Arnold Willemer

Einstieg in C++

Galileo Computing

Inhalt

Vorwort 13

1 C++ für Hektiker 19

1.1 Ein Programm 19
1.2 Abfrage und Schleifen 24
1.2.1 Abfrage und boolesche Ausdrücke 24
1.2.2 Die while-Schleife 26
1.2.3 Die for-Schleife 27
1.3 Arrays 28
1.4 Funktionen 33
1.4.1 Programmaufteilung 33
1.4.2 Rückgabewert 35
1.4.3 Parameter 37
1.5 Klassen 40
1.5.1 Konstruktor 45
1.5.2 Vererbung 49
1.5.3 Polymorphie 51
1.6 Templates 53

2 Einstieg in die Programmierung 59

2.1 Programmieren 59
2.1.1 Start eines Programms 59
2.1.2 Eintippen, übersetzen, ausführen 60
2.1.3 Der Algorithmus 62
2.1.4 Die Sprache C++ 63
2.1.5 Fragen zur Selbstkontrolle 67
2.2 Grundgerüst eines Programms 67
2.2.1 Kommentare 68
2.2.2 Anweisungen 70
2.2.3 Blöcke 71
2.3 Variablen 72
2.3.1 Variablendefinition 72
2.3.2 Geltungsbereich 74
2.3.3 Namensregeln und Syntaxgraph 75
2.3.4 Typen 78
2.3.5 Syntax der Variablendefinition 88

2.3.6 Konstanten **89**

2.4 Verarbeitung 98
2.4.1 Zuweisung **98**
2.4.2 Rechenkünstler **99**
2.4.3 Abkürzungen **100**
2.4.4 Zufallsfunktionen **103**
2.4.5 Typumwandlung **105**

2.5 Ein- und Ausgabe 106
2.5.1 Ausgabestrom nach cout **106**
2.5.2 Formatierte Ausgabe **107**
2.5.3 Eingabestrom aus cin **108**

2.6 Übungen 109

3 Ablaufsteuerung **111**

3.1 Verzweigungen 112
3.1.1 Nur unter einer Bedingung: if **112**
3.1.2 Andernfalls: else **114**
3.1.3 Fall für Fall: switch case **117**
3.1.4 Kurzabfrage mit dem Fragezeichen **121**

3.2 Boolesche Ausdrücke 122
3.2.1 Variablen und Konstanten **122**
3.2.2 Operatoren **123**
3.2.3 Verknüpfung von booleschen Ausdrücken **125**

3.3 Immer diese Wiederholungen: Schleifen 129
3.3.1 Kopfgesteuert: while **129**
3.3.2 Fußgesteuert: do. . . while **132**
3.3.3 Abgezählt: for **134**
3.3.4 Schleifensprünge: break und continue **137**
3.3.5 Der brutale Sprung: goto **139**

3.4 Beispiele 140
3.4.1 Primzahlen **140**
3.4.2 Größter gemeinsamer Teiler **145**

3.5 Übungen 148

4 Datentypen und -strukturen **149**

4.1 Das Array 149
4.1.1 Beispiel Bubblesort **154**
4.1.2 Zuweisung von Arrays **157**
4.1.3 C-Zeichenketten **158**
4.1.4 Beispiel: Zahleneingabe auswerten **159**

4.1.5 Mehrere Dimensionen 162
4.1.6 Beispiel: Bermuda 163

4.2 Der Zeiger und die Adresse 165
4.2.1 Indirekter Zugriff 166
4.2.2 Arrays und Zeiger 167
4.2.3 Zeigerarithmetik 168
4.2.4 Konstante Zeiger 170
4.2.5 Anonyme Zeiger 171

4.3 Die Struktur 171
4.3.1 Beispiel: Bermuda 174

4.4 Dynamische Strukturen 176
4.4.1 Anlegen und Freigeben von Speicher 176
4.4.2 Zur Laufzeit erzeugte Arrays 178
4.4.3 Verkettete Listen 178

4.5 Die Union 180
4.5.1 Aufzählungstyp enum 181

4.6 Typen definieren 182

5 Funktionen 183

5.1 Parameter 187
5.1.1 Prototypen 190
5.1.2 Zeiger als Parameter 191
5.1.3 Arrays als Parameter 193
5.1.4 Referenz-Parameter 195
5.1.5 Beispiel: Stack 196
5.1.6 Vorbelegte Parameter 198
5.1.7 Die Parameter der Funktion main 199
5.1.8 Variable Anzahl von Parametern 201

5.2 Überladen von Funktionen 202

5.3 Kurz und schnell: Inline-Funktionen 203

5.4 Top-Down 205
5.4.1 Beispiel: Bermuda 205

5.5 Geltungsbereich von Variablen 210
5.5.1 Globale Variablen 211
5.5.2 Lokale Variablen 212
5.5.3 Statische Variablen 212

5.6 Selbstaufrufende Funktionen 214
5.6.1 Einsatzbereich 215
5.6.2 Beispiel: binärer Baum 216
5.6.3 Türme von Hanoi 218

5.6.4 Beispiel: Taschenrechner **220**

5.7 Funktionszeiger 226

6 Klassen 229

6.1 Die Klasse als erweiterte Struktur 230
6.1.1 Funktion und Struktur heiraten **231**
6.1.2 Zugriff auf Klassenelemente **234**

6.2 Geburt und Tod eines Objekts 235
6.2.1 Konstruktor und Destruktor **235**
6.2.2 Konstruktor und Parameter **237**

6.3 Öffentlichkeit und Privatsphäre 240
6.3.1 private und public **240**
6.3.2 Beispiel: Stack **243**
6.3.3 Freunde **246**

6.4 Kopierkonstruktor 247

6.5 Überladen von Funktionen 250

6.6 Kür: Überladen von Operatoren 251
6.6.1 Addition **253**
6.6.2 Globale Operatorfunktionen **255**
6.6.3 Inkrementieren und Dekrementieren **256**
6.6.4 Der Zuweisungsoperator **257**
6.6.5 Die Vergleichsoperatoren **261**
6.6.6 Der Ausgabeoperator **262**
6.6.7 Der Indexoperator **264**
6.6.8 Aufrufoperator () **266**
6.6.9 Konvertierungsoperator **267**

6.7 Attribute 267
6.7.1 Statische Variablen und Funktionen in Klassen **267**
6.7.2 Konstanten **269**

6.8 Vererbung 272
6.8.1 Zugriff auf die Vorfahren **276**
6.8.2 Konstruktoren und Zuweisung **280**
6.8.3 Mehrfachvererbung **282**
6.8.4 Polymorphie durch virtuelle Funktionen **283**

6.9 Klassendefinition und Syntaxgraph 293

7 Weitere Besonderheiten von C++ 297

7.1 Generische Programmierung 297
7.1.1 Template-Funktionen **297**

7.1.2 Template-Klassen **301**

7.2 Makroprogrammierung mit #define 304

7.3 Namensräume 306

7.3.1 Definition eines Namensraums **306**

7.3.2 Zugriff **308**

7.3.3 Besondere Namensräume **308**

7.3.4 Anonyme Namensräume **308**

7.3.5 Syntaxgraph **310**

7.4 Katastrophenschutz mit try und catch 310

7.4.1 Eigene Ausnahmen erzeugen **311**

7.4.2 Erstellen von Fehlerklassen **315**

7.4.3 Die Ausnahmen der Standardbibliotheken **318**

8 Bibliotheken 327

8.1 Einbinden von Bibliotheken 327

8.1.1 Compiler und Header **327**

8.1.2 Linker und Bibliotheken **329**

8.2 Zeichenketten: String 330

8.2.1 Andere String-Bibliotheken **336**

8.2.2 Klassische C-Funktionen **336**

8.3 Ein- und Ausgabe: iostream 343

8.3.1 Ausgabe an cout und cerr **343**

8.3.2 Eingabe über cin **344**

8.3.3 Manipulatoren **346**

8.4 Dateioperationen 350

8.4.1 Öffnen und Schließen **351**

8.4.2 Lesen und Schreiben **353**

8.4.3 Zustandsbeobachtung **358**

8.4.4 Dateizugriffe nach ANSI-C **360**

8.4.5 Dateisystemkommandos **363**

8.4.6 Datei-Eigenschaften ermitteln **366**

8.5 Mathematische Funktionen 369

8.5.1 Die mathematische Standardbibliothek **369**

8.5.2 Komplexe Zahlen **372**

8.6 Die Standard Template Library (STL) 373

8.6.1 Algorithmen und Arrays **373**

8.6.2 Container und Iteratoren **381**

8.6.3 Die Container-Klasse vector **382**

8.6.4 Die Container-Klasse deque **387**

8.6.5 Die Container-Klasse list **389**

8.6.6 Container-Adapter **395**

8.6.7 Die Container-Klassen set und multiset 398
8.6.8 Die Container-Klassen map und multimap 400
8.6.9 Iteratortypen 402
8.6.10 Die Template-Klasse bitset 403

8.7 Zeitfunktionen 404
8.7.1 Datum und Uhrzeit 404
8.7.2 Zeit stoppen 406

8.8 Bibliotheken im Eigenbau 408
8.8.1 Aufteilen des Quelltextes 408
8.8.2 Verborgene Implementation 412
8.8.3 Dynamische Bibliotheken 414

9 Professionelles Programmieren 419

9.1 Programmentwurf 419
9.1.1 Vom Auftrag zum Prototyp 419
9.1.2 Objektorientierung 420
9.1.3 Grafische Darstellungen 423

9.2 Qualitätssicherung 425
9.2.1 Programmierung 425
9.2.2 Dokumentation 428
9.2.3 Testen 429

10 Systemnahe Programmierung 431

10.1 Bit-Operatoren 431
10.2 Shift-Operatoren 433
10.3 Zugriff auf Hardware-Adressen 434
10.4 Bit-Strukturen 435
10.5 Portabilität und der Präprozessor 436

A Programmierumgebung 441

A.1 Quelltexteditor 441
A.2 Compiler 442
A.3 Debugger 443
A.4 GNU-Compiler unter UNIX/Linux 444
A.4.1 Compiler-Optionen 444
A.4.2 make 445
A.4.3 gdb GNU debug 450

A.5 **Borland C++-Compiler** 452
A.6 **Microsoft Visual C++** 453
A.6.1 Neues Projekt 454
A.6.2 Kompilieren und starten 456
A.7 **Borland C++-Builder/Kylix** 457
A.7.1 Neues Projekt 458
A.7.2 Kompilieren und starten 460
A.8 **Bloodshed Dev-C++ (CD)** 461
A.8.1 Ein Projekt anlegen 461
A.8.2 Übersetzen und starten 463

B Musterlösungen 467

C Glossar 487

D Literatur 491

Index 493

Vorwort

Ja, ich weiß, es gibt schon das eine oder andere Buch über C++. Allerdings kann ein Buch über C++ mit vielen Intentionen geschrieben worden sein. Da gibt es das Buch von Bjarne Stroustrup, dem Erfinder von C++ – ein sehr exaktes, sehr umfassendes Buch. Es stellt den Maßstab für alle Compilerbauer dar. Aber für den Anfänger ist es leider nicht gedacht.

Anspruch

Mein Anspruch an dieses Buch ist, dass es leicht verständlich ist. Wenn ein Anfänger C++ lernen will, sollte er mit anschaulichen Beispielen an C++ herangeführt werden. Und wenn aus dem Anfänger ein Fortgeschrittener geworden ist, sollte er die meisten Fragen, die mit dem Handwerkszeug C++ zu tun haben, auch in diesem Buch beantwortet finden. Ich mag Bücher nicht, die bei der Hälfte aufhören.

Vorkenntnisse

Ich setze voraus, dass Sie wissen, was ein Computer ist, dass Sie damit grundlegend umgehen können, Lust haben, sich auf das Programmieren einzulassen, und natürlich lesen können. Ich gehe nicht davon aus, dass Sie bereits irgendeine Programmiersprache beherrschen.

Einige Bücher über C++ setzen voraus, dass der Leser C beherrscht. Andere sind zweigeteilt. Im ersten Teil wird C behandelt, und im zweiten Teil finden Sie die Erweiterungen, die C++ gebracht hat. Das hat aus meiner Sicht den Nachteil, dass Sie im ersten Teil Dinge lernen, die Sie für C++ später nicht mehr brauchen, und vor allem, dass der zweite Teil Dinge enthält, die didaktisch und inhaltlich viel weiter nach vorn gehören. Falls Sie also C bereits beherrschen, sollten Sie dennoch die ersten Kapitel lesen. Sie werden immer wieder auf Neuigkeiten stoßen, auch wenn Sie vieles schon kennen. Immerhin werden Sie durch Ihre Vorkenntnisse schnell vorwärts kommen.

Systemunabhängig

Die Sprache C++ selbst ist systemunabhängig und das gilt auch für dieses Buch. Systemspezialitäten werden höchstens am Rande kurz erwähnt. Der Inhalt des Buches hält sich weitgehend an den ANSI-Standard. Die Beispiele wurden mit dem GNU-Compiler erstellt und getestet, der auf allen Systemen verfügbar ist. Wenn Zweifel über die Portierbarkeit aufkamen, habe ich Microsoft Visual C++ und Borlands C++ Builder zu Rate gezogen.

Syntaxgrafen

Zum leichteren Verständnis der Sprachkonstrukte habe ich Syntaxgrafen verwendet. Diese sind in den letzten Jahren offensichtlich aus der Mode gekommen. In der aktuellen Literatur sind sie jedenfalls kaum zu finden.

Dennoch halte ich sie wegen ihrer Anschaulichkeit für eine wertvolle Hilfe, insbesondere für den Anfänger.

Struktogramme Auch die Nassi-Schneidermann-Diagramme habe ich wieder aufleben lassen, weil ich denke, dass sie besonders in Kapitel Kapitel 3, »Ablaufsteuerung«, ein Gefühl für sauber strukturierte Abläufe vermitteln.

Zweimal C++ Sie finden im ersten Kapitel einen Schnelleinstieg in C++. Er ist für Leute gedacht, die gern schnell und vielleicht etwas ungeduldig zu Ergebnissen kommen möchten. Er könnte auch als Basis-Skript für C++-Kurse dienen, die ja auch nicht jedes Detail berücksichtigen können. Da in diesem Kapitel manches sehr knapp behandelt wird, wird immer wieder auf die Stellen im Buch verwiesen, die ein Thema ausführlicher behandeln. Sollte Ihnen das erste Kapitel doch etwas zu hektisch sein, blättern Sie weiter, und beginnen in Ruhe noch einmal neu ab dem zweiten Kapitel. Die hinteren Kapitel sind völlig unabhängig vom ersten Kapitel. Sie behandeln alle Themen ausgiebig und detailliert, da das Buch ja schließlich später auch als Nachschlagewerk nützlich sein soll.

Reihenfolgen Weil die Einführung der Klassen das Besondere an C++ gegenüber C ist und weil die Klassen die Software-Entwicklung so sehr vereinfachen, war die Versuchung groß, das Thema möglichst früh zu behandeln. Das hätte allerdings dazu geführt, dass die Lernkurve an dieser Stelle sehr steil geworden wäre. Es wäre der Eindruck entstanden, das Thema Klassen sei schwierig. Ich habe stattdessen versucht, die Themen in eine Reihenfolge zu bringen, die gewährleistet, dass der Leser immer schrittweise folgen kann. In groben Zügen werden folgende Themen behandelt:

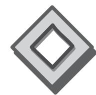
- Variablen, Konstanten und Anweisungen
- Ablaufsteuerung, Abfragen, Schleifen und boolesche Ausdrücke
- Datentypen: Arrays, Zeiger und Strukturen
- Funktionen und ihre Parameter
- Klassen und Vererbung
- Templates, Namensräume und die Ausnahmebehandlung

Praxisthemen Im Teil Praxis sind die Bibliotheken zu finden. Dazu gehören auch die klassischen C-Bibliotheken. Die Ein- und Ausgabeströme, die Strings und schließlich die STL mit ihren Algorithmen, Containern und Iteratoren werden hier behandelt. Im nachfolgenden Kapitel werden Programmentwurf und Qualitätssicherung beleuchtet. Ein Einblick in die Werkzeuge zur systemnahen Programmierung schließt diesen Teil ab.

Sie finden bei vielen Listings hinter der Überschrift in Klammern einen Dateinamen. Dann befindet sich das Listing unter diesem Namen auf der CD.

Schreibweisen

Zur besseren Lesbarkeit sind folgende Konventionen eingeführt worden: `Schlüssselwörter` werden in nichtproportionaler Schrift gesetzt. **Variablen** und **Klassen** erscheinen in fetter Schrift. Funktionsaufrufe wie open() sind grundsätzlich mit Klammern dargestellt.



Dieses Symbol am Rand weist auf ein Beispielszenario hin. Es werden nicht alle Beispiele im Buch so markiert, sondern nur jene, die im Text weiter ausgebreitet werden.

Mit diesem Symbol werden Hinweise gekennzeichnet, wie Sie sich das Leben etwas erleichtern.

Dieses Symbol soll auf Stolperfallen hinweisen. Hier schleichen sich entweder leicht Fehler ein, oder es geht um Dinge, die zu einem Datenverlust oder zum Verlust der Systemsicherheit führen können.

Danksagungen

Ich bin dankbar für die Unterstützung, die mir bei diesem Buch zuteil wurde. Zunächst musste meine Familie wieder einmal darunter leiden, dass ich wenig Zeit für sie hatte. Meine Söhne litten mehr darunter, dass sie häufig auf das Computerspielen verzichten mussten, weil ich alle Computer im Arbeitszimmer blockierte. Meine Frau Andrea sorgte dafür, dass solche Einschränkungen den Familienfrieden nicht zerstörten, und kontrollierte darüber hinaus sogar, ob das Buch auch für den Laien noch lesbar ist. Mein Lektor Stephan Mattescheck versorgte mich mit Informationen und Hinweisen. Ansonsten stand er für Fragen ständig zur Verfügung. Er und der Verlag Galileo Computing ließen mir weitestgehende Freiheiten. Friederike Daenecke bereinigte meine schlimmsten Verbrechen gegen die deutsche Sprache. Iris Warkus sorgte für das gute Aussehen des Buches. Ansonsten bleibt mir noch zu erwähnen, dass ich durch Prof. Dr. Mario Dal Cin zum ersten Mal mit C++ in Berührung kam. Er machte mir viel Mut für dieses Buch. Und Stephan Engelmann unterstützte mich mit einigen Informationen.

Norgaardholz, den 6.9.2003
Arnold Willemer

Teil 1
Crash-Kurs

1 C++ für Hektiker

Von Null auf C++ auf unter 40 Seiten

Wenn Sie gern schnell ins Thema einsteigen wollen oder wenn Sie schon Vorkenntnisse haben, dann ist dieses Kapitel für Sie. Hier geht es im scharfen Tempo einmal quer durch C++. Wenn Sie besser lernen, indem Sie ordentlich Stein auf Stein legen, überschlagen Sie dieses Kapitel!

Dieses Kapitel hat etwas vom Wilden Westen: Erst wird geschossen, dann wird erklärt. Hier wird anhand einiger Beispielprogramme erläutert, wie Sie in C++ programmieren können. Die Details werden in den späteren Kapiteln ausführlich besprochen. Dazu finden Sie an den entsprechenden Stellen Hinweise auf die ausführlicheren Abschnitte. Wenn Ihnen diese Art des Lernens nicht so liegt, dann können Sie dieses Kapitel problemlos überschlagen. Es geht im zweiten Kapitel noch einmal von vorn los, dann aber sehr viel detaillierter und ruhiger.

Um alles so schön zu komprimieren, mussten natürlich ein paar Details geopfert werden. Auch sind einige Beschreibungen eher anschaulich als wissenschaftlich exakt. Dafür wissen Sie nach dem ersten Kapitel schon einmal, wo es ungefähr lang gehen wird. Selbst wenn Sie also nach dem Lesen dieses Kapitels etwas verwirrt sein sollten, haben Sie immerhin schon einen guten Überblick. Und das hat ja auch seinen Wert.

1.1 Ein Programm

Das erste Programm enthält bereits eine ganze Menge Aktivitäten. Es gibt eine Meldung auf dem Bildschirm aus, fordert eine Zahleneingabe, errechnet das Quadrat hieraus und gibt das Ergebnis auf dem Bildschirm aus. Nur Geschirrspülen kann es nicht.

Listing 1.1 Quadratzahlen

```
#include <iostream.h>
main()
{
    int Eingabe;
    int Quadrat;
    cout << "Geben Sie eine Zahl ein: ";
    cin >> Eingabe;
    Quadrat = Eingabe * Eingabe;
```

```
    cout << "Die Quadratzahl lautet " << Quadrat << endl;
}
```

Übersetzen Nachdem Sie dieses Programm in den Editor[1] eingetippt haben und beispielsweise unter dem Namen **erst.cpp** gesichert haben, müssen Sie es durch den Compiler übersetzen lassen. Wie Sie die Übersetzung auf Ihrer IDE starten, finden Sie im Anhang. Dort ist das Übersetzen von Programmen und das anschließende Starten für die gängigsten Compiler beschrieben. Unter UNIX oder Linux geben Sie einfach folgenden Befehl ein:

```
> c++ erst.cpp
>
```

Der Compiler jammert nicht, also ist er zufrieden. Nach der erfolgreichen Kompilierung kann das Programm gestartet werden. Wenn Sie eine IDE verwenden, können Sie das Programm direkt von dort starten. Auch dazu finden Sie im Anhang eine Anleitung. Auf der Kommandozeile ist das Programm unter dem Namen **a.out** gespeichert worden, und es kann gestartet werden:

```
> ./a.out
Geben Sie eine Zahl ein:
```

Ausführung Nach dem Start wird das Programm auf dem Bildschirm die Aufforderung »Geben Sie eine Zahl ein:« anzeigen. Es wartet darauf, dass der Benutzer seine Zahl eingibt und mit der Return-Taste abschließt. Dann erscheint auf dem Bildschirm die Meldung »Die Quadratzahl lautet«, gefolgt von dem Ergebnis.

include Gehen wir das Programm Zeile für Zeile durch.

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

Die erste Zeile enthält einen #include-Befehl. Dieser Befehl liest die Datei **iostream.h** an dieser Stelle in den Quelltext ein. Er wird gebraucht, um Informationen über Programmbibliotheken einzubinden. In diesem Fall geht es um die Bibliothek für die Ein- und Ausgabe (iostream). Die grundlegende Funktionalität dafür müssen Sie nicht selbst schreiben, sondern sie wird mit dem Compiler in einer Standardbibliothek mitgeliefert. Sie können sie einfach nutzen. Wenn also eine Ein- oder Ausgabe im Programm verwendet werden soll, muss zu Anfang der Datei dieser Befehl stehen, der die Datei **iostream.h** einfügt. Die Datei **iostream.h** enthält die Informationen, die der Compiler braucht, um mit der Bibliothek ar-

1 Nähere Informationen zum Thema Editor finden Sie auf Seite 441.

beiten zu können. Welche Dateien per `#include` eingebunden werden müssen, finden Sie in der Dokumentation der verwendeten Funktionen beschrieben bzw. an den entsprechenden Stellen in diesem Buch.

```
main()
{
```

main(){ }

Der Name `main()` leitet die Hauptfunktion des Programms ein. Jedes C- oder C++-Programm hat genau eine Funktion `main()`. Hier beginnt nach dem Programmstart das Programm. Jede Funktion, also auch `main()`, enthält eine Reihe von Anweisungen, die in geschweiften Klammern stehen. Die öffnende geschweifte Klammer steht in der nächsten Zeile und kündigt den Beginn des Programms an. Die dazugehörige schließende Klammer finden Sie am Ende des Listings.

Einrückung

Wie Sie sehen, werden die nachfolgenden Zeilen etwas eingerückt. Das ist keineswegs erforderlich. In C++ können Sie Ihren Programmtext so anordnen, wie es Ihnen Freude bereitet. Allerdings hat es sich als praktisch erwiesen, nach jeder öffnenden geschweiften Klammer etwas einzurücken und diese Einrückung beim Schließen der Klammer wieder zurückzunehmen. So stehen zusammengehörige Klammern immer auf gleicher Höhe, und Sie können schneller erkennen, ob Sie eine Klammer vergessen haben.

```
int Eingabe;
int Quadrat;
```

int

Die erste Zeile im Funktionsrumpf enthält die erste Anweisung. Eine Anweisung könnte man auch als vollständigen Befehl der Sprache C++ bezeichnen. Jede Anweisung wird durch ein Semikolon abgeschlossen. Diese Anweisung ist eine Variablendefinition. Es wird festgelegt, dass es eine Variable namens **Eingabe** gibt, die ganze Zahlen aufnehmen kann. Dass die Variable ganze Zahlen aufnehmen kann, wird durch den Variablentyp `int` signalisiert, der vor dem Variablennamen steht. Das Schlüsselwort `int` steht für Integer, das ist der englische Begriff für ganze Zahlen, also Zahlen ohne Nachkommastellen. Auch die nächste Zeile ist eine Variablendefinition. Hier wird die Integer-Variable **Quadrat** definiert. Die Namen der Variablen sind frei. Ich habe die erste Variable **Eingabe** genannt, weil später die Benutzereingabe in dieser Variablen landen wird. Analog habe ich die andere Variable **Quadrat** genannt, weil sie später das Quadrat enthalten wird. Nähere Informationen zum Thema Variablen finden Sie ab Seite 72.

Neben dem Typ int gibt es die Ganzzahlentypen short für kleinere Zahlen und long für größere (siehe Seite 79). Für Fließkommazahlen, also Zahlen, die auch Nachkommastellen haben, gibt es den Typ float und für höhere Genauigkeit double (siehe Seite 84). Für einzelne Buchstaben gibt es den Typ char. Buchstaben erkennen Sie im Quelltext daran, dass sie in Hochkommata eingeschlossen sind ('A'). Dagegen schließen Anführungszeichen eine Zeichenkette, also eine Folge von Buchstaben ein ("Programmieren macht Spass"). In Variablen des Typs char können außer Buchstaben auch sehr kleine ganzzahlige Werte gespeichert werden (siehe Seite 82).

Typ	Daten
int	Ganze Zahlen
short	Ganze Zahlen (typischerweise –32.768 bis 32.767)
long	Ganze Zahlen (typischerweise –2.147.483.648 bis 2.147.483.647)
float	Fließkommazahlen (beispielsweise –2,5 oder 3,14159)
double	Fließkommazahlen höherer Genauigkeit
long double	Fließkommazahlen mit besonder großer Genauigkeit
char	Buchstaben oder ganze Zahlen von –128 bis 127

Tabelle 1.1 Grundlegende Typen

```
cout << "Geben Sie eine Zahl ein: ";
```

cout

Bildschirmausgaben werden in C++ auf das Objekt **cout** gelenkt. Das Ausgabeobjekt **cout** steht immer links, es folgen zwei Kleiner-Zeichen, die man als Umleitungsoperator bezeichnet, und dann das, was auf dem Bildschirm erscheinen soll. In diesem Fall ist es ein Text, der in Anführungszeichen eingeschlossen ist. Die Anführungszeichen selbst erscheinen nicht auf dem Bildschirm. Aber alles, was in Anführungszeichen steht, wird Zeichen für Zeichen auf den Bildschirm gegeben. In diesem Fall zeigt das Programm dem Benutzer auf dem Bildschirm an, was es von ihm möchte.

```
cin >> Eingabe;
```

cin

Das Gegenstück ist das Eingabeobjekt **cin**. Mit den zwei Größer-Zeichen werden die Daten von der Eingabe in eine Variable umgeleitet. Auf diese Weise erhält die Variable **Eingabe** ihren Wert direkt von der Tastatur.

Weitere Informationen zum Thema Ein- und Ausgabe finden Sie ab Seite 106 und ab Seite 343.

```
Quadrat = Eingabe * Eingabe;
```

Berechnung

Die Anweisung in der folgenden Zeile berechnet die Quadratzahl. Auf der linken Seite des Gleichheitszeichens befindet sich das Ziel der Berechnung. Das Ergebnis dessen, was rechts vom Gleichheitszeichen steht, wird der Variablen **Quadrat** zugewiesen. Das Gleichheitszeichen wird darum auch Zuweisungsoperator genannt. Auf der rechten Seite steht ein Ausdruck. So nennt man eine Berechnung, die ein speicherbares Ergebnis liefert. Der Stern ist das Multiplikationszeichen, also wird hier der Inhalt der Variablen **Eingabe** mit sich selbst multipliziert. Das Ergebnis wird in der Variablen **Quadrat** abgelegt. Da in der Variablen **Eingabe** die Eingabe des Benutzers gespeichert ist, befindet sich nach der Ausführung dieser Zeile das gewünschte Ergebnis in der Variablen **Quadrat**.

Neben dem Stern (*) als Multiplikationszeichen gibt es den Schrägstrich (/) als Divisionszeichen, das Pluszeichen (+) für die Addition und das Minuszeichen (–) für die Subtraktion. Tabelle 1.2 zeigt eine Übersicht der grundlegenden Operatoren. Mehr Informationen über mathematische Operatoren finden Sie ab Seite 99.

Operator	Bedeutung
+	Addition
–	Subtraktion
*	Multiplikation
/	Division
=	Zuweisung
«	Umleitung (typischerweise nach **cout**)
»	Umleitung (typischerweise von **cin**)

Tabelle 1.2 Grundlegende Operatoren

Abkürzungen

Für bestimmte Berechnungen gibt es in C++ Abkürzungen. So können Sie durch die Folge a+=5 den Inhalt der Variablen **a** um 5 erhöhen. Das ist also identisch zu a=a+5. Das funktioniert übrigens auch mit -=, *= und /=. Also würde mit der Anweisung a/=2 der Inhalt der Variablen **a** halbiert. Wenn Sie eine Variable um eins erhöhen wollen, können Sie mit a++ noch kürzer werden. Das funktioniert auch mit Minus. a-- zählt die Variable **a** um eins herunter.

```
cout << "Die Quadratzahl lautet " << Quadrat << endl;
```

nochmal cout Nun muss das Ergebnis noch auf dem Bildschirm ausgegeben werden. Schließlich will der Anwender ja wissen, was der Computer errechnet hat. Dazu wird wieder auf **cout** zurückgegriffen. Hier wird zunächst wieder ein Text in Anführungszeichen ausgegeben. Wie zuvor erscheinen zwar die Anführungszeichen nicht, aber jedes Zeichen, das der Programmierer zwischen die Anführungszeichen gestellt hat. Der Inhalt wird also nicht interpretiert, sondern direkt ausgegeben. Danach erscheint noch einmal ein Umleitungsoperator. Es folgt das Wort **Quadrat**. Da es nicht in Anführungszeichen steht, wird auf die Variable **Quadrat** zugegriffen und deren Inhalt angezeigt. **endl** bewirkt, dass die Zeile abgeschlossen wird. Die nächste Ausgabe wird in einer neuen Zeile beginnen.

1.2 Abfrage und Schleifen

Programme müssen nicht nur Zeile für Zeile ablaufen. Es ist möglich, Programmteile aufgrund des Zustands von Variablen zu übergehen oder zu wiederholen.

1.2.1 Abfrage und boolesche Ausdrücke

Mit Hilfe der Abfrage können Sie einen Anweisungsblock unter eine angegebene Bedingung stellen. Beispielsweise können Sie prüfen, ob die Zahl, durch die Sie gerade teilen wollen, vielleicht 0 ist.

Listing 1.2 Sichern gegen eine Division durch null

```
if (divisor==0)
{
    cout << "Division durch 0" << endl;
}
else
{
    quotient = dividend / divisor;
}
```

if else Die Abfrage beginnt mit dem Schlüsselwort `if`, und in Klammern folgt die Bedingung, unter der die nächste Anweisung bzw. der nachfolgende Block ausgeführt wird. Hier wird geprüft, ob der Inhalt der Variablen **divisor** gleich null ist. Dann gibt das Programm die Fehlermeldung »Division durch 0« aus. Der nachfolgende Befehl `else` leitet die Anweisung bzw. den Block ein, der ausgeführt wird, wenn die Bedingung nicht zutrifft.

Die Verwendung von `else` ist optional, kann also weggelassen werden. Nähere Informationen zu `if` und `else` finden Sie ab Seite 112.

Vergleiche

Das Symbol für die Gleichheit zweier Werte ist in C++ ein doppeltes Gleichheitszeichen (==), um es von der Zuweisung zu unterscheiden. Verwechseln Sie es nicht! Der Compiler wird Ihnen bestenfalls eine Warnung zukommen lassen, wenn Sie statt des Vergleichs eine Zuweisung in die Klammer setzen. Das Zeichen für Ungleichheit ist ein Ausrufezeichen, gefolgt von einem Gleichheitszeichen (!=). Sie können zwei Werte mit dem Kleiner-Zeichen (<) darauf überprüfen, ob der erste Wert kleiner als der zweite ist. Analog ermittelt das Größer-Zeichen (>), ob der erste Wert größer als der zweite Wert ist. Durch Anhängen eines Gleichheitszeichens kann auf kleiner oder gleich (<=) bzw. größer oder gleich (>=) getestet werden (siehe ab Seite 123).

Listing 1.3 Programmausschnitt

```
if (Eingabe>=5 && Eingabe<=10)
```

Und-Verknüpfung

Zwei kaufmännische Und-Zeichen (&&) bewirken die Und-Verknüpfung zweier logischer Ausdrücke. Im Beispiel sehen Sie, wie getestet wird, ob der Inhalt der Variablen **Eingabe** größer-gleich fünf ist und ob er kleiner-gleich zehn ist. Beide Bedingungen müssen zutreffen, damit die Gesamtbedingung zutrifft. Mit der Abfrage wird also geprüft, ob die Eingabe zwischen fünf und zehn liegt.

> Der Gesamtausdruck einer Und-Verknüpfung wird wahr, wenn beide Teilausdrücke wahr sind.

Oder-Verknüpfung

Neben der Und-Verknüpfung gibt es die Oder-Verknüpfung. Die Oder-Verknüpfung wird in C++ durch zwei senkrechte Striche (||) dargestellt. Der Gesamtausdruck ist wahr, wenn nur eine der beiden verknüpften Bedingungen wahr ist. Es dürfen allerdings auch beide wahr sein. Das Oder ist also kein »Entweder-Oder«.

> Der Gesamtausdruck einer Oder-Verknüpfung wird wahr, wenn mindestens einer der Teilausdrücke wahr ist.

Negation

Um das Gegenteil einer Bedingung ausdrücken zu wollen, können Sie sie einfach einklammern und ein Ausrufezeichen (!) davor stellen. Die Negation eines logischen Ausdrucks wird in C++ durch das Ausrufezeichen

dargestellt. Wollen Sie also ausdrücken, dass die Eingabe nicht zwischen 5 und 10 liegen soll, schreiben Sie dies einfach so:

Listing 1.4 Programmausschnitt

```
if (!(Eingabe>=5 && Eingabe<=10))
```

Die Klammer hinter dem Ausrufezeichen bewirkt, dass der gesamte Ausdruck negiert wird und nicht nur der erste. Sie können also Klammern auch in logischen Ausdrücken verwenden, um Prioritäten zu setzen.

Näheres zu den Verknüpfungen von Bedingungen finden Sie ab Seite 125. Zusammenfassend zeigt Tabelle 1.3 die logischen Operatoren.

Operator	Bedeutung
==	Gleichheit
!=	Ungleichheit
<	Kleiner
<=	Kleiner oder gleich
>	Größer
>=	Größer oder gleich
!	Negation eines booleschen Ausdrucks (NOT)
&&	Und-Verknüpfung (AND)
||	Oder-Verknüpfung (OR)

Tabelle 1.3 Boolesche Operatoren

1.2.2 Die while-Schleife

In Schleifen kann eine Anweisung oder ein Block von Anweisungen so lange wiederholt werden, wie die Schleifenbedingung zutrifft.

Listing 1.5 Zahleneingabe

```
#include <iostream.h>

main()
{
    int InZahl = 0;
    while (InZahl<1 || InZahl>6)
    {
        cout << "Geben Sie eine Zahl von 1 bis 6 ein!";
```

```
        cin >> InZahl;
    }
}
```

Schleifenbedingung

Das Programm wiederholt so lange die while-Schleife, wie die Variable **InZahl** kleiner als 1 oder größer als 6 ist. Anders ausgedrückt: Das Programm verlässt die Schleife, wenn der Wert von **InZahl** zwischen 1 und 6 liegt.[2] Es handelt sich also um eine Prüfung der Benutzereingabe. Wie bei der Abfrage steht auch hier die Bedingung in runden Klammern direkt hinter dem Kommando while.

Schleifenkörper

Innerhalb der Schleife wird die Anweisung gegeben, eine Zahl zwischen 1 und 6 einzugeben. Daraufhin kann der Benutzer eine Zahl eingeben, die in der Variablen **InZahl** gespeichert wird. Diese beiden Anweisungen werden so lange wiederholt, wie die Bedingung zutrifft. Damit ist sichergestellt, dass innerhalb der Schleife ein Ereignis eintreffen kann, das dafür sorgt, dass die Schleife wieder verlassen wird. Ansonsten befindet sich das Programm in einer Endlosschleife – ein beliebter Fehler bei jung und alt.

Initialisierung

Wichtig ist auch, was vor der Schleife passiert. Durch die Zuweisung von 0 an die Variable **InZahl** ist gewährleistet, dass die Schleife überhaupt betreten wird. Sollte die Variable **InZahl** bereits vor der Schleife einen Wert zwischen 1 und 6 haben, dann würde die Schleife gar nicht erst betreten.

Nähere Informationen zur while-Schleife finden Sie ab Seite 129.

1.2.3 Die for-Schleife

Die Schleife for ist eine Sonderform der while-Schleife. Die Initialisierung, die Bedingung für das Verbleiben in der Schleife und die Anweisung zum Erreichen des nächsten Schritts befinden sich im Kopf der Schleife. Dadurch wird erreicht, dass man keinen der wichtigen Aspekte einer Schleife vergisst. Die for-Schleife ist etwas abstrakter als die while-Schleife, aber dafür etwas übersichtlicher, wenn man sich an sie gewöhnt hat. Die for-Schleife wird besonders dann verwendet, wenn Dinge abgezählt werden.

Das folgende Programm erstellt eine Liste der ersten zehn Quadratzahlen auf dem Bildschirm.

2 Zu dieser Umformung siehe das Gesetz von De Morgan ab Seite 127.

Listing 1.6 Quadratzahlen

```
#include <iostream.h>
main()
{
    int i;
    for (i=1; i<=10; i++)
    {
        cout << i << ": " << i*i << endl;
    }
}
```

Wenn Sie das Programm starten, erscheinen in der linken Spalte die Zahlen von 1 bis 10. Darauf folgt in jeder Zeile ein Doppelpunkt, ein Leerzeichen und dann die Quadratzahl.

Klammerinhalt

In der `for`-Klammer stehen drei Elemente, jeweils durch ein Semikolon getrennt. Das erste Element ist die Initialisierungsanweisung. Sie wird genau einmal vor dem Betreten der Schleife ausgeführt. Hier wird typischerweise die Zählvariable auf ihren Startwert gesetzt. In der Mitte zwischen den Semikola steht die Schleifenbedingung. Das ist die Bedingung, unter der die Schleife weiterhin durchlaufen wird. Bei Zählungen wird hier überprüft, ob die Zählvariable noch unter dem Limit liegt. Das dritte Element ist die Schlussanweisung. Sie wird in jedem Durchgang nach Ausführung des Schleifenrumpfes ausgeführt. Typischerweise sorgt diese Anweisung dafür, dass Variablen sich so verändern, dass irgendwann die Schleifenbedingung fehlschlägt und die Schleife verlassen wird. Beispielsweise wird hier die Zählvariable erhöht.

Nähere Informationen zur `for`-Schleife finden Sie ab Seite 134.

1.3 Arrays

Ein Array ist ein Datenverbund mehrerer gleichartiger Variablen. Dabei stehen die Variablen in einer Reihe direkt nebeneinander. Das ganze Array hat einen Namen. Um ein einzelnes Element eines Arrays anzusprechen, wird die Positionsnummer verwendet.

Ein typisches Beispiel für ein Array sind die Lottozahlen. Es handelt sich um sechs völlig gleichwertige Zahlen, die nur zusammen interessant sind. Sie können in einer Array-Variablen namens **lotto** abgelegt werden. Diese Variable muss Platz für sechs ganze Zahlen haben. Um eine einzelne Zahl anzusprechen, wird der Variablenname **lotto** genannt und danach in eckigen Klammern die Position, an der die einzelne Zahl steht. Also

2 Einstieg in die Programmierung

Programmieren ist wie Basteln ohne Späne.

2.1 Programmieren

Sie wollen programmieren lernen. Wahrscheinlich haben Sie schon eine klare Vorstellung davon, was ein Programm ist. Ihre Textverarbeitung beispielsweise ist ein Programm. Sie können einen Text eintippen. Das Programm richtet ihn nach Ihren Wünschen aus, und anschließend können Sie das Ergebnis auf Ihrer Festplatte speichern oder auf Papier ausdrucken.

Damit ist der typische Ablauf eines Programms bereits umschrieben. Das Programm nimmt Eingaben des Anwenders entgegen, verarbeitet sie nach Verfahren, die vom Programmierer vorgegeben wurden, und gibt die Ergebnisse aus. Damit wird deutlich, dass sich der Anwender in den Grenzen bewegt, die der Programmierer vorgibt. Wenn Sie selbst programmieren lernen, übernehmen Sie die Kontrolle über Ihren Computer. Während Sie bisher nur das mit dem Computer machen konnten, was die gekaufte Software zuließ, können Sie als Programmierer frei gestalten, was der Computer tun soll.

2.1.1 Start eines Programms

Maschinensprache

Ein Programm ist zunächst eine Datei, die Befehle enthält, die der Computer ausführen kann. Diese Programmdatei befindet sich typischerweise auf der Festplatte oder auf einem sonstigen Datenträger. Wenn ein Programm gestartet wird, wird es zunächst von der Festplatte in den Hauptspeicher geladen. Nur dort kann es gestartet werden. Ein gestartetes Programm nennt man Prozess. Es wird vom Prozessor, der auch CPU[1] genannt wird, abgearbeitet. Der Prozessor besitzt einen Programmzeiger, der auf die Stelle zeigt, die als Nächstes bearbeitet wird. Beim Starten des Prozesses wird dieser Zeiger auf den ersten Befehl des Programms gesetzt. Jeder Befehl weist den Prozessor an, Werte aus dem Speicher zu lesen, zu schreiben oder zu berechnen. Der Prozessor kann Werte in Speicherstellen vergleichen und kann in Abhängigkeit davon mit der Abarbeitung an einer anderen Stelle fortfahren. Die Befehle, die vom Prozessor derart interpretiert werden, sind der Befehlssatz der Maschinensprache dieses Prozessors.

1 CPU: Central Processing Unit

Der Hauptspeicher enthält also sowohl die Daten als auch die Programme. Das ist der Speicher, der unter Gedächtnisschwund leidet, wenn es ihm an Strom fehlt. Er besteht aus Speicherzellen einer typbedingten Größe, die eigentlich nur ganze Zahlen aufnehmen können. Entsprechend bestehen die Befehle, die der Prozessor direkt interpretieren kann, nur aus Zahlen. Während Prozessoren solche Zahlenkolonnen lieben, ist es nicht jedermanns Sache, solche Programme zu schreiben; schließlich ist es sehr abstrakt und damit fehleranfällig. Die Programme laufen auch nur auf dem Prozessor, für den sie geschrieben sind, da die verschiedenen Prozessoren unterschiedliche Befehlssätze haben.

Assembler

Wenn tatsächlich Software in Maschinensprache entwickelt werden soll, dann verwendet man als Hilfmittel eine Sprache namens Assembler. Diese Sprache lässt sich 1:1 in Maschinensprache übersetzen, ist aber für den Menschen leichter lesbar. So entspricht der Sprungbefehl einer 6502-CPU der Zahl 76. Der passende Assembler-Befehl lautet JMP. Ein Übersetzungsprogramm, das ebenfalls Assembler genannt wird, erzeugt aus den für Menschen lesbaren Befehlen Maschinensprache.

Einsatz

Heutzutage wird Maschinensprache und Assembler nur noch sehr selten eingesetzt, da die Entwicklungszeit hoch ist und die Programme nur auf dem Computertyp laufen, für den sie geschrieben sind. Der Vorteil von Assembler-Programmen ist, dass sie extrem schnell laufen. Doch das wird durch hohe Entwicklungskosten erkauft. So wird Assembler heutzutage fast nur noch eingesetzt, wenn es darum geht, Betriebssystembestandteile zu schreiben, die durch Hochsprachen nicht abzudecken sind. Dazu gehört beispielsweise die Interrupt-Behandlung.

2.1.2 Eintippen, übersetzen, ausführen

Editor

Wenn ein Programmierer ein Programm schreibt, erstellt er eine Textdatei, in der sich Befehle befinden, die sich an die Regeln der von ihm verwendeten Programmiersprache halten. Die heutigen Programmiersprachen bestehen in erster Linie aus englischen Wörtern und einigen Sonderzeichen. Der Programmtext wird mit Hilfe eines Editors eingetippt. Ein Editor ist eine Textverarbeitung, die nicht mit dem Ziel entwickelt wurde, Text zu gestalten, sondern um effizient Programme schreiben zu können. Der Programmtext, den man als Source oder Quelltext bezeichnet, darf nur reinen Text enthalten, damit er einwandfrei weiterverarbeitet werden kann. Sämtliche Formatierungsinformationen stören dabei nur. Darum können Sie auch kein Word-Dokument als Quelltext verwenden. Wenn Sie Ihre ersten Programmtexte unbedingt in Word schreiben möchten, müssen

Sie darauf achten, dass der Quelltext als reiner Text abgespeichert wird. Zum Programmieren gibt es allerdings wesentlich geeignetere Editoren als Word. Bei den integrierten Compilern ist ein solch spezialisierter Editor bereits enthalten. Die Mindestanforderung an einen Programm-Editor ist, dass er Zeilennummern anzeigt, da die Compiler die Position des Fehlers in Form von Zeilennummern angeben. Im Anhang finden Sie weitere Hinweise dazu beim Thema Editoren (siehe Seite 441).

Compiler

Die von den Programmierern erstellten Quelltexte werden vom Computer nicht direkt verstanden. Wie schon erwähnt, versteht er nur die Maschinensprache. Aus diesem Grund müssen die Quelltexte übersetzt werden. Dazu wird ein Übersetzungsprogramm gestartet, das man Compiler nennt. Der Compiler erzeugt aus den Befehlen im Quelltext die Maschinensprache des Prozessors. Aus jeder Quelltextdatei erzeugt er eine so genannte Objektdatei.

Präprozessor

Im Falle von C und C++ besitzt der Compiler noch eine Vorstufe, die Präprozessor genannt wird. Der Präprozessor bereitet den Quelltext auf, bevor der eigentliche Compiler ihn in Maschinensprache übersetzt. Er kann textuelle Ersetzungen durchführen, Dateien einbinden und nach bestimmten Bedingungen Quelltextpassagen von der Übersetzung ausschließen. Sie erkennen Befehle, die an den Präprozessor gerichtet sind, an einem vorangestellten #.

Linker

In der Praxis besteht ein Programmierprojekt normalerweise aus mehreren Quelltextdateien. Diese werden durch den Compiler zu Objektdateien übersetzt. Anschließend werden sie vom Linker zusammengebunden. Hinzu kommt, dass ein Programm nicht nur aus dem Code besteht, den der Programmierer selbst schreibt, sondern auch Standardroutinen wie Bildschirmausgaben enthält, die immer wieder gebraucht werden und nicht immer neu geschrieben werden müssen. Diese Teile liegen dem Compiler-Paket als vorübersetzte Objektdateien bei und sind zu Bibliotheken zusammengefasst. Eine solche Bibliothek wird auch Library genannt. Der Linker entnimmt den Bibliotheken die vom Programm benötigten Routinen und bindet sie mit den neuen Objektdateien zusammen. Das Ergebnis ist ein vom Betriebssystem ausführbares Programm.

Der typische Alltag eines Programmierers besteht darin, Programme einzutippen und dann einen Übersetzer zu starten, der den Text in Prozessor-Code übersetzt. Nach dem Binden wird das Programm gestartet und getestet, ob es die Anforderungen erfüllt. Danach wendet sich der Programmierer wieder den Quelltexten zu, um die gefundenen Fehler zu korrigieren.

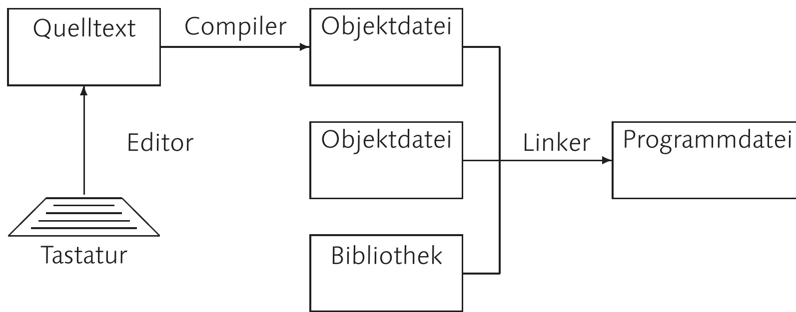


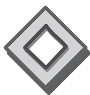
Abbildung 2.1 Entwicklungsweg eines Programms

Planung Es sollte nicht unerwähnt bleiben, dass einige Programmierer dazu neigen, vor dem Eintippen zu überlegen. Ja, manche Entwickler machen richtige Entwürfe dessen, was sie programmieren wollen. Sie behaupten, dass sie dann letztlich schneller zum Ziel kommen. Und im Allgemeinen haben sie sogar Recht.

2.1.3 Der Algorithmus

Nachdem der Weg beschrieben ist, wie Sie vom Programmquelltext zu einem ausführbaren Programm kommen, sollte die Frage beantwortet werden, wie Programme inhaltlich erstellt werden.

Eine Programmiersprache ist eine sehr formale Beschreibung eines Ablaufs. Der Computer hat keine Phantasie und keine Erfahrungen. Darum kann er Ihren Anweisungen nur dann korrekt folgen, wenn Sie ihm zwingend vorschreiben, was er tun soll, und dabei keine Missverständnisse zulassen. Alle Voraussetzungen, die das Verfahren benötigt, müssen ausformuliert werden. Ein solches Verfahren nennt man einen Algorithmus. Algorithmen werden gern mit Kochrezepten verglichen. Die Analogie passt auch ganz gut, sofern es sich um ein Kochbuch für Informatiker handelt, das davon ausgeht, dass der Leser des Rezepts eventuell nicht weiß, dass man zum Kochen oft Wasser in den Topf füllt und dass zum Braten gern Öl oder anderes Fett verwendet wird.



Wer das erste Mal einen Algorithmus entwirft, stellt dabei fest, wie schwierig es ist, ein Verfahren so zu beschreiben, dass der Ausführende zwingend zu einem bestimmten Ergebnis kommt. Als kleine Übung sollten Sie jetzt kurz die Lektüre unterbrechen und beschreiben, wie Sie zählen. Der besseren Kontrolle halber sollen die Zahlen von 1 bis 10 auf einen Zettel geschrieben werden.

Fertig? Prima! Dann sollte auf Ihrem Zettel etwa Folgendes stehen:

1. Schreibe die Zahl 1 auf den Zettel.
2. Lies die Zahl, die zuunterst auf dem Zettel steht.
3. Wenn diese Zahl 10 ist, höre auf.
4. Zähle zu dieser Zahl 1 hinzu.
5. Schreibe die Zahl auf den Zettel unter die letzte Zahl.
6. Fahre mit Schritt 2 fort.

Sie können sich darin üben, indem Sie alltägliche Abläufe formalisiert beschreiben. Notieren Sie beispielsweise, was Sie tun, um sich anzukleiden oder um mit dem Auto zur Arbeit zu fahren. Stellen Sie sich dann vor, jemand führt die Anweisungen so aus, dass er jeden Freiraum zur Interpretation nutzt, um den Algorithmus zum Scheitern zu bringen. Testen Sie es mit Verwandten oder wirklich guten Freunden aus: Die kennen in dieser Hinsicht erfahrungsgemäß die geringsten Hemmungen.

2.1.4 Die Sprache C++

Offensichtlich ist Ihre Entscheidung für C++ bereits gefallen. Sonst würden Sie dieses Buch ja nicht lesen. Je nachdem, ob Sie C++ aus eigenem Antrieb lernen oder aufgrund eines zarten Hinweises Ihres Arbeitgebers, werden Sie mehr oder weniger über C++ als Sprache wissen. Darum erlaube ich mir hier ein paar Bemerkungen.

Compiler-Sprache

Compiler

C++ ist eine Compiler-Sprache. Das bedeutet, dass die vom Programmierer erstellten Programmtexte vor dem Start des Programms durch den Compiler in die Maschinensprache des Prozessors übersetzt werden, wie das in Abbildung 2.1 dargestellt ist.

Interpreter

Einige andere Programmiersprachen wie beispielsweise BASIC oder die Skriptsprachen werden während des Programmablaufs interpretiert. Das bedeutet, dass ein Interpreter den Programmtext lädt und Schritt für Schritt übersetzt und ausführt. Wird eine Zeile mehrfach ausgeführt, muss sie auch mehrfach übersetzt werden. Da die Übersetzung während der Laufzeit stattfindet, sind Programme, die in Interpretersprachen geschrieben wurden, deutlich langsamer.

Virtuelle Maschinen

Daneben gibt es noch Mischformen. Der bekannteste Vertreter dürfte Java sein. Hier wird der Quelltext zwar auch vor dem Ablauf übersetzt. Allerdings ist das Ergebnis der Übersetzung nicht die Maschinensprache

des Prozessors, sondern eine Zwischensprache, die sehr viel schneller interpretiert werden kann als die ursprünglichen Quelltexte.

Jede dieser Vorgehensweisen hat Vor- und Nachteile. Der Vorteil einer Compiler-Sprache liegt zum einen in ihrer Geschwindigkeit. Der Programmtext wird genau einmal übersetzt, ganz gleich wie oft er durchlaufen wird, und die Übersetzung wird nicht zur Laufzeit des Programms durchgeführt. Ein weiterer Vorteil besteht zum anderen darin, dass viele Fehler bereits beim Kompilieren entdeckt werden. Interpreter haben den Vorteil, dass ein Programmierfehler nicht sofort das Programm abstürzen lässt. Der Interpreter kann stoppen, den Programmierer auf den Ernst der Lage hinweisen und um Vorschläge bitten, wie die Situation noch zu retten ist. Gerade Anfänger schätzen diese Vorgehensweise sehr, weil sie Schritt für Schritt ihr Programm ausführen lassen und sehen können, was der Computer tut. In Compiler-Sprachen können Sie das auch, benötigen dazu allerdings die Hilfe eines Debuggers (siehe Seite 443). Die Sprachen, die mit virtuellen Maschinen arbeiten, bieten die Möglichkeit, ein fertig übersetztes Programm auf verschiedensten Computern laufen zu lassen. Da die Programme vorübersetzt sind, sind sie deutlich schneller als reine Interpreter-Sprachen.

Wurzeln in C

Portabler Assembler

Bjarne Stroustrup hat C++ aus der Programmiersprache C weiterentwickelt. C wurde in den 70er Jahren von Kernighan und Ritchie im Zusammenhang mit der Entwicklung von UNIX entwickelt. Bis dahin waren Betriebssysteme in Assembler geschrieben worden. Dadurch waren die Betriebssysteme auf das Engste mit den Prozessoren und den anderen Hardware-Gegebenheiten ihrer Entwicklungsmaschinen verbunden. Für die Entwicklung von UNIX wurde eine Sprache geschaffen, die in erster Linie ein »portabler Assembler« sein sollte. Der erzeugte Code musste schnell und kompakt sein. Es war erforderlich, dass auch maschinennahe Aufgaben gelöst werden konnten. Auf diese Weise mussten nur kleine Teile von UNIX in Assembler realisiert werden, und das Betriebssystem konnte leicht auf andere Hardware-Architekturen umgesetzt werden. Bei dem Entwurf der Sprache C wurden die Aspekte der seinerzeit modernen Programmiersprachen berücksichtigt. So konnten in C die damals entwickelten Grundsätze der strukturierten und prozeduralen Programmierung umgesetzt werden.

ANSI C

Im ursprünglichen Design der Sprache gab es noch einige unschöne Dinge, die sich mit der Zeit als lästig erwiesen. Sie wurden endgültig abge-

streift, als C zum ANSI-Standard wurde. Dabei kamen noch ein paar Elemente hinzu, so dass sich ANSI-C vom Kernighan-Ritchie-Standard, auch K&R genannt, in einzelnen Punkten entfernte und nicht mehr in allen Punkten kompatibel war. Dieser Prozess ist allerdings bereits seit vielen Jahren abgeschlossen. Inzwischen gibt es kaum noch K&R-Programme.

Standardsprache

Die Sprache C entwickelte sich im Laufe der 80er und 90er Jahre zum Standard. Seit dieser Zeit wurden immer mehr Betriebssysteme in C geschrieben. Die meisten APIs[2] sind in C formuliert und in dieser Sprache auf dem direktesten Wege zu erreichen. Aber auch in der Anwendungsprogrammierung wurde vermehrt C eingesetzt. Immerhin hatte C in den Betriebssystemen bewiesen, dass es sich als Entwicklungssprache für große Projekte eignete.

Objektorientierte Programmierung

Große Projekte erfordern, dass das Schreiben des Programms auf mehrere Teams verteilt wird. Damit ergeben sich automatisch Abgrenzungsschwierigkeiten. Projekte, an denen viele Programmierer arbeiten, müssen so aufgeteilt werden können, dass jede kleine Gruppe einen klaren eigenen Auftrag bekommt. Zum Schluss wäre es ideal, wenn man die fertigen Teillösungen einfach nur noch zusammenstecken müsste. Die Notwendigkeit von Absprachen sollte möglichst reduziert werden, da einer alten Regel zufolge die Dauer einer Konferenz mit dem Quadrat ihrer Teilnehmer ansteigt.

Um die Probleme großer Projekte in den Griff zu bekommen, gibt es unterschiedliche Ansätze. Stichworte sind hier die strukturierte Programmierung, die Modularisierung und das Geheimnisprinzip. Die letztgenannten Ansätze sorgen für eine Zergliederung großer Projekte in Module dergestalt, dass jedes Modul nur so viel veröffentlicht, wie zu ihrer Benutzung erforderlich ist. Damit werden die notwendigen Absprachen reduziert.

Darauf aufbauend werden auch in der objektorientierten Programmierung Module gebildet. Dazu kommt allerdings der Grundgedanke, Datenobjekte in den Mittelpunkt des Denkens zu stellen und damit auch als Grundlage für die Zergliederung von Projekten zu verwenden. Lange Zeit waren Programmierer auf die Algorithmen fixiert, also auf die Verfahren, mit denen Probleme gelöst werden können. Mit der objektorientierten Programmierung dreht sich der Blickwinkel. Statt des Verfahrens werden

2 API ist die Abkürzung für Application Programming Interface und bezeichnet die Programmierschnittstelle vom Anwendungsprogramm zur Systemumgebung.

die Daten als Dreh- und Angelpunkt eines Programms betrachtet. Die Erkenntnis, dass jedes Verfahren durch das Objekt bestimmt wird, auf das es angewandt wird, ist eigentlich nahe liegend. So werden in der objektorientierten Programmierung die Funktionalitäten an die Datenstrukturen gebunden. Dadurch ergibt sich eine sehr organische Möglichkeit, große Projekte aufzuteilen.

Die Unterstützung der objektorientierten Programmierung durch die Einführung der Klassen ist der wichtigste Teilaspekt, der C++ von C abhebt. Gleichzeitig erlebte die objektorientierte Programmierung mit der Einführung von C++ ihre große Popularität. Heutzutage darf sich kaum noch eine Programmiersprache auf die Straße trauen, wenn sie nicht objektorientiert ist.

C++ und die anderen Sprachen

Hybrid Keine Frage, objektorientierte Programmierung ist in. C++ hat die objektorientierte Programmierung zwar nicht eingeführt, aber zumindest populär gemacht. Puristen der objektorientierten Programmierung werfen C++ vor, dass es eine Hybrid-Sprache ist. Das bedeutet, dass C++ nicht nur objektorientierte Programmierung zulässt, sondern es auch erlaubt, in klassischem C-Stil zu programmieren. Das ist aber durchaus so gewollt. C++ unterstützt die objektorientierte Programmierung, erzwingt sie aber nicht. Bjarne Stroustrup sah dies wohl eher als Vorteil. Er wollte eine Programmiersprache schaffen, die den Programmierer darin unterstützt, seiner Strategie zu folgen. Dieser sehr pragmatische Ansatz ist vielleicht der Grund, warum C++ trotz Java und C# in der professionellen Entwicklung noch immer eine wesentliche Rolle spielt. Mit C++ ist es einem Anfänger möglich, kleine Programme zu schreiben, ohne zuerst die objektorierte Programmierung verstehen zu müssen. Und gerade damit hat C++ den Erfolg der objektorientierten Programmierung vermutlich erst möglich gemacht. Denn so können Programmierer in kleinen Schritten die Vorteile der objektorientierten Programmierung kennen lernen und nutzen.

Effizienz Ein anderer Vorteil von C++ ist seine Geschwindigkeit. Stroustrup legte großen Wert darauf, dass die Schnelligkeit und Kompaktheit von C erhalten bleibt. Damit bleibt C++ die Sprache erster Wahl, wenn es um zeitkritische Programme geht. Aber selbst wenn das Programm nicht in Echtzeit ablaufen muss, wird sich der Anwender freuen, wenn sein Programm möglichst schnell ist. Wer eine langsamere Sprache verwendet, muss dieses Defizit dem Anwender gegenüber gut begründen können.

Zuletzt hat die nach wie vor stabile Position von C++ auch etwas mit Marktstrategien zu tun. Java wurde entwickelt, um einmal geschriebenen Code auf allen Plattformen ohne Neukompilierung laufen lassen zu können. Dies konnte Microsoft nicht recht sein. So versucht der Konzern seit Jahren, diese Sprache zu unterlaufen. Der letzte Schlag ist die Entwicklung einer eigenen Konkurrenzsprache namens C#. So haben Projektleiter die Entscheidung, ob sie die Sprache Java verwenden, die auf allen Plattformen läuft, aber von Windows irgendwann nicht mehr unterstützt werden wird. Oder sie verwenden die Sprache C#, die von Windows intensiv unterstützt wird, aber von allen anderen Plattformen nicht. Der lachende Dritte scheint C++ zu sein. Jedenfalls nehmen die Forderungen nach C++-Kenntnissen bei Freiberuflern in den letzten Jahren wieder zu.[3]

Der Markt

2.1.5 Fragen zur Selbstkontrolle

- Schreiben Sie einen Algorithmus für das Kochen von Kaffee. Bitten Sie einen Freund diesen Anweisungen folgen, die Sie durch die geschlossene Tür geben. Welche Anweisungen führten zu Fehlern? Welche Anweisungen waren fehlinterpretierbar? Würden Sie den Algorithmus anders schreiben, wenn der Freund nicht wüsste, was Kaffee ist und wofür man ihn braucht (den Kaffee, nicht den Freund)?
- Warum können Sie kein noch so einfaches Programm für den Mac auf einem Windows-PC starten?
- Warum ist ein Programm, das mit einem Interpreter übersetzt wird, typischerweise langsamer als eines, das vom Compiler übersetzt wurde?

Die Antworten zu den letzten beiden Fragen finden Sie auf Seite 467.

2.2 Grundgerüst eines Programms

Ein Programm ist eine Aneinanderreihung von Befehlen, die dem Computer zur Abarbeitung zugeführt werden können. Wie schon erwähnt, wird ein C++-Programm in einem Texteditor geschrieben und dann von einem Compiler[4] in ausführbaren Code übersetzt. Danach kann es gestartet werden.

main()

Ein C++-Programm besteht mindestens aus der Funktion `main()`. Die Silbe »main« ist englisch und bedeutet so viel wie die deutsche Vorsil-

3 Sie finden auf den Seiten von Internetbörsen für Freiberufler wie beispielsweise http://www.gulp.de häufig Statistiken über die geforderten Sprachkenntnisse.

4 In diesem Fall ist mit dem Compiler der komplette Übersetzer gemeint, der aus Compiler und Linker besteht. Diese Ungenauigkeit ist im allgemeinen Sprachgebrauch durchaus üblich.

be »Haupt«. Was in C++ genau unter einer Funktion zu verstehen ist, wird später noch sehr viel ausführlicher behandelt und soll Sie hier nicht verwirren. Wichtig ist, dass Sie in jedem C++-Programm irgendwo den Namen **main** mit einem Klammerpaar finden werden. In manchen Fällen stehen zwischen den Klammern ein paar kryptisch anmutende Zeichen. Aber auch das sollte Sie zu Anfang nicht stören. Ignorieren Sie es von ganzem Herzen, bis Sie wissen, wozu Sie es brauchen.

Die Hauptfunktion beginnt direkt nach einer öffnenden geschweiften Klammer und endet mit der schließenden Klammer. Zwischen den geschweiften Klammern stehen die Befehle der Hauptfunktion. Sie werden nacheinander ausgeführt. Das folgende kurze Listing[5] enthält nichts und ist damit das kleinste denkbare C++ Programm.

Listing 2.1 Minimalprogramm

```
main()
{
}
```

Dieses Programm tut gar nichts. Wenn es etwas täte, stünden die Befehle zwischen den geschweiften Klammern.

2.2.1 Kommentare

Wenn wir schon einmal ein Programm haben, das nichts tut, wollen wir es auch um Befehle erweitern, die nichts tun: Kommentare.

Freundlichkeit unter Kollegen

Sie können in das Programm Kommentare einfügen, die vom Compiler völlig ignoriert werden. Der Zweck solcher Kommentare ist es zu dokumentieren, warum das Programm so und nicht anders geschrieben wurde. Die Kommentare richten sich an Programmierer. Dies können Kollegen sein, die Ihr Programm korrigieren oder erweitern sollen. Aber noch öfter helfen Sie sich selbst damit. In der Praxis ist es so, dass Sie vermutlich auch herangezogen werden, wenn eines Ihrer Programme nicht korrekt läuft oder ergänzt werden soll. Da Sie zwischendurch andere Programme geschrieben haben werden, vielleicht geheiratet haben, umgezogen sind und noch drei weitere Programmiersprachen gelernt haben, werden Sie sich nicht mehr an jedes Detail erinnern. Sie werden dankbar sein, wenn Sie in den Programmen ausführliche Hinweise finden, wie und warum es so funktioniert oder zumindest funktionieren soll.

5 Als Listing bezeichnet man den Quelltext eines Programms.

Es gibt zwei Arten, einen Text davor zu schützen, dass der Compiler versucht, ihn als Programm zu interpretieren. Die einfachere Art der Kommentierung ist es, zwei Schrägstriche direkt hintereinander zu setzen. Damit gilt der Rest der Zeile als Kommentar.

Listing 2.2 Zeilenweises Kommentieren

```
main()
{
    // Hier beginnt der Kommentar.
    // Die naechste Zeile braucht ihr
    // eigenes Kommentarzeichen.
}
```

Kommentarblock

Daneben gibt es die Möglichkeit, einen größeren Text in Kommentarklammern einzuschließen und damit dem Compiler zu entziehen. Der Anfang des Kommentars wird durch den Schrägstrich, gefolgt von einem Stern, festgelegt. Der Kommentar endet mit der umgekehrten Zeichenfolge, also mit einem Stern und dann einem Schrägstrich.

Listing 2.3 Blockweises Kommentieren

```
main()
{
  /* Hier beginnt der Kommentar.
     Die naechste Zeile braucht kein
     eigenes Kommentarzeichen.
   */
}
```

Die meisten Compiler können diese Kommentarklammern nicht verschachteln. Das bedeutet, dass Sie beliebig oft den Kommentaranfang mit Schrägstrich und Stern definieren können. Das erste Ende-Zeichen aus Stern und Schrägstrich beendet den Kommentar. Der Compiler wird den folgenden Text als Programm ansehen und versuchen, ihn zu übersetzen.

Listing 2.4 Das geht schief!

```
main()
{
  /* Hier beginnt der Kommentar
     /*
       Die naechste Zeile braucht kein
```

```
        eigenes Kommentarzeichen
       */
     Dies wird der Compiler wieder uebersetzen wollen.
  */
}
```

Kurzkommentar

Mit /* und */ können nicht nur große Blöcke, sondern auch kurze Teile innerhalb einer Zeile kommentiert werden. Diese Möglichkeit haben Sie mit dem doppelten Schrägstrich nicht, weil dieser Kommentar ja immer bis zum Ende der Zeile geht. Im folgenden Beispiel ist die schließende Klammer von main() außerhalb des Kommentars.

```
main( /* hier könnte auch noch etwas stehen */ )
{
```

Inhalt der Kommentare

In vielen Programmierkursen und auch in einigen Firmen finden Sie umfangreiche Beschreibungen, die sich darüber auslassen, wo Kommentare erscheinen sollen und was sie beinhalten sollen. Der Zweck dieser Vorschriften ist, dass die Investition in die Programmierung nicht verloren geht. Wenn ein unverständlicher Programmcode später kontrolliert wird, kann es sein, dass dann Fehler eingebaut werden, nur weil der Programmierer nicht verstanden hat, was der Zweck des Programmteils war. Die einfachste Regel in der Kommentierung lautet darum:

> Kommentieren Sie nicht, *was* Sie gemacht haben, sondern *warum* Sie es so gemacht haben!

Ein Kommentar »Zwei Werte werden addiert« ist nutzlos. Denn jeder Programmierer, der die Sprache C++ auch nur halbwegs kennt, kann das direkt im Quelltext ablesen. Dagegen ist die Aussage »ermittle den Bruttobetrag aus Nettobetrag und MWSt« hilfreich, da dadurch jeder Leser weiß, warum diese Werte addiert werden.

2.2.2 Anweisungen

Ein Programm besteht nicht nur aus dem Rahmen und Kommentaren, sondern auch aus Anweisungen. Anweisungen sind die Befehle, aus denen ein Programm besteht. Eine Anweisung kann dazu dienen, etwas zu speichern, zu berechnen, zu definieren oder auf dem Bildschirm auszugeben. Anweisungen sind die Grundeinheiten, aus denen Programme bestehen.

Anweisungen können in C++ beliebig formatiert werden. Es besteht keine Verpflichtung, sie in einer bestimmten Position anzuordnen. Auch können Anweisungen über beliebig viele Zeilen gehen. Allerdings besteht der Compiler extrem kleinlich darauf, dass Anweisungen immer mit einem Semikolon abgeschlossen werden.

2.2.3 Blöcke

Zusammenfassung

Mehrere Anweisungen können zu einem Block zusammengefasst werden. Ein Block wird durch geschweifte Klammern eingefasst. Der Compiler wird einen Block wie eine einzige Anweisung behandeln.

Sicher ist Ihnen schon aufgefallen, dass die Hauptfunktion `main()` ebenfalls solche geschweiften Klammern hat. In der Tat ist dies der Block, in dem die Befehle des Programms zusammengefasst werden.

Konventionen

Um die Übersicht zu erhöhen, pflegt man alle Befehle innerhalb eines Blocks einzurücken. Achten Sie vor allem am Anfang darauf, dass die zusammengehörigen geschweiften Klammern auf einer Ebene stehen. Das folgende Beispiel mit Pseudobefehlen zeigt, wie das Einrücken korrekt ist.

Listing 2.5 Eingerückte Blöcke

```
main()
{
    Hier sind Pseudo-Anweisungen;
    Diese gehört dazu;
    {
        Neuer Block, also einrücken;
        Wir bleiben auf diesem Niveau;
        Das kann ewig weitergehen;
    }
    Nun ist die Klammer geschlossen;
    Und die Einrückung ist auch zurück;
}
```

Einrückweite

Wie weit Sie einrücken, ist Geschmacksache. Aus Erfahrung kann man sagen, dass eine Einrückung von einem Zeichen schwer erkennbar ist. Zwei Leerschritte sind darum das absolute Minimum. Drei oder vier Schritte sind weit verbreitet. Auch wenn viele Editoren acht Schritte für den Tabulator verwenden, wird der Code dadurch oft unübersichtlich, weil er zu weit nach rechts herausragt.

Index

() Aufrufoperator 266
+ 98, 99
  Operator überladen 253
++ 101, 256
+= 101
, 136
-> 231
// 68
/* . . . */ 69
= 98
== 113, 124
  Operator überladen 261
? 121
[] 149
  überladen 264
61
#define 97, 304, 411, 436, 445
#else 436
#endif 436
#if 436
#ifdef 436
#ifndef 411
#include 20, 328
% 99
<< 107, 262, 343, 353, 433
>> 108, 344, 433

A

abs() 371
Absolutbetrag 371
Abstrakte Basisklasse 290
acos() 370
Adresse 165
Algorithmus 62
AND 125
  bitweise 432
AnsiString 336
any() 403
Arcus Cosinus 370
Arcus Sinus 370
Arcus Tangens 370
argc 199
argv 199
Array 149, 167
  als Funktionsparameter 193
  kopieren 157
  mehrdimensional 162
ASCII 94
asctime() 405
asin() 370
Assembler 60
at() 384
atan() 370
atan2() 370
atof() 339
atoi() 339
atol() 339
Aufrufoperator 266
Aufzählungstyp 181
Ausgabe
  formatierte 108
  Manipulator 108

B

back() 386
bad() 360
BASIC 63
Basisklasse 272
Bedingte Kompilation 436
begin() 384, 385
Bermuda 163
  Schiffe 174
  Spielfeld 163
  Top-Down 205
Betragsfunktion 371
Bibliotheken 61, 327
  dynamisch 414
binary_search() 385
Bit 78
Bit-Operatoren 431
Bit-Struktur 435
Bitoperation
  AND 432
  OR 433
  XOR 433
Bitset 403
Block 71
Bogenmaß 370
bool 122

boolalpha 347
Borland 457
break 137
Breakpoint 443
Bubblesort 154
Buchstaben
  Konstante 94
Byte 78

C

C++-Builder 457
C-Header 328
C-Strings 336
capacity() 384
case 117
cast 105
catch 310
ceil() 372
cerr 344
char 21, 82
  unsigned 84
cin 22, 108, 344
class 230, 298
clear() 360
clock() 407
close() 353
closedir() 365
Compiler 61, 63, 442
  Optionen 444
const 97, 269
  Funktion 263
  Referenzparameter 195
  Zeiger 170
Container 381
  map 400
  set 398
continue 137, 138
copy() (STL) 375
Copy-Konstruktor 247
cos() 370
cosh() 370
Cosinus 370
cosinus hyperbolicus 370
count() 403
cout 22, 106, 343
CString 336
ctime() 405

D

dangling else 116
Datei
  Blockweise lesen 355
  End Of File 359, 363
  Fehler 358
  Lesen 353
  Öffnen 351
  Position ändern 357
  Puffer leeren 358
  Schließen 353
  Schreiben 353
  Status ermitteln 366
  Zeilenweises Einlesen 354
Dateioperationen 350
Dateiposition 357
Debugger 187, 443
  GNU 450
default 117
Deklaration
  Funktionen 190
  Variablen 211
Dekrementieren 102, 256
delete 176
delete[] 178
deque 387
Destruktor 235
do while 132
Dokumentation 428
double 21, 85
  long 85
Dynamische Bibliotheken 414

E

Editor 60
Einrücken 71
Elementinitialisierer 236
else 114
  dangling 116
End Of File 359
end() 384, 385
endl 107, 108, 346
Endlosschleife 129
enum 181
EOF 363
eof() 359
equal() (STL) 377

erase() 391, 398
exception 318
exceptions() 322, 360
Exklusiv Oder 433
exp() 370
explicit 239
extern 328, 409

F

fabs() 371
fail() 360
Fallunterscheidung 117
false 122
fclose() 361
Fehlerklassen 315
feof() 363
fgets() 362
FIFO 388
FILE 361
fill() 377
find() 399, 401
find() (STL) 374
find_if() (STL) 377
flip() 403
float 21, 84
floor() 372
fmod() 372
fopen() 361
for 134, 168
for_each() (STL) 381
Formatierte Ausgabe 346
fprintf() 341, 363
fputs() 362
Fragezeichen 121
fread() 362
frexp() 371
friend 246, 255, 263
front() 388
fstream 350
  clear() 360
  close() 353
  eof() 359
  fail() 360
  flush() 358
  good() 358
  open() 351
  read() 355
  write() 355
Funktionen 183
  Parameter 187
  statisch 308, 412
  überladen 202, 250
Funktionsobjekt 379
Funktionsparameter
  Array 193
Funktionszeiger 226
fwrite 362

G

gdb 450
Generische Programmierung 297
getline 354
gettimeofday() 406
ggT 145
global 75
GNU-Compiler 444
GNU-Debugger 450
good() 358
goto 139
Größter gemeinsamer Teiler 145
Grundrechenarten 99

H

Handle 361
Header-Dateien 410

I

if 24, 112
ifstream 350, 352
Indexoperator
  überladen 264
Indirektionsoperator 166
Initialisierer 302
Initialisierung 73
  Klassenelemente 236
Initialisierung mit 0 153
Inkrementieren 101, 168, 256
inline 203, 233, 410
insert() 391, 398
int 21, 79
Interpreter 63
iostream 343
Iterator 331, 333, 385, 402

J

Java 63

K

Kaskadierende Zuweisung 99
Kaskadierung
  Ausgabeumleitung 107
Komma 136
Kommentare 68
Komplexe Zahlen 372
Konstante
  Klasse 302
Konstante Funktionen 263, 270
Konstante Zeiger 170
Konstanten 89
  bool 122
  Buchstaben 94
  Deklaration 97
  in Klassen 269
  Zeichenkette 96
Konstruktor 235, 280
  Kopier- 247
  Standard 239
Konvertierung
  String nach Zahl 335
  Zahl nach String 335
Konvertierungskonstruktor 238
Konvertierungsoperator 267
Kopierkonstruktor 247
  Vererbung 281
Kurzschluss 128
Kylix 457

L

L-Value 99, 157, 173
Label 139
labs() 371
left 346
Library 61
limits.h 87
Lineare Liste 178
Linker 61, 329
Linux 458
list 389
Liste 389
localtime() 405
log 371
log10() 371
Logarithmus 371
lokal 75
long 21, 79
long double 85

M

main() 21, 67
make 445
Makro 300, 304
Manipulatoren 108, 346
Mantisse 93
map
  Container 400
Maschinensprache 59
math.h 369
Mehrdimensionale Arrays 162
Mehrfachvererbung 282
memcpy() 157
Memory Leak 177
merge() 395
MFC 336
Microsoft 453
modf() 372
Modulo 99
mutable 271

N

Nachkommaformatierung 350
Namensräume 306
Namensregeln 75
new 176
NICHT 127
none() 403
NOT 127
NULL 165

O

objektbasierte Programmierung 291
Objektdatei 61
Oder 433
ODER-Verknüpfung 126
ofstream 350, 352
open() 351
opendir() 365
operator+ 253
Operatoren

überladen 251
OR 126
bitweise 433
ostream 263
out_of_range 384

P

Parameter 187
Arrays 193
Referenz 195
Variable Anzahl 201
Polymorphie 283, 316
pop_back() 386
pop_front() 388
Postfix 103, 256
pow() 371
Prädikat 377
Präfix 103, 256
Präprozessor 61, 328, 411, 436
printf() 341
priority_queue 397
private 240
Programmierung
generische 297
protected 276
Prototypen 190, 211, 409
Prozessor 59
public 240
push_back() 386
push_front() 388

Q

Qualitätssicherung 425
Quelltext 60
queue 396

R

rand() 103
read() 355
readdir() 365
Record 171
Referenz
Parameter 195
rein virtuelle Funktion 290
Rekursion 214
reserve() 384
reset() 403
resize() 384
Rest 99
return 183
reverse() (STL) 376
reverse_iterator 334
right 346

S

Schablone 297
Schlüsselwörter 77
Schleifen 129
Schnittstelle 243
Seek 357
seekg() 357
seekp() 357
set
Container 398
set() 403
setfill() 346
setw() 108, 346
Shift 433
short 21, 79
sin() 370
sinh() 370
Sinus 370
Sinus hyperbolicus 370
size() 403
sizeof 86
sort() (STL) 375, 384
Sortieren
Bubblesort 154
Source 60
Späte Bindung 285
Speicherlecks 177
splice() 393
sprintf() 341
sqrt 371
srand() 103, 104
Stack 187, 196, 444
stack (STL) 395
Standard Template Library 373
Standardkonstruktor 239
stat() 366
static 267, 308, 412
Variable 212
statische Variablen 212
std 308

Step 443
STL 373
  copy() 375
  deque 387
  equal() 377
  erase() 398
  find() 374, 399, 401
  find_if() 377
  for_each() 381
  insert() 398
  list 389
  merge() 395
  reverse() 376
  sort() 375, 384
  stack 395
  vector 382
strcat() 338
strcmp() 338
strcpy() 337
String 158, 330
  nach Zahl konvertieren 335
strlen() 338
strncpy() 337
strstr() 339
struct 171
Struktogramm 114
Struktur 171
switch 117
Syntaxgraph
  #define 305
  Basisklassenliste 294
  Binäre Zahlenkonstante 91
  Buchstabe und Ziffer 76
  case 119
  Dezimale Zahlenkonstante 89
  do-while 133
  enum 181
  Fließkommakonstante 94
  for 135
  Fragezeichen 121
  Funktion 189
  Funktionsdefinition 185
  Funktionsparameter 189
  Funktionszeiger 226
  Hexadezimale Zahlenkonstante 92
  if 112
  if-else 114
  Klassendefinition 294
  Klassenelemente 294
  Namen 76
  Namensraum 310
  namespace 310
  Oktale Zahlenkonstante 92
  struct 174
  switch 119
  Template-Funktion 299
  Typbeschreibung 88
  Variablendefinition 88
  while 130
  Zeichenkettenkonstante 96
  Zeichenkonstante 94
Systemnahe Programmierung 431

T

tan() 370
Tangens 370
Tangens hyperbolicus 370
tanh() 370
Tastatureingabe 108
tellg() 358
tellp() 358
Template 297, 410
test() 403
this 234
throw 311
time() 404
time_t 404
tm 405
Top-Down-Design 205
Trigonometrische Funktionen 369
true 122
try 310
Typ 21, 78
typedef 182
typename 298
Typkonvertiertung 105

U

Überladen
  Funktionen 202, 250
  Operatoren 251
UND 432
UND-Verknüpfung 125
UNICODE 83
union 180

unsigned 79
unsigned char 84
using 277, 308

V

va_arg 201
va_end 201
va_list 201
va_start 201
Variable 21, 72
  Definition 88
  Geltungsbereich 74
  global 75
  Initialisierung 73
  lokal 75
vector 382
Vererbung 272, 316
Verkettete Liste 178
Verzeichnis auslesen 365
virtual 285
Virtuelle Funktionen 283
Virtuelle Maschinen 63
Virtueller Destruktor 289
Visual C++ 453
void 186
void * 171
Vorgabeparameter
  Klasse 243
VTable 289

W

wchar_t 83
while 129
write() 355
Wurzel 371

X

XOR
  bitweise 433

Z

Zählschleife 134
Zahl
  nach Zeichenkette konvertieren 335
Zeichenkette 158
  Konstante 96
Zeiger 165, 167
  Funktionen 226
  Inkrementieren 168
Zeitmessung 406
Ziffern 95
Zufallsfunktionen 103
Zuweisung 98
  kaskadierend 99
Zuweisungsoperator
  Vererbung 282