynamische Speicherreservierung	165
8.4	Übungen	171
9	Grundkurs: Noch ein paar Tipps	173
9.1	Wie gehe ich neue Programme an?	173
9.2	Wo finde ich Hilfe?	174
9.2.1	Hilfe zu Compiler-Meldungen	174
9.2.2	Hilfe bei der Lösung von Programmieraufgaben	175
9.2.3	Hilfe bei Programmen, die nicht richtig funktionieren	179
9.2.4	Debuggen	179
9.3	Programme optimieren	181

Teil II – Aufbaukurs: die Standardbibliothek	183
10 Aufbaukurs: Einführung	185
10.1 Bibliotheken verwenden	185
10.2 Hilfe zu den Bibliothekselementen	186
11 Aufbaukurs: Mathematische Funktionen	189
11.1 Die mathematischen Funktionen	189
11.1.1 Mathematische Konstanten	191
11.1.2 Verwendung der trigonometrischen Funktionen	192
11.1.3 Überläufe	192
11.2 Zufallszahlen	193
11.3 Komplexe Zahlen	195
11.4 Übungen	196
12 Aufbaukurs: Strings	197
12.1 String-Literale	197
12.1.1 Escape-Sequenzen	198
12.1.2 Zeilenumbrüche	200
12.2 Strings erzeugen	201
12.3 Strings aneinanderhängen	202
12.4 Strings vergleichen	202
12.5 Sonstige String-Manipulationen	205
12.6 C-Strings	206
12.7 Umwandlungen zwischen Strings und Zahlen	207
12.8 Übungen	208
13 Aufbaukurs: Ein- und Ausgabe	209
13.1 Daten auf die Konsole ausgeben	209
13.2 Formatierte Ausgabe	210
13.2.1 Ausgabebreite	210
13.2.2 Füllzeichen	211
13.2.3 Genauigkeit	211
13.2.4 Formatierte Ausgabe mit printf()	212
13.3 Deutsche Umlaute	213
13.4 Daten über die Konsole (Tastatur) einlesen	216
13.5 Fehlerbehandlung	217
13.6 Streams	219
13.7 Textdateien	220
13.7.1 In Textdateien schreiben	221
13.7.2 Aus Textdateien lesen	223

13.8	Binärdateien	226
13.9	Übungen	228
14	Aufbaukurs: Zeit und Datum	229
14.1	Zeit und Datum	229
14.2	Laufzeitmessungen	235
14.3	Übungen	237
15	Aufbaukurs: Container	239
15.1	Die STL	239
15.2	vector – ein intelligenter Daten-Container	242
15.2.1	Einsatz eines Containers	243
15.2.2	Größenmanagement von Containern	244
15.2.3	Typische Memberfunktionen	245
15.3	Der Gebrauch von Iteratoren	246
15.4	Die Algorithmen	249
15.4.1	generate()	252
15.4.2	stable_sort()	253
15.5	Schlüssel/Wert-Paare	254
15.6	Übungen	256
16	Aufbaukurs: Programme aus mehreren Quelltextdateien	257
16.1	Quelltext verteilen	257
16.1.1	Funktionen über Dateigrenzen hinweg verwenden	258
16.1.2	Klassen über Dateigrenzen hinweg verwenden	258
16.1.3	Variablen über Dateigrenzen hinweg verwenden	259
16.1.4	Typdefinitionen über Dateigrenzen hinweg verwenden	260
16.2	Mehrfacheinkopieren von Headerdateien verhindern	261
16.3	Übungen	263
Teil III – Objektorientierte Programmierung	265	
17	OOP-Kurs: Klassen	267
17.1	Objektorientiert denken – objektorientiert programmieren	267
17.1.1	Objektorientiertes Programmieren	267
17.1.2	Wie sind Objekte beschaffen?	268
17.1.3	Wie findet man einen objektorientierten Lösungsansatz?	270
17.1.4	Objekte und Klassen	271
17.2	Klassendefinition	274
17.2.1	Zugriffsrechte	275

17.2.2 Quelltext- und Headerdatei	277
17.2.3 Klassen zu Visual-Studio-Projekten hinzufügen	280
17.3 Membervariablen	283
17.3.1 Anfangswerte	284
17.3.2 Private-Deklaration	288
17.3.3 Eingebettete Objekte	290
17.3.4 Konstante Membervariablen	292
17.3.5 Statische Membervariablen	293
17.4 Memberfunktionen	294
17.4.1 Definition innerhalb der Klassendefinition	294
17.4.2 Definition außerhalb der Klassendefinition	295
17.4.3 Der this-Zeiger	296
17.4.4 Statische Memberfunktionen	297
17.4.5 Konstante Memberfunktionen	298
17.4.6 Get-/Set-Memberfunktionen	299
17.5 Die Konuktoren	302
17.5.1 Definition und Aufruf	302
17.5.2 Ersatz- und Standardkonuktoren	304
17.6 Der Destruktor	307
17.7 Übungen	308
18 OOP-Kurs: Vererbung	311
18.1 Das Prinzip der Vererbung	311
18.1.1 Der grundlegende Mechanismus	312
18.1.2 Die Syntax	313
18.1.3 Wann ist Vererbung gerechtfertigt?	314
18.1.4 Einige wichtige Fakten	315
18.2 Das Basisklassenunterobjekt	316
18.2.1 Zugriff	317
18.2.2 Instanzbildung	320
18.3 Die Zugriffsspezifizierer für die Vererbung	322
18.4 Verdecken, überschreiben und überladen	323
18.4.1 Verdeckung	324
18.4.2 Überladung	324
18.4.3 Überschreibung	325
18.5 Der Destruktor	325
18.6 Mehrfachvererbung	326
18.7 Übungen	326
19 OOP-Kurs: Polymorphie	329
19.1 Grundprinzip und Implementierung	330
19.2 Späte und frühe Bindung	333

19.2.1	Frühe Bindung	333
19.2.2	Späte Bindung	334
19.3	Generische Programmierung	335
19.3.1	Basisklassen-Arrays	336
19.3.2	Basisklassenparameter	338
19.4	Typidentifizierung zur Laufzeit (RTTI)	339
19.4.1	Umwandlung mit <code>dynamic_cast</code>	339
19.4.2	Der <code>typeid()</code> -Operator	341
19.5	Abstrakte Klassen	341
19.5.1	Rein virtuelle Funktionen	342
19.5.2	Abstrakte Klassen	342
19.6	Übungen	343
20	OOP-Kurs: Ausnahmebehandlung	345
20.1	Fehlerprüfung mit Ausnahmen	346
20.2	Ausnahmen abfangen	348
20.3	Ausnahmen auslösen	351
20.4	Programmfluss und Ausnahmebehandlung	353
20.4.1	Wo wird der Programmfluss nach einer Ausnahme fortgesetzt? ..	354
20.4.2	Die Problematik des gestörten Programmflusses	354
20.5	Übungen	356
Teil IV – Profikurs	357
21	Profikurs: Allgemeine Techniken	359
21.1	Vorzeichen und Überlauf	359
21.2	Arithmetische Konvertierungen	361
21.3	Lokale <code>static</code> -Variablen	361
21.4	Der <code>?:-</code> -Operator	362
21.5	Bit-Operatoren	362
21.5.1	Multiplikation mit 2	363
21.5.2	Division durch 2	364
21.5.3	Klein- und Großschreibung	364
21.5.4	Flags umschalten	365
21.5.5	Gerade Zahlen erkennen	365
21.6	Zeiger auf Funktionen	367
21.7	Rekursion	369
21.8	<code>constexpr</code> -Funktionen	371
21.9	Variablendefinition in <code>if</code> und <code>switch</code>	372

22 Profikurs: Objektorientierte Techniken	375
22.1 Zeiger auf Memberfunktionen	375
22.2 Friends	377
22.3 Überladung von Operatoren	378
22.3.1 Syntax	378
22.3.2 Überladung des Inkrement-Operators <code>++</code>	379
22.3.3 Überladung arithmetischer Operatoren <code>+</code> , <code>+=</code>	380
22.3.4 Überladung der Streamoperatoren <code><>></code>	381
22.4 Objekte vergleichen	382
22.4.1 Gleichheit	382
22.4.2 Größenvergleiche	384
22.5 Objekte kopieren	386
23 Profikurs: Gültigkeitsbereiche und Lebensdauer	391
24 Profikurs: Templates	395
24.1 Funktionen-Templates	396
24.2 Klassen-Templates	397
25 Profikurs: Reguläre Ausdrücke	401
25.1 Syntax regulärer Ausdrücke	401
25.1.1 Zeichen und Zeichenklassen	402
25.1.2 Quantifizierer	403
25.1.3 Gruppierung	404
25.1.4 Assertionen (Anker)	405
25.2 Musterabgleich mit regulären Ausdrücken	405
25.3 Suchen mit regulären Ausdrücken	406
25.4 Ersetzen mit regulären Ausdrücken	407
26 Profikurs: Lambda-Ausdrücke	409
26.1 Syntax	409
26.2 Einsatz	411
A Anhang A: Lösungen	413
B Anhang B: Die Beispiele zum Buch	433
B.1 Installation der Visual Studio Community Edition	433
B.2 Ausführung der Beispielprogramme	436
B.2.1 Ausführung mit VS Community Edition 2017	437

B.2.2	Ausführung mit beliebigen integrierten Entwicklungsumgebungen	438
B.2.3	Ausführung mit GNU-Konsolen-Compiler	439
C	Anhang C: Zeichensätze	441
C.1	Der ASCII-Zeichensatz	441
C.2	Der ANSI-Zeichensatz	442
D	Anhang D: Syntaxreferenz	445
D.1	Schlüsselwörter	445
D.2	Elementare Typen	446
D.3	Strings	447
D.4	Operatoren	448
D.5	Ablaufsteuerung	449
D.6	Ausnahmebehandlung	451
D.7	Aufzählungen	451
D.7.1	enum	451
D.7.2	enum class (C++11)	452
D.8	Arrays	452
D.9	Zeiger	453
D.10	Strukturen	453
D.11	Klassen	454
D.12	Vererbung	457
D.13	Namensräume	457
E	Anhang E: Die Standardbibliothek	459
E.1	Die C-Standardbibliothek	459
E.2	Die C++-Standardbibliothek	460
Index	463

Vorwort

Sie besitzen einen Computer und wissen nicht, wie man programmiert?

Das ist ja furchtbar! Jetzt erzählen Sie mir nur nicht, dass Sie Ihren Computer nur zum Spielen und Internetsurfen benutzen. Dann wäre ich wirklich enttäuscht.

Ach so, jemand hat Ihnen erzählt, dass Programmieren sehr kompliziert sei und viel mit Mathematik zu tun hätte.

Tja, dann wollte sich dieser jemand wohl ein wenig wichtigmachen oder hat selbst nichts vom Programmieren verstanden, denn Programmieren ist nicht schwieriger als Kochen oder das Erlernen einer Fremdsprache. Und mehr mathematisches Verständnis als es von einem Schüler der sechsten oder siebten Klasse verlangt wird, ist definitiv auch nicht nötig.

Reizen würde Sie das Programmieren schon, aber Sie wissen ja gar nicht so recht, was Sie programmieren könnten.

Keine Angst, sowie Sie mit dem Programmieren anfangen, werden Ihnen zahlreiche Ideen kommen. Und weitere Anregungen finden sich in guten Lehrbüchern zuhause beziehungsweise ergeben sich beim Austausch mit anderen Programmierern.

Immer noch Zweifel? Sie würden sich das Programmieren am liebsten im Selbststudium beibringen? Sie suchen ein Buch, das Sie nicht überfordert, mit dem Sie aber dennoch richtig professionell programmieren lernen?

Aha, Sie sind der unsicher-anspruchsvolle Typ! Dann dürfte das vorliegende Buch genau richtig für Sie sein. Es ist nicht wie andere Bücher primär thematisch gegliedert, sondern in Stufen – genauer gesagt vier Stufen, die Sie Schritt für Schritt auf ein immer höheres Niveau heben.

Aufbau des Buchs

Die **erste Stufe** ist der Grundkurs. Hier schreiben Sie Ihre ersten Programme und lernen die Grundzüge der Programmierung kennen. Danach besitzen Sie fundiertes Basiswissen und können eigenständig Ihre ersten Programmideen verwirklichen.

Bestimmte Aufgaben kehren bei der Programmierung immer wieder: beispielsweise das Abfragen des Datums, die Berechnung trigonometrischer Funktionen, die Verwaltung größerer Datenmengen oder das Schreiben und Lesen von Dateien. Für diese Aufgaben gibt es in der C++-Standardbibliothek vordefinierte Elemente. Wie Sie diese nutzen, erfahren Sie im Aufbaukurs – der **zweiten Stufe**.

Die **dritte Stufe** stellt Ihnen die objektorientierte Programmierung vor und lehrt Sie, wie Sie den Code immer größerer Programme sinnvoll organisieren.

Die **vierte Stufe** schließlich stellt Ihnen noch einige letzte C++-Techniken vor, die Sie vermutlich eher selten einsetzen werden, die ein professioneller C++-Programmierer aber kennen sollte.

Abgerundet wird das Buch durch einen umfangreichen **Anhang**, der unter anderem eine C++-Syntaxübersicht und eine Kurzreferenz der Standardbibliothek beinhaltet.

Nicht verzagen!

Natürlich gibt es auch Zeiten des Verdrusses und des Frusts. Oh ja, die gibt es! Aber seien wir ehrlich: Wäre der Weg nicht so steinig, wäre die Freude am Ziel auch nicht so groß. Was sind das denn für trostlose Gesellen, die in ihrer gesamten Zeit als Programmierer noch keine Nacht durchwacht haben, weil sie den Fehler, der das Programm immer zum Abstürzen bringt, nicht finden konnten? Und was soll man von einem Programmierer halten, der noch nie aus Versehen ein Semikolon hinter eine if-Bedingung gesetzt hat? (Und dem die Schamesröte ins Gesicht schoss, als er einen vorbeikommenden Kollegen um Hilfe bat und ihn dieser nach einem flüchtigen Blick auf den Quellcode auf den Fehler aufmerksam machte.) Sind das überhaupt echte Programmierer?

Wer programmieren lernen will, der muss auch erkennen, dass bei der Programmierung nicht immer alles glatt geht. Das ist nicht ehrenrührig, man darf sich nur nicht unterkriegen lassen. Sollten Sie also irgendwo auf Schwierigkeiten stoßen – sei es, dass Sie etwas nicht ganz verstanden haben oder ein Programm nicht zum Laufen bekommen –, versuchen Sie sich nicht zu sehr in das Problem zu verbohren. Legen Sie eine kleine Pause ein oder lesen Sie erst einmal ein wenig weiter – oftmals klärt sich das Problem danach von selbst.

Dirk Louis

Saarbrücken, im Frühjahr 2018

Teil I: Grundkurs

In diesem Teil erlernen Sie die Grundlagen von C++ und der Programmierung im Allgemeinen.

Das bedeutet, dass wir eine Menge Stoff zu bewältigen haben. Doch wir werden es gelassen angehen. Zuerst eignen wir uns ein wenig Theorie an, anschließend betrachten wir die technische Seite der Programmerstellung und ab Kapitel 3 tauchen wir dann so richtig in die C++-Programmierung ein.

C++ steht in dem Ruf, eine besonders mächtige und leistungsfähige, aber leider auch eine sehr schwer zu erlernende Programmiersprache zu sein. Letzteres ist wohl darauf zurückzuführen, dass die vielfältigen Möglichkeiten und die Freiheiten, die C++ dem Programmierer bietet, einer ebenso großen Zahl an Konzepten, Techniken und unterschiedlichen Syntaxformen gegenüberstehen. Und gerade diese Syntaxformen – das lässt sich nicht leugnen – können auf Anfänger schon recht abschreckend wirken. Einige Kostproben gefällig? Zeilen der Form:

```
virtual const char* f() const noexcept;
```

sind in C++ keineswegs unüblich und auch Berechnungen der Form:

```
i = m++*+n;
```

sind möglich. Besondere Freude bereiten aber Deklarationen wie z. B.:

```
int *(*f(int))(int, int);
```

Falls Ihnen jetzt Zweifel kommen, ob Sie mit C++ wirklich die richtige Wahl getroffen haben, so lassen Sie sich versichern:

- C++ ist viel einfacher, als es manchmal den Anschein hat.
- Schon bald werden Sie Ihre eigenen C++-Programme schreiben.
- Mit jedem Programm, das Sie schreiben, wird Ihnen C++ vertrauter und selbstverständlicher erscheinen.

Am Ende dieses Buchs werden Sie nicht nur in der Lage sein, attraktive und professionelle Programmvorhaben anzugehen, Sie werden auch verstehen, was die erste der obigen Beispielzeilen bedeutet, und Sie werden den Kopf darüber schütteln, warum der Verfasser der eigentlich doch ganz einfachen zweiten Beispielzeile den Code nicht lesefreundlicher formatiert hat.

Nur die Bedeutung der dritten Beispielzeile werden Sie nach der Lektüre dieses Buchs immer noch nicht verstehen. Aber trösten Sie sich: Die Konstruktion, die in dieser Zeile deklariert wird¹, ist so abgehoben, dass Sie unter zehn professionellen C++-Programmier-

¹ eine Funktion, die einen Funktionszeiger als Rückgabetyp besitzt

rern vermutlich höchstens einen finden werden, der diese Konstruktion erkennt, geschweige denn sie selbst schon einmal eingesetzt hätte.

Warum aber ist C++ so mächtig? Warum gibt es so viele Konzepte in der Sprache und warum ist die Syntax so kryptisch? Die Antwort auf diese Fragen liegt in der Geschichte von C++.

■ 1.1 Von C zu C++

„The times they are a changin“ – die Zeiten ändern sich – heißt es in einem berühmten Song von Bob Dylan. Sicherlich hatte Dylan dabei nicht die Entwicklungen in der IT-Branche und der Softwareerstellung im Auge, doch allgemeine Wahrheiten lassen sich eben auf viele verschiedene Bereiche anwenden – und manchmal eben auch auf den Bereich der Softwareentwicklung.

Dort hat im Laufe der letzten Jahrzehnte tatsächlich ein grundlegender Wandel stattgefunden.

1.1.1 Rückblick

Wann die Entwicklung des Computers, der Rechenmaschine, begonnen hat, ist gar nicht so leicht zu sagen. Es hängt sehr davon ab, wie weit man zurückgehen möchte. In einer Aprilausgabe der renommierten Fachzeitschrift „Spektrum der Wissenschaften“ wurde vor einigen Jahren beispielsweise von einer Aufsehen erregenden Entdeckung berichtet: Amerikanische Archäologen hatten auf einer Insel bei Neuguinea einen frühzeitlichen Computer entdeckt. Aus Seilen und Rollen hatten die Ureinwohner aus der Elektronik bekannte Schaltbausteine wie AND-Gatter, OR-Gatter und Inverter erstellt und zu einem echten Rechenwerk zusammengesetzt. Die Archäologen nahmen an, dass die damalige Priesterkaste diesen „Computer“ als eine Art Orakel betrieb und ihr Wissen um die Konstruktion dieses Orakels zum Machterhalt nutzte. Schematische Abbildungen zur Funktionsweise des Rechenwerks und eine Einführung in die digitale Schaltungslogik rundeten den Artikel ab. Natürlich handelte es sich um einen Aprilscherz, aber immerhin: Unter dem Eindruck von so viel Logik und Wissenschaft blieb der gesunde Menschenverstand einiger Leser auf der Strecke. In den nachfolgenden Monaten ergingen daraufhin einige böse Briefe an die Redaktion von aufgebrachten Lesern, die die sensationelle Nachricht sofort weiterverbreitet und sich dabei bei ihren Professoren und Kollegen blamiert hatten.

Nicht ganz so weit zurückliegend, dafür aber verbrieft, ist die Erfindung des Lochkartensystems durch den Deutsch-Amerikaner Hermann Hollerith. Um das Jahr 1890 entwickelte er ein Verfahren, bei dem Daten durch Lochung bestimmter Felder auf vorgefertigten Karten (eben den Lochkarten) kodiert und festgehalten wurden. Mit Hilfe spezieller Maschinen, den sogenannten Hollerith- oder Lochkartenmaschinen, konnte man diese Daten automatisch auswerten, beispielsweise zur Erstellung von Serienbriefen, zur statistischen Datenerfassung oder allgemein zur Auswertung großer Datenmengen.

Der erste offiziell anerkannte, noch mechanische Computer war der 1936 gebaute Z1 des Berliners Konrad Zuse. Kurz darauf folgten Röhren-, später Transistoren- und schließlich Chip-Rechner. In der Zwischenzeit hatte sich auch bei der Softwareentwicklung Einiges getan: Anstatt Lochkarten zu stanzen, gab man Maschinenbefehlprogramme über ein Terminal ein. Irgendwann wurden die Maschinenbefehle durch die Sprache Assembler ersetzt und schließlich kamen die ersten höheren Programmiersprachen, die interpretiert oder kompiliert wurden.



Interpreter und Compiler

Maschinenbefehle sind „Wörter“, die aus einer Folge von Nullen und Einsen bestehen, also beispielsweise 0011000010101011. Das eigentliche Rechenwerk eines Computers, der *Prozessor*, versteht nur diese binären Befehle (wobei noch zu beachten ist, dass jeder Prozessortyp seinen eigenen spezifischen Sprachschatz hat). Da das Programmieren mit diesen Befehlen für Menschen viel zu mühsam und schwierig ist, kam man auf die Idee, die Programmquelltexte in einer anderen Sprache aufzusetzen und dann mit Hilfe eines passenden Übersetzerprogramms in Maschinenbefehle umschreiben zu lassen. Die üblichen menschlichen Sprachen sind aber viel zu komplex und uneindeutig, um sie maschinell übersetzen zu können. Aus diesem Grunde wurden eigene Programmiersprachen wie C oder Basic entwickelt, mit einfacher Grammatik und geringem Wortschatz, die für Menschen leichter zu erlernen und für Übersetzerprogramme leichter in Maschinenbefehle umzusetzen sind.

Grundsätzlich gibt es zwei Kategorien von Übersetzerprogrammen: die Interpreter und die Compiler.

Ein Interpreter lädt den Quelltext des Programms, übersetzt ihn stückweise und lässt die übersetzten Anweisungen direkt vom Prozessor ausführen. Endet das Programm, endet auch die Ausführung des Interpreters.

Ein Compiler lädt den Quelltext des Programms, übersetzt ihn komplett (wobei er auch kleinere Optimierungen vornehmen kann) und speichert das kompilierte Programm in einer neuen Datei (unter Windows eine .exe-Datei) auf der Festplatte. Wenn der Anwender die .exe-Datei danach aufruft, wird das Programm direkt vom Betriebssystem geladen und ausgeführt.

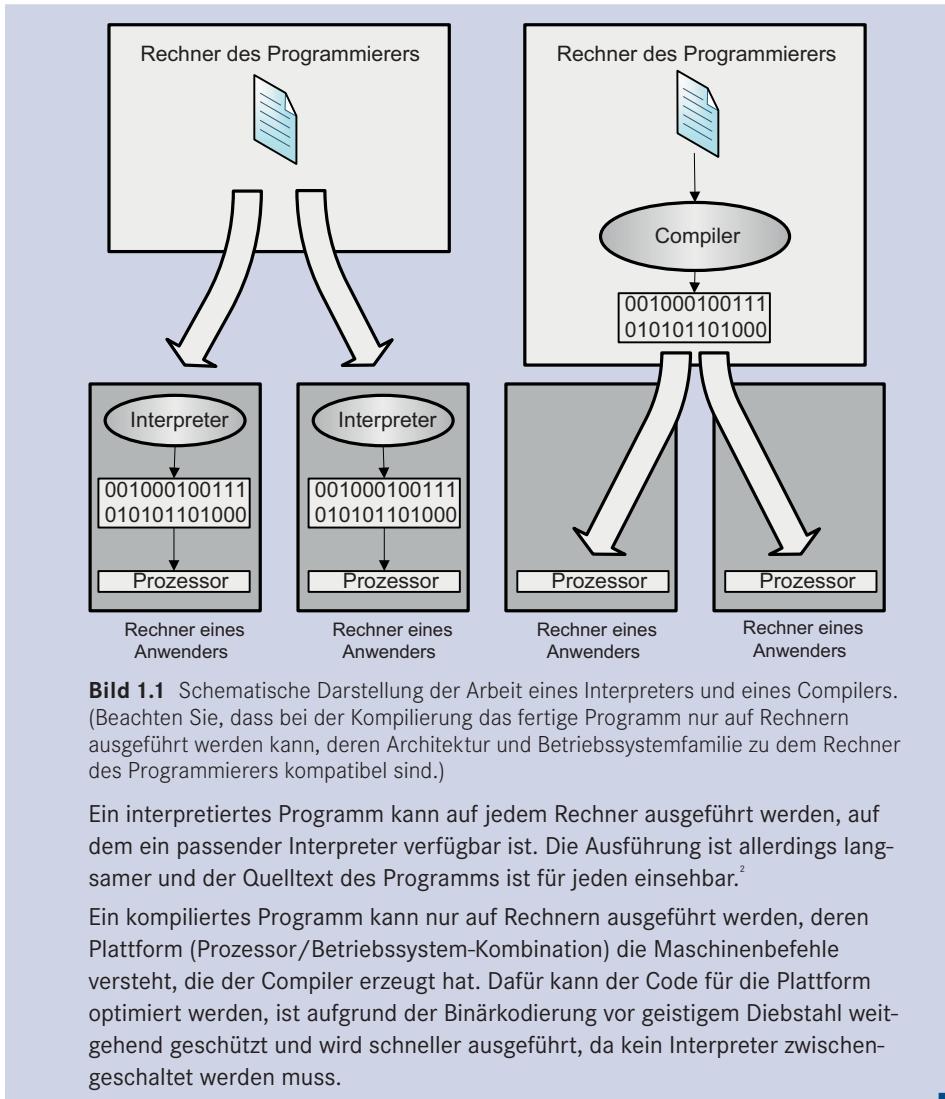


Bild 1.1 Schematische Darstellung der Arbeit eines Interpreters und eines Compilers. (Beachten Sie, dass bei der Kompilierung das fertige Programm nur auf Rechnern ausgeführt werden kann, deren Architektur und Betriebssystemfamilie zu dem Rechner des Programmierers kompatibel sind.)

Ein interpretiertes Programm kann auf jedem Rechner ausgeführt werden, auf dem ein passender Interpreter verfügbar ist. Die Ausführung ist allerdings langsamer und der Quelltext des Programms ist für jeden einsehbar.²

Ein kompiliertes Programm kann nur auf Rechnern ausgeführt werden, deren Plattform (Prozessor/Betriebssystem-Kombination) die Maschinenbefehle versteht, die der Compiler erzeugt hat. Dafür kann der Code für die Plattform optimiert werden, ist aufgrund der Binärkodierung vor geistigem Diebstahl weitgehend geschützt und wird schneller ausgeführt, da kein Interpreter zwischengeschaltet werden muss.

1.1.2 Die strukturierte Programmierung

Anfangs erlaubten die Programmiersprachen nur Programme, in denen die einzelnen Anweisungen, die der Computer bei Ausführung des Programms abarbeiten sollte, von oben nach unten im Programmquelltext angegeben werden mussten.

² Moderne „Interpreter“, wie sie beispielsweise für die Ausführung von Java-, C#- oder .NET-C++-Programmen verwendet werden, interpretieren daher vorkompilierte Programme und nennen sich JIT-Compiler (= „Just in time“-Compiler), weil sie die Programmausführung kaum verzögern.

1. Tue dies
2. Tue das
3. Mache jetzt jenes
4. ...

Programmieranfänger dürfte dieses Prinzip begeistern, denn es ist leicht zu verstehen. Doch in der Praxis stellte sich damals bald heraus, dass diese Art der Programmierung ihre Grenzen hat: Einmal implementierte Teillösungen lassen sich schlecht wiederverwerten, größere Programme sind mangels Strukturierung sehr unübersichtlich. Abhilfe brachte hier die Entwicklung strukturierter Programmiersprachen wie Algol, Pascal oder C. Diese unterstützten

- die Steuerung des Programmablaufs durch Verzweigungen oder Schleifen und
- die Modularisierung des Codes durch Funktionen, d. h., für häufig benötigte Teilprobleme wie zum Beispiel die Ausgabe eines Textes konnte man eine eigene Funktion definieren, die genau diese Aufgabe übernahm. Im Rest des Programms brauchte man dann zur Ausgabe des Textes nur noch die entsprechende Funktion aufzurufen (anstatt immer wieder den Block mit den Anweisungen zur Textausgabe zu kopieren).

C, das Anfang der 1970er-Jahre speziell für die Implementierung des UNIX-Betriebssystems entwickelt wurde, nahm unter diesen Programmiersprachen schnell eine Sonderstellung ein, denn

- C war strukturiert,
- C war für eine höhere Programmiersprache sehr maschinennah und damit auch sehr leistungsfähig,
- C-Programme waren klein und schnell.

Die beiden letzten Punkte waren für den Erfolg von C entscheidend. Bedenken Sie, dass wir uns in den 1970ern und 1980ern befinden. Die ersten PCs kamen auf den Markt, 1976 der erste Apple-Computer, 1981 der erste IBM-PC. 1986 stellte IBM den AT vor, in der Standardausführung mit 265 KByte Arbeitsspeicher, 20 MByte Festplatte und einer Taktung von 6 MHz. Die Dimensionen dürften deutlich machen, worauf es zu diesen Zeiten bei der Programmierung ankam: Die Programme mussten klein sein (damit sie auf die Festplatte und in den Arbeitsspeicher passten) und sie mussten schnell sein, damit man trotz der (aus heutiger Sicht) langsamem Prozessoren vernünftig mit ihnen arbeiten konnte. Mit C konnte man Programme schreiben, die diese Anforderungen bestens erfüllten.

C-Programme waren schnell, weil sie kompiliert wurden und weil die Maschinennähe der Sprache es dem Compiler zudem leicht machte, den Quelltext in optimalen Maschinencode zu übersetzen.

Und C-Programme waren klein, weil die Sprache C auf Symbole und Operatoren setzte statt auf gut lesbare Befehle. Das Grundgerüst einer if-else-Verzweigung (die anhand einer Bedingung entscheidet, welcher von zwei Anweisungsblöcken auszuführen ist) umfasste in Pascal beispielsweise 29 Zeichen:

```
if Bedingung then begin
  Anweisungen
  end
else begin
```

```
    Anweisungen  
end;
```

während C mit gerade einmal zwölf Zeichen auskam:

```
if Bedingung {  
    Anweisungen  
} else {  
    Anweisungen  
}
```

Diese sehr kryptische, aber eben auch prägnante und Speicherplatz sparende Syntax hat C++ von C übernommen.

1.1.3 Chips sind billig, Programmierer teuer

Mit der rasanten Entwicklung im Hardwarebereich und dem Boom in der Softwarebranche verschoben sich in den 1990er-Jahren zunehmend die Ausgangsbedingungen. Die Rechner wurden immer schneller, die Speichermedien immer dichter – für Standardsoftware lohnte es sich nicht mehr, tage-, ja wochenlang mit der Optimierung eines Programms zu verbringen, nur damit dieses noch einen Tick schneller lief oder einige KByte weniger Speicher belegte. Dafür wurden die Programme immer umfangreicher und damit auch immer unübersichtlicher und schwerer zu warten. In der Softwarebranche begann man umzudenken. Designziele wie Schnelligkeit und Schlankheit der ausgelieferten Software traten hinter den Forderungen nach benutzerfreundlicher, leistungsfähiger Software und kostengünstiger Entwicklung und Wartung zurück. Allein mit strukturierter Programmierung war dies nicht mehr zu leisten, zu hinderlich war vor allem die unnatürliche Trennung von Daten und Funktionen. Diese Trennung wurde im objektorientierten Programmiermodell aufgehoben. Das objektorientierte Programmiermodell führte eine neue Form der Repräsentation von Daten und, darauf aufbauend, neue Techniken zur Strukturierung und Wiederverwertung von Code ein, ohne die moderne Softwareentwicklung kaum noch denkbar ist. Obwohl es schon sehr früh Programmiersprachen gab, die rein objektorientiert waren, begann der Siegeszug der objektorientierten Programmierung erst Mitte der 1980er-Jahre als Bjarne Stroustrup die Sprache C um objektorientierte Konzepte erweiterte und damit C++ aus der Taufe hob. C++ ist eine Übermenge von C, d.h., man kann mit einem C++-Compiler nicht nur in C++ programmierte, objektorientierte Programme kompilieren, sondern auch ältere C-Programme. Dies war und ist sehr wichtig, denn schließlich gibt es große Mengen an bestehendem C-Code, der weiterhin gepflegt werden will.



Paradigmen moderner Programmiersprachen

Für den Prozessor besteht ein Programm aus einer Abfolge von Maschinenbefehlen. Heißt dies aber, dass auch die Quelltexte der verschiedenen Programmiersprachen aus einer Abfolge von Maschinenbefehlen bestehen müssen? Nein! Die höheren Programmiersprachen organisieren daher ihre Quelltexte in Konstrukten, die sich stärker am menschlichen Denken orientieren, und benutzen den

Compiler oder Interpreter dazu, diese Konstrukte in Maschinenbefehle für den Prozessor umzuwandeln.

Die verschiedenen Modelle zur Strukturierung des Quelltextes bezeichnet man auch als *Paradigmen*. Das klassische Paradigma der imperativen Programmierung organisiert den Quelltext z.B. in Variablen und Funktionen. Die Variablen dienen zum Speichern von Daten und mit den Funktionen können die Daten manipuliert und bearbeitet werden. Diesem Paradigma gehören Sprachen wie C oder Pascal an. Die fertigen Programme sind sehr schlank und meist recht schnell in der Ausführung. Bei größeren Projekten ist es aber meist schwierig, die Übersicht zu behalten, welche Daten von welchen Funktionen in welcher Reihenfolge bearbeitet werden dürfen oder müssen.

An diesem Punkt setzt das Paradigma der objektorientierten Programmierung an. Es fasst zusammengehörende Daten und ihre verarbeitenden Funktionen zur höheren Organisationsform eines Objekts zusammen und beschreibt gleichartige Objekte in einer Klassendefinition.³ (Ein OOP-Programm zur Berechnung von Rechtecken würde z.B. die Variablen `laenge` und `breite` sowie die Funktionen `BerechneUmfang()`, `BerechneInhalt()` zu einer Klasse `Rechteck` zusammenfassen und dann für jedes zu berechnende Rechteck ein Objekt der Klasse `Rechteck` erzeugen.)

Obwohl es beim Schreiben einer Klassendefinition mehr zu beachten gilt als beim Aufsetzen von ein paar Funktionen, lohnt sich der Mehraufwand, denn die anschließende Arbeit mit den Objekten ist viel einfacher und sicherer als die Arbeit mit den Funktionen. Aus diesem Grunde ist das objektorientierte Paradigma für größere Projekte weitaus besser geeignet als die imperativen Programmierung. Dass objektorientierte Programme etwas umfangreicher und minimal langsamer als imperativen Programme sind, spielt bei der Leistungsfähigkeit der heutigen Rechner meist keine Rolle mehr.

1.1.4 Fassen wir zusammen

Sie haben nun bereits einiges über Herkunft und Konzeption der Sprache C++ erfahren. Wenn Sie bisher noch nie programmiert haben, werden Sie vielleicht erschrocken sein, weil Sie im Zuge der Ausführungen mit einer Vielzahl von Begriffen konfrontiert wurden, die Sie nur schwer einordnen oder verstehen konnten (Maschinencode, Compiler, Funktionen, Klassen, Objekte, Programmierparadigmen etc.). Keine Sorge – bald werden Ihnen diese Begriffe so geläufig sein wie die ersten Zeilen aus Schillers Ballade „Die Bürgschaft“⁴. Im Moment sollten Sie nur Folgendes verstanden haben:

³ Einige objektorientierte Sprachen erlauben auch die direkte Definition von Objekten oder benutzen andere Begriffe, um die verschiedenen OOP-Konzepte zu bezeichnen.

⁴ Sollten Sie die „Bürgschaft“ noch nicht gelesen haben, holen Sie dies unbedingt nach. Programmieren ist ein Spaß, humanistische Bildung eine Muss.

**Merksatz**

C++ setzt auf C auf und ist letzten Endes so etwas wie eine Überarbeitung und Erweiterung von C um objektorientierte Konzepte.⁵

Dies führte dazu, dass

- C++ eine Hybridsprache ist, in der man sowohl rein strukturiert (wie in C) als auch objektorientiert programmieren kann (und vermischen lassen sich beide Programmierstile natürlich auch).
- C++ die spartanische Syntax von C übernommen hat.
- C++ für etliche Programmieraufgaben (etwa die Arbeit mit Texten – oder „Strings“, wie es im Programmierjargon heißt) doppelte Lösungen anbietet: einmal die von C geerbte Lösung und dann noch die eigene, objektorientierte Lösung.
- C++ ein wahres Ungetüm von Programmiersprache ist: enorm leistungsfähig und vielseitig, aber gerade deshalb auch sehr unübersichtlich. Eine Sprache, die für jeden Programmierer zugleich Segen wie Fluch sein kann.

■ 1.2 Von der Idee zum fertigen Programm

Die Entwicklung von Programmen läuft unabhängig von der verwendeten Sprache üblicherweise nach dem folgenden Muster ab:

1. Man hat ein Problem, eine Idee, eine Aufgabe, zu deren Lösung man einen Computer einsetzen möchte.

So könnten Sie zum Beispiel daran interessiert sein, aus einem Betrag den Mehrwertsteueranteil herauszurechnen.

2. Als Nächstes wird die Aufgabe als Algorithmus, also als eine Folge von Befehlen in natürlicher Sprache formuliert. Größere Probleme werden dabei in Teilaufgaben und Teilaspekte aufgeteilt. (Ob der Algorithmus tatsächlich auf dem Papier oder nur im Kopf des Programmierers entwickelt wird, hängt von der Komplexität der Aufgabe und der Genialität des Programmierers ab.)

Unser Algorithmus ist recht einfach und besteht aus vier Teilaufgaben:

- *den Gesamtbetrag vom Benutzer abfragen,*
- *den Gesamtbetrag in das Programm einlesen,*
- *den Anteil der Mehrwertsteuer berechnen,*
- *das Ergebnis ausgeben.*

⁵ Tatsächlich ist jeder gute C++-Compiler auch in der Lage, (nahezu) beliebigen C-Code zu übersetzen. (Es gibt nur ganz wenige Punkte, in denen C++ die von C übernommene Syntax und Semantik verändert hat.) Umgekehrt gab es in der Übergangszeit – und gibt es wohl auch heute noch – nicht wenige Programmierer, die in C++ mehr oder weniger guten, alten C-Code schrieben.

3. Der Algorithmus wird vom Programmierer in die Anweisungen einer Programmiersprache umgesetzt. Dies ergibt den sogenannten Quelltext oder Quellcode.

Der Quelltext unseres kleinen Beispieldiagramms könnte etwa wie folgt aussehen:⁶

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

int main()
{
    double preis;
    double mwst;

    cout << endl;

    // 1. Eingabe von Benutzer anfordern
    cout << " Geben Sie den Preis ein: ";

    // 2. Eingabe einlesen
    cin >> preis;

    // 3. Mehrwertsteueranteil berechnen
    mwst = 0.19 * preis / 1.19;

    // 4. Berechneten Mehrwertsteueranteil ausgeben
    cout << " In dem Preis sind " << mwst
        << " Euro Mehrwertsteuer enthalten." << endl;

    cout << endl;
    return 0;
}
```

4. Dieser Quelltext muss dann durch ein spezielles Programm, den *Compiler*, in Maschinenanweisungen (Binärkode) übersetzt werden, die das eigentliche Herz des Computers – der Prozessor – versteht und ausführen kann. Das Ergebnis ist eine ausführbare Datei (eben ein Programm), die unter Microsoft Windows standardmäßig die Dateierweiterung .exe trägt.
5. Das ausführbare Programm kann auf jedem Rechner, dessen Prozessor den erzeugten Maschinencode versteht, gestartet werden.



Was genau ist ein Programm?

Die noch in Deutsch formulierten Befehle? Die in C++ formulierten Befehle? Oder die binär kodierten Maschinenanweisungen? Im weitesten Sinne können Sie in allen drei Fällen von Ihrem Programm reden. Wenn Sie es dagegen genau nehmen wollen, bezeichnen Sie die noch in Ihrer Muttersprache aufgesetzte Befehlsfolge als *Algorithmus*, die in C++ formulierte Version des Algorithmus als Quelltext Ihres Programms und erst den vom Compiler erzeugten Maschinencode als Ihr ausführbares Programm.

⁶ Vermutlich wird Ihnen dieser Code vollkommen unverständlich erscheinen. Machen Sie sich darüber keine Gedanken. Zum einen ist der Code schon recht komplex, zum anderen geht es hier ja nicht darum, dass Sie das Beispiel nachstellen, sondern nur darum, dass Sie einmal einen C++-Quelltext gesehen haben.

■ 1.3 Näher hingeschaut: der C++-Compiler

Ich muss noch einmal auf den C++-Compiler zurückkommen, der unsere C++-Quelltexte übersetzt. Nicht nur weil er ein strenger Lehrer ist, auf den man Novizen vorbereiten muss, sondern auch, weil mit seiner Arbeit ein wichtiges C++-Konzept verbunden ist: das der Deklaration.

Lesen Sie sich die folgenden Ausführungen entspannt durch. Sie müssen nicht auf Anhieb alles verstehen, aber Sie sollten sich unbedingt mit den angesprochenen Konzepten schon einmal grob vertraut machen. Wenn Sie dann in den weiteren Kapiteln mit den besagten Konzepten nochmals konfrontiert werden, bringen Sie bereits grundlegende Vorkenntnisse mit und wissen, dass Sie hier bei Bedarf noch einmal nachlesen können.

1.3.1 Der Compiler ist ein strenger Lehrer

Bevor der Compiler eine Quelltextdatei übersetzen kann, muss er diese analysieren. Dabei prüft er auch gleich, ob die Syntax korrekt ist und die einzelnen Bausteine richtig verwendet wurden. Bei der kleinsten Ungenauigkeit verweigert er die Arbeit und gibt eine Fehlermeldung aus – was ihn bei Anfängern nicht gerade beliebt macht.

Trotzdem ist der Compiler nicht Ihr Feind. Betrachten Sie ihn vielmehr als strengen, aber gerechten Kritiker, der nur das Beste für Ihr Programm will.



Wenn Sie beim Kompilieren Fehlermeldungen angezeigt bekommen, kümmern Sie sich zunächst nur um die erste Fehlermeldung. Nachdem Sie den zugehörigen Fehler korrigiert haben, kompilieren Sie den Quelltext neu. Nicht selten verschwinden dann auch nachfolgende Fehlermeldungen, bei denen es sich um Folgefehler handelt.

Oben wurde bereits erwähnt, dass der Compiler nicht nur prüft, ob die Syntax korrekt ist, sondern auch, ob die einzelnen Bausteine ihrer Bestimmung gemäß verwendet werden. Dies wirft die Frage auf, woher der Compiler denn weiß, wie die Bausteine korrekt verwendet werden?

Nun, in einem C++-Quelltext gibt es grundsätzlich drei Gruppen von Bausteinen:

- Schlüsselwörter wie `if`, `class` oder `for`,
- symbolische Zeichen wie `+`, `::` oder `//`,
- Elemente, die der Programmierer definiert (oder die aus einer Bibliothek stammen).

Die beiden ersten Gruppen sind fest in der Sprache verankert, sodass es logisch ist, dass ein C++-Compiler weiß, wie die Elemente dieser Gruppen zu verwenden sind. Der Vorrat an diesen Elementen ist allerdings so klein, dass man damit kein sinnvolles Programm schreiben kann. Der C++-Programmierer ist also darauf angewiesen, dass er neue Elemente einführt: beispielsweise Variablen, in denen er Daten abspeichern kann, oder Funktionen, mit denen er Daten bearbeiten kann, oder neue Datentypen, die beschreiben, welche Art von