eXamen.press

eXamen.press ist eine Reihe, die Theorie und Praxis aus allen Bereichen der Informatik für die Hochschulausbildung vermittelt.

For further volumes:
http://www.springer.com.series/5520

Gilbert Brands

Das C++ Kompendium

STL, Objektfabriken, Exceptions

2. Auflage

Springer

Prof. Dr. Gilbert Brands
Fachhochschule Emden/Leer
Constantiaplatz 4
26723 Emden
gilbert.brands@ewetel.net

ISSN 1614-5216
ISBN 978-3-642-04786-2 e-ISBN 978-3-642-04787-9
DOI 10.1007/978-3-642-04787-9
Springer Heidelberg Dordrecht London New York

Einbandabbildung: KuenkelLopka GmbH

Gedruckt auf säurefreiem Papier

Springer ist Teil der Fachverlagsgruppe Springer Science+Business Media (www.springer.com)

Inhaltsverzeichnis

Band I

Einführung in die Programmierung 1
1 (Statt eines) Vorwort(s) 1
2 Die ersten Schritte: Anweisungslisten 4
2.1 Die Grundregeln 4
2.2 Zur Arbeitstechnik 7
2.3 Aufbau der Anweisungsliste 7
2.4 Die Konstruktion der Anweisungen 9
3 Der Anfang 14
3.1 Arbeit mit dem Entwicklungssystem 16
3.2 Erste Erkenntnisse 17
3.3 Weitere Arbeitsschritte 18
4 Die Sprachelemente von C 21
4.1 Die Datentypen 21
4.2 Die Schnittstellendefinitionen 25
4.3 Bibliotheksfunktionen 27
4.4 Weitere Teile des Programmkode 29
4.5 Eigene Datentypen 33
5 Die Sprachelemente von C++ 35
5.1 Überladen von Funktionen 35
5.2 Überladen von Operatoren 36
5.3 Namensbereiche 36
5.4 Klassen, Konstruktor, Destruktor 38
5.5 Vererbung 40
5.6 Zeigervariablen in C++ 42
5.7 Virtuelle Vererbung 43
5.8 Mehrfachvererbung 45
5.9 Referenzen 46
5.10 Templates 46
6 Zur Arbeitsweise 48

1 Zur professionellen Arbeitsweise 51
1.1 Arbeitsphilosophie und Methodik 51
1.1.1 Die Auswahl der Programmiersprache 51

1.1.2 Anforderungen an eine Anwendung 52
1.1.3 Der Fehlerbegriff 53
1.1.4 Prüfen und Testen 56
1.1.5 Der Einfluss der Theorie 59
1.2 (Wieder-)Verwendbarkeit von Code 60
1.2.1 Bibliotheksmodule 61
1.2.2 Dokumentation von Code 67
1.3 Qualitätssicherung 77
1.4 Schnittstellenkonventionen 78
1.4.1 Erzeugung und Vernichtung von Zeigerbereichen . . . 79
1.4.2 Typzuweisung (cast – Operationen) 84
1.4.3 Eigentumsrechte 88
1.4.4 Die Größe von Feldern 91
1.4.5 Pufferüberläufe 95
1.4.6 Importverwendung 96
1.4.7 Operatorenverwendung 102

2 Container und Algorithmen 107
2.1 Einleitung 107
2.2 Template-Klassen, inline-Funktionen 109
2.2.1 Template-Klassen und Template-Funktionen 109
2.2.2 Spezialisierungen 112
2.2.3 Offener Code 113
2.2.4 Partielle Übersetzung 114
2.2.5 Default–Parameter und template-template-Parameter . 115
2.2.6 Rückgabe von Typen 116
2.2.7 Zahlen als Templateparameter 117
2.2.8 Effizienz und inline -Code 118
2.3 Zugriffe auf Daten: Verallgemeinerte Zeiger 119
2.3.1 Iteratoren 119
2.3.2 Einsatz von Iteratoren 121
2.3.3 Spezialisierungen für Container 122
2.3.4 Iteratorkategorien 123
2.3.5 Iteratoren und konstante Iteratoren 124
2.3.6 Iteratorabstand und Iteratorvorschub 126
2.3.7 Iteratorgültigkeit 127
2.3.8 Spezielle Attributtypen 128
2.3.9 Rückwärtsiteratoren 130
2.4 Verwaltung des Objektspeichers 131
2.4.1 Einführung 131
2.4.2 Allokator–Klassen 132
2.4.3 Eigene Allokatorklassen 134
2.5 Feld- oder Listencontainer 135
2.5.1 Felder(STL-Klasse vector) 136
2.5.2 Segmentierte Felder (STL-Klasse deque) 140

2.5.3 Warteschlangen (STL-Klassen Stack und Queue) . . . 142
2.5.4 Bitfelder . . . 143
2.5.5 Zeichenketten Strings . . . 144
2.5.6 Objekte und Zeiger in Containern . . . 147
2.5.7 Verkette Listen (STL-Klasse list) . . . 148
2.6 Bäume . . . 152
2.6.1 Teilordnung und Vollordnung . . . 152
2.6.2 Heap (STL-Klasse priority_queue) . . . 154
2.6.3 Binärer (Rot-Schwarz)-Baum . . . 157
2.6.4 STL-Klassen set und map/Hashsortierung . . . 170
2.6.5 B+ – Bäume . . . 175
2.7 Algorithmen und Container . . . 186
2.7.1 Sortierrelationen . . . 187
2.7.2 Suchen in unsortierten Containern . . . 189
2.7.3 Suchen in sortierten Containern . . . 189
2.7.4 Bubblesort-Sortieralgorithmus . . . 190
2.7.5 Quicksort-Sortieralgorithmus . . . 191
2.7.6 Heapsort-Sortieralgorithmus . . . 194
2.8 Suchen in Strings . . . 197
2.8.1 Einführende Bemerkungen . . . 197
2.8.2 Naive Suche . . . 198
2.8.3 Boyer-Moore-Algorithmus . . . 199
2.8.4 Suffix-Bäume . . . 203
2.9 Algorithmen der STL . . . 209
2.9.1 Grunddesign der Algorithmen . . . 210
2.9.2 Suchalgorithmen für einzelne Elemente . . . 212
2.9.3 Suchen nach mehrfach auftretenden Elementen . . . 213
2.9.4 Vollständige Übereinstimmung . . . 214
2.9.5 Binärsuche . . . 214
2.9.6 Anzahlen bestimmter Elemente . . . 214
2.9.7 Unterschiede und Ähnlichkeiten . . . 215
2.9.8 Enthaltensein von Elementen . . . 215
2.9.9 Kopieren von Containern . . . 216
2.9.10 Austauschen von Elementen . . . 216
2.9.11 Löschen von Elementen . . . 217
2.9.12 Reihenfolgeänderungen . . . 217
2.9.13 Extremalwerte . . . 219
2.9.14 Mischen von Containern . . . 220
2.10 Relationen und eigene Algorithmen . . . 222
2.10.1 Binäre und unäre Operatoren . . . 222
2.10.2 Adapterklassen für komplexe Operationen . . . 224
2.10.3 Aufwandsabschätzung . . . 229
2.10.4 Ein Beispiel . . . 231

3 Nützliche Werkzeuge 233
3.1 Namensbereiche und hilfreiche Templates 233
3.2 Umwandeln in Strings 236
3.3 Parameterstrings 241
3.3.1 Grundgerüst 241
3.3.2 Das Zerlegen und Rekonstruieren eines Strings 243
3.3.3 Arbeiten mit dem XMLString 246
3.4 Ablaufverfolgung (TRACE) 247
3.4.1 Debugger oder Tracer? 247
3.4.2 Eine einfache Trace-Klasse 248
3.4.3 Konditionelle Trace-Klassen 249
3.4.4 Trace-Gruppen 250
3.5 Objektstatistiken 253
3.6 Laufzeitmessungen 254
3.7 Datenkompression 257
3.7.1 Ein wenig Theorie 257
3.7.2 .. und eine Kompressionsklasse für die Praxis 261
3.8 Temporäre Dateien 265
3.9 Verschlüsselte Dateien 270
3.9.1 Die Aufgabenstellung 270
3.9.2 Der Algorithmus 271
3.9.3 Der Einsatz des Algorithmus 273
3.9.4 Die Implementation 275
3.9.5 Bemerkungen zur Verschlüsselung 280
3.10 Textdateien und Verzeichnisse 282
3.11 Laufwerksimulation 285
3.11.1 Die „File Allocation Table“ FAT 285
3.11.2 Verzeichnisse 286
3.11.3 Dateideskriptor 287
3.11.4 Simulation eines Laufwerks 288
3.11.5 Freie Sektoren und Zuordnung zu Dateien 290
3.11.6 Initialisierung eines Laufwerks 291
3.11.7 Laufwerk öffnen 294
3.11.8 Dateien öffnen 296
3.11.9 Verzeichnisse erzeugen 297
3.11.10 Verzeichnis wechseln 297
3.11.11 Löschen von Dateien 298
3.11.12 Löschen von Verzeichnissen 299
3.11.13 Abschlussbemerkungen 300

4 Lineare Algebra/mehrdimensionale Felder 301
4.1 Matrizen in C++ 301
4.1.1 Normal besetzte Matrizen 301
4.1.2 Schwach besetzte Matrizen 310
4.1.3 Compilezeitoptimierungen – Vektoren und Matrizen . . 315

4.2 Numerisch–Mathematische Klassen 316
4.2.1 Das Rundungsproblem 316
4.2.2 Algebraische Eigenschaften 318
4.2.3 Konstantenvereinbarungen 320
4.2.4 Vergleiche und Nullprüfungen 321
4.2.5 Anwendung auf schwach besetzten Matrizen 324
4.3 Einige Algorithmen der linearen Algebra 324
4.3.1 Lineare Gleichungssysteme 325
4.3.2 Eigenwerte von Matrizen 330

5 Ausnahmen und Zeigerverwaltung 333
5.1 Zur Arbeitsweise mit Ausnahmen 334
5.2 Typermittlung und Zugriffsstandardisierung 340
5.2.1 Ableitung definierter Typen 340
5.2.2 Zugriffsnormierung 342
5.2.3 Ermittlung der Typart 343
5.3 Verwaltung von Zeigervariablen 346
5.3.1 Manuelle Ausnahmeverwaltung 346
5.3.2 Platzhalter- oderTrägervariable 347
5.3.3 Eine Instanz – mehrere Variable 349
5.3.4 Mehrfachreferenzen und automatische Verwaltung . . . 354
5.3.5 Zeigerkopien 357
5.3.6 Mischen der Funktionalität, Zulässige Zuweisungen . . 359
5.3.7 Vollautomatische Policy-Auswahl 361
5.4 Steuerung der Ausnahmebehandlung 363
5.4.1 Anforderungen an die Ausnahmesteuerung 363
5.4.2 Implementation 1: Realisierung der Objektleitung . . . 366
5.4.3 Implementation II: Mischen von Strategien 371
5.5 Anwendungsbeispiel: Transaktionsmanagement 378

6 Objektfabriken 385
6.1 Laufzeitobjektfabrik 385
6.1.1 Motivation 385
6.1.2 Die Basisklasse für Fabrikobjekte 387
6.1.3 Klassenidentifikation und Persistenzmodell 390
6.1.4 Die eigentliche Objektfabrik 396
6.1.5 Benutzung neuer Methoden 399
6.1.6 Trennung von Anwendung und Bibliothek 405
6.1.7 Dynamische Einbindung einer DLL 408
6.2 Compilezeit-Objektfabriken 409
6.2.1 Typlisten 410
6.2.2 Zugriff auf einen Typ in der Liste 412
6.2.3 Algorithmen auf Typlisten 413
6.2.4 Arbeiten mit Typlisten 420
6.2.5 Beispiel: Compiletime-Objektfabrik 427
6.3 Applets und Sandbox in C++ 430
6.3.1 Das Sandbox-Konzept 430

6.3.2 Sandbox in C++ Umgebungen . . . 431
6.3.3 Die Applet-Basisklasse . . . 432
6.3.4 Der Security-Manager . . . 434
6.3.5 Aufrufe und Probleme . . . 435

7 Grafen . . . 437
7.1 Grafen und ihre Speicherung . . . 437
7.2 Arten des Eckenverbundes . . . 441
7.2.1 Distanzlisten . . . 442
7.2.2 Verbundenheit von Grafen . . . 444
7.2.3 Abspalten disjunkter Subgrafen . . . 444
7.2.4 Zyklenfreie (Sub)Grafen . . . 445
7.3 Spannende Bäume . . . 446
7.3.1 Breitensuche . . . 447
7.3.2 Tiefensuche . . . 448
7.3.3 Minimale (Maximale) Bäume . . . 449
7.4 Wege in Grafen . . . 450
7.4.1 Beliebige Wege und Zyklen . . . 451
7.4.2 Wege mit kleiner Kantenanzahl . . . 451
7.4.3 Minimale (Maximale) Wege . . . 451
7.4.4 Rundwege in Grafen . . . 454
7.4.5 Rundreise durch die Ecken . . . 457
7.5 Netzwerke . . . 458
7.5.1 Flüsse in Netzwerken . . . 459
7.5.2 Flüsse mit Nebenbedingungen . . . 460
7.5.3 Belegungsprobleme . . . 461

8 Intervalle . . . 463
8.1 Einführung . . . 463
8.2 Funktion eines Intervallcontainers . . . 464
8.3 Intervallimplementation . . . 465
8.4 Relationen zwischen Intervallen . . . 467
8.4.1 Überlappung/Durchschnitt . . . 467
8.4.2 Vereinigung und Differenz . . . 468
8.5 Intervallcontainer . . . 470
8.5.1 Relationen zwischen Intervallen . . . 470
8.5.2 Containerimplementation . . . 471

9 Ausdrücke . . . 475
9.1 Einführung . . . 475
9.2 Zerlegung der Ausdrücke . . . 476
9.2.1 Überführung von Methoden in Objekte . . . 476
9.2.2 Typkonversion . . . 477
9.2.3 Gerüste für binäre und unäre Ausdrücke . . . 480
9.3 Datenobjekte in den Ausdrücken . . . 481
9.3.1 Felder . . . 482

9.3.2 Variablen 483
9.3.3 Konstante 484
9.3.4 Funktionsobjekte 486
9.4 Ein Beispiel 486

Band II

10 Speicherverwaltung (und ein wenig mehr) 489
10.1 Die Laufzeitproblematik 489
10.2 Das einfache Referenzkonzept 490
10.3 Referenzen mit temporärer Zwischenspeicherung 494
10.3.1 Die Strategie 495
10.3.2 Die Basisklasse 496
10.3.3 Die Ankerobjekte der Speicherverwaltung 498
10.4 Ein universeller Datenpuffer 500
10.5 Speicherklasse für kleine Objekte 504
10.5.1 Basis einer Speicherbank 505
10.5.2 Objekte fester Größe 507
10.5.3 Die allgemeine Allokatorklasse 508
10.5.4 Eine Basisklasse für die Allokatornutzung 510

11 Koordination von Abläufen 511
11.1 Grafische Anwenderschnittstellen 511
11.1.1 Bildschirmobjekte und Ereignisse 512
11.1.2 Ereignisketten 515
11.1.3 Änderung des Objektbaumes 521
11.1.4 Das Gesamtdesign 522
11.1.5 Grafische Anwendungsentwicklung 523
11.2 Funktoren – Aktoren 527
11.2.1 Verschieben von Funktionsaufrufen 528
11.2.2 Aufruf von (virtuellen) Klassenmethoden 532
11.3 Filterschlangen 534
11.3.1 Einfache Schlangen (Einführung) 534
11.3.2 Filterobjekt aus der Datenübertragung (Beispiel) 541
11.3.3 Verzweigungen 543

12 Bildverarbeitung 551
12.1 Vorbemerkungen 551
12.2 Analogbearbeitung von Bildern 553
12.2.1 Farbe, Kontrast, Helligkeit 553
12.2.2 Größenänderungen, Drehungen, Verzerrungen 554
12.2.3 Schärfung des Bildes 557
12.3 Strukturelle Bearbeitung/Digitalisierung 560
12.3.1 Digitalisierung von Konturen 560

12.3.2 Relationen zwischen Kantenlinien 565
12.3.3 Vorverarbeitung/Skelettierung von Bildern 569
12.4 Bildvergleiche 570
12.4.1 Pixelgestützte Ähnlichkeitsanalyse 570
12.4.2 Methoden der linearen Algebra 571

13 Computergrafik 575
13.1 Einleitung 575
13.2 Systemumgebung 576
13.2.1 Systeminitialisierung 576
13.2.2 System-Basisklasse und aktives Objekt 577
13.2.3 Objektinitialisierung und Projektionsmatrizen 580
13.2.4 Ereignisfunktionen 582
13.3 Daten und Datencontainer 586
13.3.1 Punkte und Punktcontainer 586
13.3.2 Punkte auf einem Gitter und Flächendarstellung 589
13.3.3 Indizierte Punktlisten 591
13.4 Objekte und Szenen 593
13.4.1 Basisklasse 593
13.4.2 Objektklasse 594
13.4.3 Szenen 595
13.4.4 Objektbibliotheken 597
13.5 Beleuchtungseffekte 599
13.5.1 Grundlagen 599
13.5.2 Lichtquellen 600
13.5.3 Objekteigenschaften 603
13.6 Perspektivische Projektion 606
13.6.1 Grundlagen der perspektivischen Darstellung 606
13.6.2 Projektionsdefinition 608
13.7 Flächendarstellungen 610
13.7.1 Texturen 610
13.7.2 Funktionen 612
13.7.3 Bezierflächen 613
13.7.4 NURBS-Freiformflächen 615
13.8 Listenverwaltung durch OpenGL 617
13.9 Offene Probleme 618

14 Datenstrukturen und ASN.1 619
14.1 Einführung in die Syntax 620
14.2 Binärkodierung 629
14.3 Übersetzen von Quellkode: Interpreter–Modus 636
14.3.1 Parsen der Kodebestandteile 636
14.3.2 Konstruktion der Felddatentypen 643
14.3.3 Bereichsdefinitionen 648
14.3.4 Elimination selbstdefinierter Typen 650
14.3.5 Auflösung der gegenseitigen Abhängigkeiten 652

14.4 Prüfung von Datensätzen 654
14.5 Datenbank und Anwendungsverknüpfung 657
14.5.1 Ein einfaches Datenbankmodell 657
14.5.2 Anwendung auf die ASN.1–Objekte 660
14.5.3 Verknüpfung mit anderen Datenobjekten 661
14.6 Verknüpfung mit den Filterklassen 663
14.7 Compilezeit – Implementation 671

15 Zahlendarstellungen 677
15.1 Ganze Zahlen 679
15.1.1 Basisalgorithmen 680
15.1.2 Anmerkungen zur Implementation 696
15.1.3 Verbesserung der Effizienz 697
15.2 Quotientenkörper 712
15.3 Restklassenkörper 715
15.3.1 Theoretische Grundlagen 715
15.3.2 Implementation der Restklasse 717
15.4 Fliesskommazahlen 719
15.4.1 Grundlagen 719
15.4.2 Klassenkonstruktion 721
15.4.3 Addition und Subtraktion 723
15.4.4 Division i 724
15.4.5 Division ii 724
15.4.6 Division iii 726
15.4.7 Relationen 726
15.4.8 Reelle Konstanten und Funktionen 728
15.4.9 Interpolation von Werten 731
15.5 Die Körper F_{2^m}, F_{p^m} 732
15.6 Metaprogramme und Körpererweiterungen 734
15.6.1 Theoretische Vorbemerkungen 734
15.6.2 Implementation der Algorithmen 736

16 Numerische Anwendungen 745
16.1 Rundungsfehler 745
16.2 Kontrolle von Fehlern 748
16.3 Arbeiten mit Polynomen 751
16.3.1 Eigenschaften von Operatoren 751
16.3.2 Nullstellen I: Berechnen 754
16.3.3 Nullstellen II: Finden 756
16.4 Intervallmathematik 758
16.4.1 Grundlagen 758
16.4.2 Vergleiche gerundeter Zahlen 759
16.4.3 Zwischenbilanz 763
16.4.4 Intervalltypen 764
16.4.5 Implementierung einer Intervallklasse 769
16.4.6 Einsatz der Intervallrechnung 776

17 Prä- und Postprozessing 781
17.1 Hintergrund 781
17.2 Präprozessing 781
17.2.1 Präprozessing mit spezieller Funktion 782
17.2.2 Präprozessing durch Objektmethode 782
17.2.3 Präprozessing mit Singleton-Objekt 783
17.2.4 Varianten, Kritik 784
17.3 Prä- und Postprozessing 785
17.3.1 Prä- und Postprozessing-Methodenverwaltung 786
17.3.2 Methodentypisierung 787
17.3.3 Die Funktorklasse(n) 788
17.3.4 Instanziierungsmethoden 790

18 Programm- und Prozesssteuerung 791
18.1 Allgemeines 791
18.2 Threads 793
18.2.1 Allgemeines 793
18.2.2 Erzeugen und Kontrollieren 794
18.2.3 Exklusive Programmteile 796
18.2.4 Synchronisation von Threads 799
18.3 Kommunikation zwischen Prozessen 803
18.3.1 Sockets 803
18.3.2 Verteilte Objekte 808
18.4 Parallele und massiv parallele Prozesse 810

Stichwortverzeichnis 813

Einführung in die Programmierung

1 (Statt eines) Vorwort(s)

Programmieren ist ein Handwerk. Wie in jedem Handwerk muss man viel und lange üben und mit der Sache praktisch umgehen, um es zur Meisterschaft zu bringen, und wer sich über die Meisterschaft hinaus noch ein wenig Genialität bewahrt, darf sich getrost auch als Künstler seines Fachs sehen.[1]

Meisterschaft in einem Handwerk erfordert aber auch die Auseinandersetzung mit Techniken, die über den voraussichtlichen späteren täglichen Bedarf hinausgehen. In Bezug auf die Programmierung umfasst dies beispielsweise auch ziemlich abstrakte Konstrukte, mit denen sich der Compiler zu beschäftigen hat und die im Bedarfsfall die Produktion von ausführendem Code sicherer und schneller machen. Nach diesem Gesichtspunkt sollte zumindest die Lehre ausgerichtet sein, und hier bietet gerade C++ einen gegenüber anderen Programmiersprachen einen sehr weiten Horizont.

Ohne dies dem Leser jetzt als Wertung für die praktische Arbeit zu verstehen geben zu wollen, finde ich es bedauerlich, wenn gerade in der Lehre Programmiersprachen wie C++ zugunsten einfacherer Sprachen wie Java geradezu abgeschafft werden. Wer mit C++ arbeiten kann, wird ohne Probleme auch mit Java guten Code produzieren können, was man umgekehrt aber entschieden verneinen muss. Auf eine andere Ebene übersetzt, wird wohl kaum jemand bezweifeln, dass ein Tischler, dessen Ausbildung sich auf das Anfertigen von Fußbänkchen beschränkt hat, mit einem intarsien- und schnitzereibeladenen Chorgestühl etwas überfordert ist. Die Argumentation, mit dem kleineren Werkzeug sicherer programmieren zu können, weil „gefährliche“ Sachen wie Zeiger und Mehrfachvererbung nicht vorhanden sind, ist wohl auch eher geeignet, die handwerkliche Qualifikation des Argumentierenden

[1] Womit ich schon in Konflikt mit weiten Bereichen der heutigen Kunstszene gerate, die die Meisterschaft im Umgang mit der Materie durch meisterhafte Worthülsen ersetzt. In der Technik wird diese Sparte Künstler oft durch Unternehmensberater vertreten, in der Politik durch so genannte Experten.

in Zweifel zu ziehen, denn wenn man weiß, was man macht, erledigt sich das Argument schnell von selbst.[2]

Kurz – um diese leider notwendige Polemik abzuschließen – fühlen Sie sich aufgefordert, zum Erlernen von Programmiertechniken zu C++ zu greifen, unabhängig vom späteren tatsächlichen Bedarf. Der regelt sich ohnehin von alleine, denn in der Produktion ist angesagt, das geeignetste Werkzeug für die Lösung der Aufgabe heranzuziehen und nicht in ideologischen Grabenkämpfen zu verharren. Im Laufe dieses Buches werden wir nahezu sämtliche Eigenschaften von Java mit einigen Handgriffen auch in C++ realisieren (was der Sicherheitsargumentation den restlichen Boden entzieht) und gleichzeitig dies als nur eine von mehreren möglichen Strategien erkennen.

Wie Eingangs erwähnt, besteht der Weg zur Meisterschaft in einem Handwerk aus ein wenig Theorie und viel praktischer Übung, und so ist dieses Buch aufgebaut. Ich habe versucht, so viele Arbeitsgebiete wie möglich anzusprechen und die dort eingesetzten Techniken zu vermitteln oder doch zumindest die Prinzipien, nach denen die dort jeweils eingesetzten professionellen Werkzeuge funktionieren. Die praktische Lösung steht also – durchaus im Sinne des Zeitgeistes – immer im Vordergrund, und sämtliche Programmiertechniken, die dazu notwendig sind, werden ohne große Berücksichtigung des Schwierigkeitsgrades eingesetzt. Das mag für den einen oder anderen Leser etwas ungewohnt und auch anstrengend sein, ist er doch in vielen hier gestellten Aufgaben persönlich aufgefordert, fehlende Kodeteile zu ergänzen und die Anwendung zum Laufen zu bringen. Aber nur so, durch eigene praktische und – eindeutig nicht im Sinne des Zeitgeistes – manchmal recht mühsame Beschäftigung mit dem Programmiersystem und seinen Tücken kommt man irgendwann zu einer gewissen Virtuosität. Lesen Sie also dieses Buch nicht als Feierabendlektüre und glauben dann, alles verstanden zu haben – implementieren Sie auch die Beispiele und Aufgaben und erleben Sie das Programmieren am Objekt.

Durch diese Vorgehensweise können die Kapitel in weiten Teilen auch in anderer Reihenfolge als der des Inhaltsverzeichnisses bearbeitet werden (wo Rückgriffe notwendig sind, finden Sie entsprechende Kommentare); andererseits setzt die Bearbeitung jeweils einige Grundkenntnisse sowie die Bereitschaft, einige neue Sachen parallel zu erlernen, voraus. Damit auch Anfänger sich erfolgreich in die C++ Programmierung einarbeiten können, ist das nullte Kapitel für diejenigen Leser gedacht, die noch über keinerlei Programmiererfahrungen verfügen. Leser mit ersten Programmiererfahrungen in C oder C++ können es überspringen, und Quereinsteigern von anderen Programmiersprachen wird einiges bekannt vorkommen; allgemein sei ein kurzes Querlesen aber trotzdem empfohlen, denn so lernen wir uns ein wenig kennen und schließlich und endlich haben Sie mit dem Kauf des Buches auch für diesen Teil bezahlt. Wer noch keine Erfahrung mit objektorientierter Programmierung hat, sollte zumindest die Anmerkungen über C++ lesen. Die mit

[2] Im Gegenteil lässt sich sogar argumentieren, dass C++ in Bezug auf die Typsicherheit beispielsweise Java um Längen voraus ist.

dem Kapitel Null beginnende Kapitelnummerierung weist auf die verschiedenen Einstiegsmöglichkeiten hin und ist obendrein themenkonform, beginnen doch alle Felder in C/C++ grundsätzlich ebenfalls mit dem Index Null.

Für ein effektives Arbeiten mit diesem Buch sollten Sie außerdem über einen Rechner mit einem C/C++–Programmentwicklungssystem, möglichst mit grafischer Bedienoberfläche und Debugger, verfügen. Das Programmentwicklungssystem sollte über ein online–Hilfesystem für die Bibliotheken verfügen. Laden Sie sich auch die C- und C++-Bibliothekshandbücher aus dem Internet auf Ihre System. Solche Hilfen sind nebst der Nutzung des Internets selbst als Informationsmedium in das Buchkonzept miteinbezogen. Sollten Sie einmal mit dem hier gebotenen Stoff nicht zurecht kommen, lässt sich so schneller und befriedigender die Lösung finden als durch ein noch umfangreicheres Buch, das sich in einer Fülle von Details verwirrt, viele Leser langweilt und alle durch einen noch höheren Preis erfreut. Nicht zwingend notwendig aber meistens recht hilfreich ist außerdem eine Arbeitsgruppe, in der Sie die erreichten Ergebnisse überprüfen oder neue Ideen sammeln können.

> **Wichtig!** Programmieren lebt von der Erfahrung. Probieren Sie alles, was in diesem Buch beschrieben wird, selbst aus. Trotz in der Regel recht ausführlicher Erläuterungen gibt es vieles, was erst durch die Praxis den Aha-Effekt auslöst. Glauben Sie nie, ausschließlich durch eine Theoriebetrachtung tatsächlich in der Lage zu sein, später eine funktionierende Anwendung entwickeln zu können. Machen Sie sich auf den einen oder anderen Misserfolg im ersten Anlauf gefasst, und verfallen Sie nicht in Frust, wenn sich nach stundenlangem Suchen die Ursache für die Fehlfunktion als ziemlich lächerlich herausstellt. Viele Aufgaben in diesem Buch sollen Sie dabei unterstützen, wobei ich auf Lösungen oft bewusst verzichte. Wenn Sie verstanden haben, was am Schluss realisiert sein soll, schaffen Sie das auch ohne Musterlösungen.[3]

Zu einer guten Programmierung sind noch einige weitere allgemeine Regeln zu beachten, die sich auch als „korrektes Benehmen“ umschreiben lassen und nicht nur für C++ Gültigkeit haben. Auch damit sollte man sich sehr frühzeitig vertraut machen, um sich nicht später sehr mühsam gewisse Sachen wieder abgewöhnen zu müssen. Das Ziel hierbei ist, von vornherein einen „Programmierstil“ zu fördern, der Fehler bereits bei der Programmerstellung eliminiert und nicht zu Überraschungen beim Anwender führt.[4]

Der Start in die Programmierung weicht ebenfalls vom Üblichen ab, hat sich aber praktisch bewährt. Programmieren verlangt zumindest in den Anfängen zwei Lernprozess beim Studierenden: Er muss sich eine bestimmte Denkweise aneignen, ohne die er einem Computer keine Aufgabe begreiflich machen kann, und er

---

[3] Und achten Sie auch auf die Fußnoten! Neben der einen oder anderen Anekdote zur Auflockerung oder vielleicht auch bissigen Kommentaren, mit denen nicht jeder einverstanden sein muss, finden Sie dort auch den einen oder anderen Hinweis zur Lösung einer Aufgabe.

[4] „Programmierstil“ ist eigentlich ein Unwort, wie Sie bei der weiteren Lektüre noch feststellen können. Hier passt es aber ganz gut, um die Aussage zu umschreiben.

muss die Vokabeln und Syntax einer Sprache lernen. Den unterschiedlichen Lernprozessen wird durch zwei Lernabschnitte Rechnung getragen: Zunächst wird die Denkweise in der normalen Umgangssprache trainiert und anschließend die zu der Denkweise gehörenden C/C++-Sprachelemente in Verbindung mit einfachen Übungen vorgestellt.[5]

2 Die ersten Schritte: Anweisungslisten

Die ersten Lernschritte zum Programmentwickler führen wir ohne Programmiersprache durch. Wir erstellen zunächst „Programme“ in normaler Umgangssprache und trainieren dabei die Besonderheiten, die beim Umgang mit Rechnern zu beachten sind. Nehmen Sie diesen Teil des Lehrgangs besonders ernst. Vieles in der Umgangssprache ist sehr vage formuliert oder drückt gar nicht die Prozesse aus, die bei der Ausführung einer Aufgabe im Gehirn ablaufen – wir bewältigen trotzdem alles fast mühelos. Die für den Umgang mit einem Computer notwendige Detailliertheit und Präzision der Beschreibung der Vorgänge lässt sich aber schrittweise erreichen und trainieren – im Gegensatz zur Programmiersprache, die von vornherein die maximale Präzision erfordert und dadurch Anfänger oft überfordert.

Ihre ersten Programmierschritte werden prozedural sein, das heißt sie beschreiben den Ablauf von Vorgängen und versuchen nicht, komplexere Zusammenhänge zwischen Wirkeinheiten (*Objekten*) zu beschreiben. Lassen Sie sich darin nicht von Anhängern der „objektorientiert von Anfang an“–Doktrin verwirren, die manchmal etwas rüde gegen diesen Einstieg opponieren. Der Übergang zu einer Herangehensweise, die neben Objekten auch Muster und Eigenschaften (pattern, traits) einbezieht,[6] erfolgt recht schnell.

2.1 Die Grundregeln

Ein prozedurales Programm ist auch in Umgangssprache eine Liste von Anweisungen, die ausgeführt werden müssen, um von einem vorgegebenen Startpunkt das gewünschte Ergebnis zu erhalten. Die Aufgabe kann beispielsweise sein, die Buchstaben eines vorgegebenen Satzes lexikalisch zu ordnen, also

```
"Hallo Welt!"      ===>    "!aeHlllotW"
```

Stellen Sie als erstes immer sicher, dass Sie die Aufgabe verstanden haben! Meine Studenten bekommen selten ein Aufgabenblatt, sondern meist eine mündliche

[5] Das ist die sogenannte „prozedurale“ Vorgehensweise, die heute meist als antiquiert im Vergleich zur „objektorientierten“ bezeichnet wird. Lassen Sie sich davon nicht abschrecken, denn es kommt nur auf das Endergebnis an, und das wird hier stimmen. Auf die philosophischen Unterschiede zwischen prozedural und objektorientiert komme ich später noch einmal zurück.

[6] Den Blick darauf verstellen sich die OOP-Anhänger meist selbst durch gewisse Scheuklappen.

Aufgabenstellung, die sie selbst notieren und deren Übereinstimmung mit meinen Vorstellungen sie kontrollieren müssen. Das erfordert unter Umständen einige Diskussion, und sie müssen einige Teile selbst zusammen suchen. Wird beispielsweise gefordert, dass Teile einer Anwendung „genau so funktionieren wie FILE in C“, so müssen sie sich über die Eigenschaften des Datentyps FILE informieren, und die Ausrede, „das hat aber auf dem Aufgabenblatt nicht gestanden“, gilt nicht.[7]

Das korrekte Verständnis wird durch eine präzise Formulierung der Aufgabenstellung und der zu beachtenden Randbedingungen (*präziser als die mündliche erste Formulierung, die noch vom automatischen Mitdenken Gebrauch macht und Randbedingungen meist gar nicht beachtet*) sowie ein vollständiges Beispiel, bestehend aus Eingabe- und Ausgabewert wie oben, dokumentiert. Das Beispiel muss vollständig sein, das heißt es besteht bei komplexeren Aufgaben mit mehreren möglichen Ergebnissen auch aus verschiedenen Teilbeispielen für jeden auftretenden Fall.

Ein Fallbeispiel sollte möglichst einfach sein, denn es dient später als Prüfstein, ob Ihr „Programm“ korrekt arbeitet. Aufgrund Ihrer Anweisungen muss nämlich aus dem Eingabewert der Ausgabewert entstehen, und zwar ohne, dass sie zwischendurch die Lust verlieren. Das bezieht sich sowohl auf diese Lerneinheit, in der Sie selbst die Anweisungen von Hand ausführen müssen, als auch auf die späteren Aufgaben, in denen die Maschine in annehmbarer Zeit zu einem Ergebnis kommen muss beziehungsweise im Fehlerfall eine realistische Chance bestehen muss, den Fehler zu finden. Beispiele kann man sich wie hier von Hand konstruieren oder in komplexeren Fällen aus dem Internet besorgen beziehungsweise sich von bereits bestehenden Programmen konstruieren lassen.

Ist die Aufgabenstellung mit dem Auftraggeber geklärt und das Beispiel als vollständige und korrekte Arbeitsvorlage akzeptiert, so kann die Anweisungsliste konstruiert werden. Diese besteht aus einzelnen nummerierten Sätzen und muss drei Konstruktionsprinzipien genügen: Sie muss **zustandssicher**, **unterbrechbar** und **elementar** sein.

Unter **Zustandssicherheit** verstehen wir Klarheit darüber, was wo gemacht werden soll. Die Anweisung *„notieren Sie diese (ganze) Zahl“* ist nicht zustandssicher, da nicht gesagt wird, wo der Wert zu notieren ist. Das Notieren einer Information hat nur dann Sinn, wenn später auf sie zurückgegriffen werden soll, und die Anweisung *„nehmen Sie die notierte ganze Zahl“* macht ohne Ortsangabe spätestens dann keinen Sinn mehr, wenn mehr als eine Zahl notiert worden ist. Die Anweisung muss daher beispielsweise lauten *„notieren Sie die Zahl auf dem Zettel GZ_NOTIZ“*, wobei zur Eindeutigkeit vorher noch festgelegt werden muss *„Stellen Sie einen Zettel mit der Aufschrift GZ_NOTIZ zum Merken ganzer Zahlen zur Verfügung“*. Mit

[7] Das mag dem Einen oder Anderen vielleicht etwas eigenartig erscheinen, gibt jedoch die reale Welt recht gut wieder: der Auftraggeber formuliert seine Wünsche in einem Lastenheft in seiner Sprache (z.B. Betriebswirtschaft), der Programmierer muss dies zunächst in die Sprache seiner Welt übersetzen und anschließend eine Rückübersetzung als Pflichtenheft erstellen, das er nun mit dem Auftraggeber auf die korrekte Erfassung der Wünsche und Möglichkeiten abstimmen muss. Hat er das schlampig gemacht, ist Ärger mit den Anwälten des Auftraggebers vorprogrammiert. Warum also nicht gleich hier anfangen, das hier ordentlich zu üben?

anderen Worten, es müssen alle Hilfsmittel zu Beginn der Anweisungsliste festgelegt werden, und die Anweisungen beziehen sich jeweils auf den Inhalt bestimmter Hilfsmittel oder der Ein-/Ausgabemittel.

Zum Begriff der **Unterbrechbarkeit** stellen Sie sich vor, einer Ihrer Mitstudierenden bearbeitet die Anweisungen Schritt für Schritt an der Tafel, und zwar nach klassischer Beamtenmentalität: Alles schön wörtlich nehmen und nicht drüber nachdenken.[8] An irgendeiner Stelle lösen Sie ihn durch einen anderen Studenten ab, der vor der Tür gewartet und deshalb nichts mitbekommen hat. Außer der Satznummer, mit der er die Bearbeitung fortsetzen soll, erhält er keinerlei Informationen. Die Anweisungen müssen so konstruiert sein, dass die Arbeit nach der Unterbrechung reibungslos und korrekt fortgesetzt werden kann. Dies schließt die Zustandssicherheit, aber auch noch weitere Eigenschaften ein.

Eine Anweisung ist **elementar**, wenn zu ihrer Durchführung genau ein Arbeitsschritt notwendig ist. Nehmen wir als Beispiel die Anweisung *„Gehe zum Anfang von Kapitel Zwei“*. Man kann jetzt entweder im Text blättern, bis die Überschrift „Kap. 2“ gefunden wird, oder im Inhaltsverzeichnis nachschauen, auf welcher Seite das Kapitel beginnt, auf jeden Fall aber mehrere Schritte hintereinander durchführen. Wenn hier unterbrochen wird, ist nicht in jedem Fall klar, was zu geschehen hat: Nehmen wir an, es ist zurück zu blättern, aber vor der Unterbrechung wird Variante Zwei (*im Inhaltsverzeichnis nachschauen*) ausgewählt. Vom Inhaltsverzeichnis zurückblättern führt bei einer Fortsetzung aber nicht zu Kapitel Zwei. Diese Eigenschaft schließt die Unterbrechbarkeit der Anweisungsliste ein, fordert aber zusätzlich auch die Unterbrechbarkeit der Arbeit selbst.

Wie man sich leicht überlegen kann, ist das erste Erfolgsrezept zum Erreichen dieser Eigenschaften das Vermeiden einer Reihe von Worten in den Anweisungen (*erinnert Sie das an ein Gesellschaftsspiel? Richtig*). Beispielsweise dürfen Sie folgende Formulierungen nicht einsetzen:

- . . . nimm nun **das nächste** Zeichen . . . (*welches denn?*)
- . . . wiederhole **das Ganze** . . . (*alles, nur einen Teil, oder was?*)
- . . . **und so weiter** bis **zum Ende** . . . (*und so weiter was? Wie ist das Ende zu erkennen?*)
- . . . **suche** das Zeichen . . . (*wie macht man das denn?*)
- . . . **bestimme** die Position . . . (*wie geht man dabei vor?*)
- . . . **mache** (***wiederhole***) das, **bis** . . .

Die Liste können Sie selbst mit weiteren Formulierungen erweitern. In der Praxis ist nun folgendermaßen vorzugehen: Schreiben Sie eine oder mehrere Arbeitsanweisungen auf und vergewissern Sie sich, dass auf diesem Weg das Ziel zu erreichen ist. Prüfen Sie nun jede Arbeitsanweisung, ob nach einer Unterbrechung eine Fortsetzung ohne zusätzliche Informationen möglich ist. Die eine oder andere Formulierung, die nicht verwendet werden darf, fällt dann sicher noch auf,

[8] Ich bin selbst Beamter, also darf ich so was sagen.

und in einer Gruppe lässt sich dies auch sehr gut trainieren, in dem man sich bei der Ausführung der Anweisungen eines Mitstudenten vorsätzlich besonders dumm anstellt (*besser formuliert: Genau die Fehler macht, die die Formulierung noch zulässt, also bewusst gegen das Ziel arbeitet*). Ist das noch möglich, so ist entweder die Formulierung zu präzisieren oder der Arbeitsschritt ist durch Unterteilung in weitere Anweisungen zu verfeinern. Probe und Verfeinerung wird so lange wiederholt, bis keine Korrekturen mehr notwendig sind.

Der Vorteil dieser Vorgehensweise ist die Möglichkeit der schrittweisen Verbesserung der Arbeitsweise. In C/C++ muss das Endergebnis sofort erscheinen, Zwischenschritte sind nicht möglich. Genau das führt aber zu Verständnisproblemen.

> **Aufgabe.** Schlagen Sie das Buch an einer beliebigen Stelle auf und suchen Sie den Beginn des zweiten Kapitels. Sie dürfen nur vor oder zurück blättern. Erzeugen Sie eine Anweisungsliste, die die Aufgabe löst, sowie eine Liste verbotener Worte.

2.2 Zur Arbeitstechnik

Arbeiten Sie möglichst mit einem Textverarbeitungsprogramm, wenn Sie diese Übungen durchführen. Änderungen können dann sehr leicht durchgeführt werden – im Gegensatz zu handschriftlicher Ausarbeitung, deren Aufwand manchen Bearbeiter veranlassen mag, Unfug erst mal stehen zu lassen.

Falls Sie einen Übungspartner haben: Erstellen Sie die Liste gemeinsam an einem Rechner, wobei nach jeweils zehn Minuten die Tastatur ausgetauscht wird. Während der eine seine Anweisungen oder Vereinbarungen erfasst, kann der andere seine Eingaben planen (*Erweiterungen oder Änderungen*).

Beschränken Sie die Diskussion auf ein Minimum und lassen Sie Ihren Partner arbeiten (*auch wenn es schön ist, durch Fragen und Vorschläge abzulenken*). Gehen Sie aber die Anweisungen ihres Partners so kritisch wie möglich durch. Stellen Sie dazu ein einfaches Beispiel auf und lassen Sie sich anschließend die Anweisungen Ihres Partners vorlesen und führen Sie sie aus. Was dabei falsch zu verstehen sein könnte, verstehen Sie bitte auch falsch. Verboten sind dabei Korrekturanweisungen des Vorlesers. Kommt ein falscher Zustand bei der Ausführung einer Anweisung heraus, muss der Anweisungstext (und ggf. auch andere Teile der Befehlsliste) so geändert werden, dass das bei einer erneuten Durchführung der Aufgabe nicht mehr passiert.

2.3 Aufbau der Anweisungsliste

Eine Anweisungsliste beginnt mit einer Überschrift, die im Hinblick auf die spätere Umsetzung in Programmkode ausdrücken sollte, was die Aufgabe dieser Anweisungsliste ist, also nicht „Liste 13“ oder „GB_AW.47.1.2“. Das ist zwar zulässig, erschwert aber später das Wiederfinden. Im Folgenden wird zwischen den

Winkelklammern <...> das Kapitel einer Anweisungsliste angegeben, gefolgt von einem Beispiel.

```
<Name der Anweisungsliste mit Aussage über
 die Aufgabe>: Lexikalisches Sortieren
```

Der Name sollte auch nicht allzu lang beziehungsweise sinnvoll abkürzbar sein, denn viele Listen können in anderen wiederverwendet werden, und jedes Mal dann eine ellenlange Überschrift anzugeben, um die Unterliste zu spezifizieren, ist auch recht lästig. Gegebenenfalls kann der genaue Zweck der Liste durch einen Kommentar angegeben werden.

Im zweiten Schritt wird die Aufgabe durch Beispiele (*die Theorie*) verdeutlicht:

```
<Theorie>  "Hallo Welt"   ==>  "aeHlllotW"
           "Hallo Luise"  ==>  "aeHiLllosu"
```

Die Beispiele oder die Theorie sollten möglichst vollständig sein, das heißt alle in der Praxis auftretenden Fälle abdecken, wie hier beispielsweise die Reihenfolge von Groß- und Kleinbuchstaben. Fragen Sie sich kritisch, ob Sie alle Fälle berücksichtigt haben oder doch noch etwas fehlt, denn mehr, als Sie beschreiben, soll Ihr „Programm“ auch nicht erledigen. Bei großen Lücken kann es dann natürlich für die Praxis unbrauchbar werden.

Nach der Bezeichnung und der Theorie folgt die wichtigste Überlegung in Bezug auf die Zusammenarbeit mit Kunden oder anderen Listenprogrammierern: Die Festlegung der Ein- und Ausgabe, im Fachjargon auch als Schnittstellendefinition bezeichnet.

```
<Beschreibung der Ein- und Ausgabe>
   Der Satz wird auf einem Blatt entgegengenommen
   Der sortierte Satz wird auf dem gleichen Blatt
     wieder an den Auftraggeber zurückgegeben.
     Weitere Rückgabedaten gibt es nicht.
<für den Auftraggeber>
   Der Satz ist auf kariertem Papier, beginnend im
     ersten Kästchen oben links aufzuschreiben. Er
     endet mit einem Punkt. Dahinter stehen keine
     auszuwertenden Zeichen mehr. Es wird das
     englische Standard-Alphabet verwendet. Die
     Zeichen werden in schwarzer Farbe dargestellt.
     Alle Zeichen, auch Sonderzeichen und
     Leerzeichen außer dem Punkt sind zugelassen.
     Die Zeichen müssen sich radieren und
     überschreiben lassen. Der Punkt als
     Abschlusszeichen einer Zeichenkette bleibt an
     der gleichen Position stehen und kann auf dem
     Ausgabeblatt zum Erkennen des Zeichenketten-
     endes verwendet werden.
```

Die Schnittstelle wird festgelegt, bevor über die Programmierung als solche in größerem Umfang nachgedacht wird (*Ideen sind natürlich erlaubt*). Nach Festlegung der Schnittstelle trennen sich die Wege der verschiedenen Beteiligten, die nun unabhängig voneinander ihre Anwendungen bearbeiten. Da später alle Teile zu einem funktionierenden Ganzen zusammengefügt werden, müssen sich alle darauf verlassen, dass an der Schnittstellendefinition nichts mehr geändert wird beziehungsweise sich jeder buchstabengetreu an die Festlegungen hält. Es ist daher wichtig, so detailgetreu wie möglich zu formulieren und auch scheinbar unwichtige Einzelheiten festzulegen.

Die Beschreibung ist hier in zwei Teilbereich unterteilt, die die eigentlich ausgetauschten Objekte und die Eigenschaften der Objekte bezeichnen. Dies müssen Sie in Ihren Schnittstellenbeschreibungen nicht strikt nachvollziehen, sondern können alles unter dem Oberbegriff „Schnittstellenbeschreibung“ notieren. Ein Teil der festzulegenden Details wird Ihnen besonders bei den ersten Versuchen erst auffallen, wenn Sie bereits weiter in der Bearbeitung der Liste fortgeschritten sind. Ergänzen Sie in solchen Fällen die Beschreibung und kontrollieren Sie anschließend, ob sich die Änderungen auf bereits erstellte Teile der Anweisungsliste auswirkt.

Bei solchen Festlegungen ist es wichtig, sich über die Konsequenzen genau im Klaren zu sein (*Präzision der Sprache und des Denkens*). Hier wird beispielsweise festgelegt, dass die Ausgaben *„auf dem Eingabeblatt“* erfolgt. Das hat Konsequenzen für beide Partner: Der Nutzer darf Ihnen nun beispielsweise nicht das Originaldokument mit dem Satz geben, falls er es hinterher noch benötigt, sondern muss eine Kopie anfertigen, weil das übergebene Original während der Arbeit zerstört wird. Würde festgelegt, dass die Ausgabe auf einem neuen Blatt erfolgt, könnte er auch das Original übergeben und hinterher unversehrt zurückfordern. Außerdem muss er Ihnen ein Blatt geben, dass den Austausch von Zeichen ermöglicht. Zu dünnes oder dokumentenecht beschriebenes Papier ist ungeeignet, da nicht radiert werden kann. Auch für Sie hat das Konsequenzen, da nun nur noch bestimmte Algorithmen in Frage kommen. Falls Ihnen bei der Umsetzung in eine Anweisungsliste eine Vorgehensweise einfällt, die das Original unversehrt lässt, dem Nutzer aber ein weiteres Blatt aufnötigt: Vergessen Sie es, es passt nicht!

Weitere Festlegungen betreffen die Arten von Dokumenten, die bearbeitet werden sollen. Dokumente mit chinesischen Schriftzeichen dürfen Sie zurückweisen, wenn das englische Standardalphabet vereinbart wurde, ebenso Dokumente mit bunten Buchstaben. Eventuell müssen auch noch weitere Überlegungen einfließen, die hier nicht aufgezählt werden.

2.4 Die Konstruktion der Anweisungen

Nachdem Festlegung der Beziehungen zu anderen Akteuren muss nun überlegt werden, wie die Aufgabe zu bewältigen ist und welche Hilfsmittel benötigt werden. Da wir nicht mit „das letzte Zeichen“ oder „das nächste Zeichen“ arbeiten dürfen, müssen wir sie durchnummerieren, was auf kariertem Papier durch kleine Indexzahlen an den Kästchen natürlich einfach ist. Die Schnittstellenfestlegung „kariertes

Papier“ erhält so zusätzlichen Sinn.[9] Falls Sie im ersten Durchgang nicht auf diesen Trick gekommen sind: Jetzt wäre es an der Zeit, weiter oben nachzubessern. Vermeiden Sie aber auf jeden Fall Festlegungen, die dem Auftraggeber zusätzliche Arbeit aufbürden.

Da „bis zum Ende“ auch nicht verwendet werden darf, muss zunächst der Index des letzten Zeichens festgestellt werden. Da „suche den Punkt“ auch nicht zulässig ist, müssen wir wohl jedes Zeichen einzeln untersuchen, also „untersuche das Zeichen an Position Eins, wenn es kein Punkt ist, dann das Zeichen an Position Zwei, . . .“. Jetzt hätten wir schon ein erstes lauffähiges Programm konstruiert. Wir schreiben es mit nummerierten Sätzen auf.

```
<Festlegung der Hilfsmittel>
   Blatt "Satzlänge" für den Index des letzten
      Zeichens
<Anweisungsliste mit nummerierten Anweisungen,
 Teilliste zur Ermittlung der Zeichenanzahl>

Vereinbarung: Das Eingabeblatt erhält die Bezeichnung "input".
Mit "input[1]" bezeichnen wir das erste Kästchen.

1. Ist in "input[1]" ein Punkt? Falls JA
   1.1  Schreibe eine Null auf "Satzlänge",
   1.2  gehe zu "KAP2".
2. Ist in "input[2]" ein Punkt? Falls JA
   2.1  Schreibe eine Eins auf "Satzlänge",
   2.2  gehe zu "KAP2".
3. ...

KAP2: Auf dem Blatt "Satzlänge" steht die Anzahl
      der ermittelten Zeichen ohne den
      abschließenden Punkt.
      Es folgen nun die Befehle zum Sortieren.
```

Das Programm erfüllt alle Anforderungen: Die verbotenen Worte werden nicht verwendet, und bei einer Unterbrechung genügt die Angabe der Zeilennummer, die als nächste ausgeführt werden soll. Unsere einzige neue, aber offenbar zulässige Erfindung sind Unterlisten im Fall von Verzweigungen. Die Regel lautet offenbar: „Wenn die Prüfung positiv ausfällt, setze den Zeilenzeiger auf die erste Zeile der Unterliste, sonst auf die nächste Zeile der Hauptliste“. Das kann man sinnvoll auch nicht weiter unterteilen, das heißt Die JA/NEIN-Verzweigung ist elementar.

---

[9] Es sind auch völlig andere Arbeitsprinzipien denkbar, die einen anderen Schnittstellenaufbau erlauben. Beispielsweise können Sie auch eine Schablone vorgeben, die vor- und zurückbewegt werden kann, von der man aber im Zweifelsfall nicht weiß, welchen Index auf kariertem Papier das Zeichen hat, das unter der Schablone zu sehen ist.

Was hier beschrieben wird, ist also nur ein Weg, bei dem dann natürlich alles ineinander greifen muss. Aber lassen Sie sich davon nicht zu der Annahme verleiten, dass alles so sein muss, auch wenn Ihnen im Moment nichts anderes einfällt.

Dieses Programm läuft zwar, hat aber den Schönheitsfehler, dass wir nicht wissen, wie lang wir es machen sollen. Beschränken wir es auf 100 Schritte und jemand streicht aus dem großen Brockhaus alle Punkte bis auf den letzten und gibt uns das zum Sortieren, haben wir ein Problem, denn wir haben in der Schnittstelle keine Begrenzung vorgesehen, und ein nachträglicher Einbau ist unzulässig. Hier hilft nur eine Wiederholungsschleife weiter, wobei wir nun den Zeichenindex variabel machen müssen, in dem wir ihn auf dem Hilfsmittel notieren und in jedem Schleifendurchlauf verändern:

```
Vereinbarung: Ist "a" ein Merkzettel für Ganze Zahlen, so
bezeichnen wir mit "input[a]" das Kästchen mit dem Index,
der auf "a" abgespeichert ist.

1. Schreibe eine Eins auf "Satzlänge".
2. Ist das Zeichen in "input[Satzlänge]" Punkt, so gehe
   zu Schritt 5.
3. Addiere zum Inhalt von "Satzlänge" eine Eins und schreibe
   das Ergebnis auf "Satzlänge" zurück.
4. Gehe zu Schritt 2.
5. Ziehe eine Eins vom Inhalt von "Satzlänge"
   ab und speichere das Ergebnis auf "Satzlänge".
6. ... nun geht es an das Sortieren.
```

Nun haben wir ein kompaktes Programm, das die Anforderungen erfüllt und Sätze beliebiger Länge auswerten kann. Beachten Sie besonders Zeile 5 des Programms! Ohne diese Korrektur wäre nun auf dem Merkblatt die Eingabelänge einschließlich des Punktes gespeichert, das Ergebnis wäre also ein anderes als bei unserem ersten Versuch. Hier haben wir eine sehr typischen Problemfall der Programmierung vor uns, auf den besonders Anfänger leicht hineinfallen. Vom Programm ohne Zeile 5 „glaubt“ der Programmierer, dass es das korrekte Ergebnis liefern wird, prüft aber nicht nach, und gerade bei solchen Zählungen liegt der Ausgabewert dann oft um eine Einheit unter oder über dem korrekten Wert. Ein Probelauf der neuen Programmversion gegen die alte klärt den Fehler aber schnell auf und zeigt gleichzeitig, dass das wichtige Testen von Abläufen nicht erst am Ende erfolgen soll, wenn alles fertig ist, weil dann im Fehlerfall nicht klar ist, wo gesucht werden muss. Zu Testen ist jeweils nach Erreichen wichtiger Zwischenergebnisse, so dass bei der Erweiterung von korrekt arbeitenden fertigen Teilen ausgegangen werden kann.

An der Stelle ist eine Warnung notwendig, die insbesondere die Gruppenarbeit betrifft. Nehmen wir an, Sie sind bis zu Schritt Zwei gelangt, und nun kommt jemand auf die Idee, es wäre doch besser, in Schritt Eins eine Null auf „Satzlänge“ zu schreiben, damit der Punkt von vornherein nicht mitgezählt wird. Wenn man darauf eingeht, ist das Ergebnis meist eine Diskussion, an deren Ende keiner mehr weiß, was eigentlich gemacht werden sollte. Deshalb gilt:

```
Regel: Fertige Programmteile werden nicht in Frage
gestellt, so lange sie nicht definitiv fehlerhaft sind.
```

Hierbei hilft auch das regelmäßige Wechseln der Tastatur: Vielleicht ist der Kollege am Ende seines Zeitintervalls doch zu einem Ende gekommen, das Ihren Einwand erledigt – oder Sie führen ohne große Diskussion einfach die Korrekturen durch.

Den zweiten und komplizierteren Teil des Programms können wir zunächst in einer unzulässigen Form formulieren und dann schrittweise verfeinern.

```
Vereinbarung: Die implizite alphabetische
   Reihenfolge ist AaBbCc...Zz. Auf dieser
   Reihenfolge gelten die Relationen
   <, =, >, zum Beispiel  a<m, d=d, s>f

1. Vergleiche das erste Zeichen mit dem zweiten. Steht das
   weite Zeichen im Alphabet vor dem ersten, so tausche die
   Zeichen aus.
2. Wiederhole die Operation durch Vergleichen des dritten
   Zeichens mit dem ersten und fahre anschließend mit dem
   vierten usw. bis zum Ende fort.
   An der ersten Position steht nun das alphabetisch erste
   Zeichen der Zeichenkette.
3. Wiederhole nun die Schritte mit dem zweiten Zeichen als
   festgehaltenen usw. bis zum Ende.
```

Das ist natürlich alles hochgradig verboten, aber die Vorgehensweise ist geklärt. Fangen wir mit der Aufspaltung von Schritt 1. an. Um das Programm unterbrechbar zu machen, ist folgende Vorgehensweise notwendig (*beim ersten Mal benötigen Sie eventuell einige Versuche, bis Sie hier hin gelangen*):

```
1. Stelle einen Merkzettel "zt" bereit.
2. Falls "input[1]" > "input[2]"
   2.1  Schreibe das Zeichen in "input[1]" auf "zt".
   2.2  Schreibe das Zeichen in "input[2]" auf
        "input[1]".
   2.3  Schreibe das Zeichen auf Zeichentausch auf
        "input[2]".
```

Das Austauschen ist tatsächlich so kompliziert, denn Sie müssen die Buchstaben ausradieren und danach wieder in umgekehrter Reihenfolge hinschreiben. Wird nach eine Radieroperation unterbrochen, so ist der ausradierte Buchstabe verloren und das „Programm“ funktioniert nicht.

Die Wiederholung organisieren wir von vornherein als Schleife. Die Anzahl der Wiederholungen ist ja aus der ersten Anweisungsliste bekannt.

```
1. Stelle einen Merkzettel "zt" bereit.
2. Stelle einen Merkzettel "n1" bereit.
3. Schreibe eine 2 auf "n1".
4. Falls Zeichen "input[1]" > Zeichen "input[n1]"
   4.1  Schreibe Zeichen "input[1]" auf "zt".
   4.2  Schreibe Zeichen "input[n1]" auf "input[1]".
   4.3  Schreibe Zeichen auf "zt" auf "input[n1]".
```

```
5. Addiere eine Eins zum Inhalt von "n1" und schreibe das
   Ergebnis auf "n1".
6. Falls der Inhalt von "n1" kleiner ist als der Inhalt von
   "Satzlänge"
   6.1  Gehe zu 4.
```

Auch den letzten Teil können wir als Schleife realisieren, die um die gerade konstruierte herum gelegt wird. Der Inhalt von “n1“ zu Beginn eines inneren Schleifendurchlaufs wird dabei abhängig vom Wert der äußeren Schleifenzählvariablen. Auch bei der Formulierung der Abbruchbedingung für die äußere Schleife ist etwas Sorgfalt notwendig.

```
Stelle einen Merkzettel "zt" bereit.
1. Stelle einen Merkzettel "n1" bereit.
2. Stelle einen Merkzettel "n2" bereit.
3. Schreibe eine 1 auf "n2".
4. Erhöhe den Inhalt von "n2" um Eins und schreibe das
   Ergebnis auf "n1".
5. Falls Zeichen "input[n2]" > Zeichen "input[n1]"
   5.1 Schreibe Zeichen "input[n2]" auf "zt".
   5.2 Schreibe Zeichen "input[n1]" auf "input[n2]".
   5.3 Schreibe Zeichen auf "zt" auf "input[n1]".
6. Addiere eine Eins zum Inhalt von "n1" und
   schreibe das Ergebnis auf "n1".
7. Falls der Inhalt von "n1" kleiner ist als der Inhalt von
   "Satzlänge"
   7.1  Gehe zu 5.
8. Addiere eine Eins zum Inhalt von "n2" und schreibe das
   Ergebnis auf "n2".
9. Falls der Inhalt von "n2" kleiner ist als der  Inhalt von
   "Satzlänge", vermindert um Eins
   9.1  Gehe zu zu 4.
10. Gebe den Zettel "input" mit dem Ergebnis an den
    Auftraggeber zurück.
```

Damit haben wir unsere Aufgabe erledigt. Zur Prüfung sollte die Liste nun durch mehrere Bearbeiter und Beispiele getestet werden, um eventuell noch bestehende Lücken zu entdecken. Die Zwischenschritte und fehlerhaften Anweisungen sollten Sie nicht löschen, sondern kommentiert an das Ende Ihres Dokuments schieben. Bei weiteren Aufgaben können Sie sich so die Verfeinerungsschritte nochmals ins Gedächtnis rufen, und Fehler geben oft wertvolle Hinweise darauf, was man in Zukunft vermeiden oder beachten soll.

Teilaufgaben wie die Liste zur Bestimmung der Zeichenanzahl werden voraussichtlich auch in anderen Aufgaben zu erledigen sein. Sie können deshalb als separate Liste notiert und innerhalb der Liste für das Sortieren aufgerufen werden.

```
Vereinbarung: Bei Aufruf von Anweisungslisten aus einer
Liste heraus wird nach Beendigung der Unterliste das Programm
an der auf dem Merkzettel "Rücksprungmerker" notierten Stelle
fortgesetzt

5. Notiere den Schritt 7 für die Fortsetzung dieser Liste
   auf dem Zettel "Rücksprungmerker".
6. Rufe Liste "Zeichenanzahl" mit dem Originaldokument "input"
   als Eingabewert.
7. Schreibe den von der gerufenen Liste herausgegebenen Wert
   auf den Merkzettel "Satzlänge".
8. ...
```

Aufgabe. Vertiefen Sie nun Ihre erlangten Kenntnisse durch Erstellung von Anweisungslisten für folgende Arbeiten:

1. Kehren Sie die Reihenfolge der Zeichen eines Satzes um.
2. Prüfen Sie, ob und an welcher Stelle eine bestimmte Zeichenkette erstmals in einem Satz auftritt.
3. Löschen Sie eine bestimmte Zeichenkette in einem Satz.
4. Fügen Sie eine Zeichenkette an einer vorgegebenen Position in einen Satz ein.
5. Ersetzen Sie eine Zeichenkette in einem Satz durch eine andere.

Gehen Sie jeweils schrittweise vor uns präzisieren Sie zunehmend die Anweisungen. Beachten Sie: Nur wirklich elementare Anweisungen kön-nen direkt in die Programmiersprache übersetzt werden. Es existieren in den Programmiersprachen keine Zwischenstufen mit zunehmender Präzision/Detailgenauigkeit; die Erstfassung ist auch die Endfassung. Die normale Sprache bietet Ihnen die Möglichkeit, beliebige Zwischenstufen zu formulieren und sich dadurch an den für eine Programmiersprache notwendigen Denkprozess zu gewöhnen.

3 Der Anfang

Wenn Sie das letzte Kapitel sorgfältig bearbeitet haben, können Ihre Anweisungen nun mehr oder weniger wörtlich in die Anweisungen einer Programmiersprache übersetzt werden und sollten dann fast schon ein funktionierendes Programm ergeben.

Wenn ein solches Programm in eine Programmiersprache übersetzt wird, werden Sie feststellen, dass einige Konstrukte dort wesentlich kompakter definiert sind.[10] Um ein vorgreifendes Beispiel zu nennen, wird eine sich wiederholende Anweisungskette der Art

[10] Das gilt nur für „höhere“ Programmiersprachen. Wenn Sie maschinennah in Assembler programmieren, werden Sie feststellen, dass dies tatsächlich weitgehend eine Übersetzung ihrer Anweisungsliste in maschinenverständliche Kürzel darstellt.

```
1. setze „I" auf Null
2. ...
3. erhöhe den Inhalt von „I" um 1
4. ist „I" kleiner als 100, so gehe zu 2.
```

durch die kompakte Anweisung

```
for(i=0;i<100;i=i+1){ ... }
```

ersetzt, womit Sie auch gleich ein erstes Sprachelement kennengelernt haben. Die `for`-Anweisung besitzt ihren aus drei jeweils durch Semikolon getrennten Teilen bestehenden Steuerungsanteil zwischen den runden Klammern. Im ersten Teil wird die Initialisierung der in der Schleife verwendeten Größe vorgenommen. Der zweite Teil enthält die logische Prüfung, ob die Schleife durchlaufen werden soll. Diese wird vor jedem Durchlauf, also auch vor dem ersten, durchgeführt. Ist die Auswertung logisch TRUE, werden die zwischen den geschweiften Klammern stehenden Anweisungen ausgeführt. Nach der Ausführung wird der dritte Steuerteil, die Veränderung der Kontrollgröße, durchgeführt und anschließend wieder die logische Kontrolle durchgeführt.[11]

Diese Konstruktion hat nicht nur Bequemlichkeitsgründe. In Ihrer Anweisungsliste kontrollieren Sie erst am Ende, ob die Größe `I` den maximalen Wert überschritten hat, was aber nicht immer korrekt ist. Die kompakte Programmiersprachenanweisung sorgt dafür, dass der korrekte Bereich auch beim ersten Durchlauf nicht verletzt werden kann, führt also zu weniger Fehleranfälligkeiten.[12] Außerdem ist mit Befehlen wie `gehe zu ..` relativ viel Unfug anzustellen, wenn man sich nicht an bestimmte Regeln hält, weshalb die entsprechende (und auch real existierende) Programmiersprachenanweisung `goto` möglichst nicht genutzt werden sollte. Auch dabei hilft der Aufbau der Programmiersprache.

Um zügig voran zu kommen, empfiehlt sich folgende Arbeitsweise:

(a) Gehen Sie zunächst wie in Teil 0.1 vor, das heißt notieren Sie die Aufgabenstellung, ein Beispiel (*wenn möglich*) und erstellen Sie die Schnittstelle.
(b) Übertragen Sie die Schnittstelle in die Programmiersprache, nehmen Sie gegebenenfalls Korrekturen vor, die für die Verwendung der Programmiersprache notwendig sind.

---

[11] Damit ist diese Konstruktion eingeführt. Weitere Details folgen im Buch dann, wenn sie für die Umsetzung einer Aufgabe notwendig sind. Verglichen mit anderen Büchern, die viele Seiten für die Erklärung des for-Konstrukts, ist das natürlich lächerlich wenig, aber ich baue auch auf ein anderes Konzept: durch die vorhergehenden Übungen sollten Sie intuitiv erkennen können, was sich hinter dem Konstrukt verbirgt, was einiges an Erläuterungstext überflüssig macht, und durch praktische Übungen werden anfängliche Verständnisprobleme auch meist sehr schnell ausgemerzt. Verfallen Sie also nicht in Panik, wenn es zunächst so knapp weitergeht, sondern greifen Sie zum Entwicklungssystem und probieren Sie aus, ob alles so funktioniert, wie Sie meinen, es verstanden zu haben.

[12] Es existieren natürlich auch Programmiersprachenkonstrukte, die näher an der Anweisungsliste liegen, aber davon später.

(c) Entwicklen Sie einen Algorithmus und erstellen Sie schrittweise die Anweisungsliste. Verwenden Sie insbesondere bei Schleifen die kompakteren Sprachkonstrukte der Programmiersprache anstelle der Sprungbefehle an bestimmte Zeilen. Der Detailgrad kann schrittweise reduziert werden, wenn Ihre Sicherheit bei der Verwendung der Sprachstrukturen wächst.
(d) Schreiben und Übersetzen Sie das Programm. Testen Sie es anschließend.

Vermeiden Sie es auf jeden Fall, sich zu früh an den Rechner zu setzen. Sie müssen sich auf jeden Fall über die Vorgehensweise insgesamt im Klaren sein, bevor Sie die Programmzeilen erstellen. Bei der Umsetzung in eine Programmiersprache bleiben von den Anweisungslisten nur noch die letzten atomaren Anweisungszeilen übrig, und es ist sehr verlockend, sich sofort an den Rechner zu setzen und die schrittweise Problementwicklung auszulassen. Es ist überaus frustrierend, dann vor einem leeren Bildschirm zu sitzen oder der Lösung nicht näher zu kommen oder sogar völligen Unfug zu programmieren; die Ursache liegt meist in der unzureichenden Vorbereitung ohne Computer. Nehmen Sie also die Trockenübung ernst und versuchen Sie nicht, den Weg abzukürzen.[13]

3.1 Arbeit mit dem Entwicklungssystem

Für einen Anfänger ist es sinnvoll, mit einem Entwicklungssystem mit grafischer Oberfläche zu arbeiten, da solche Systeme viele Hilfen anbieten. Zwar beinhalten solche System auch jeweils eine Menge Feinheiten, die man erst im Laufe der Zeit kennenlernt, aber der Anfang ist glücklicherweise immer recht einfach, und die meisten Systeme sind einander so ähnlich, dass ein Umstieg relativ problemlos möglich ist. Für die meisten Systeme existieren Wikis oder FAQs für den ersten Einstieg sowie Nutzergruppen, in denen man auch schon mal Fragen stellen kann.[14]

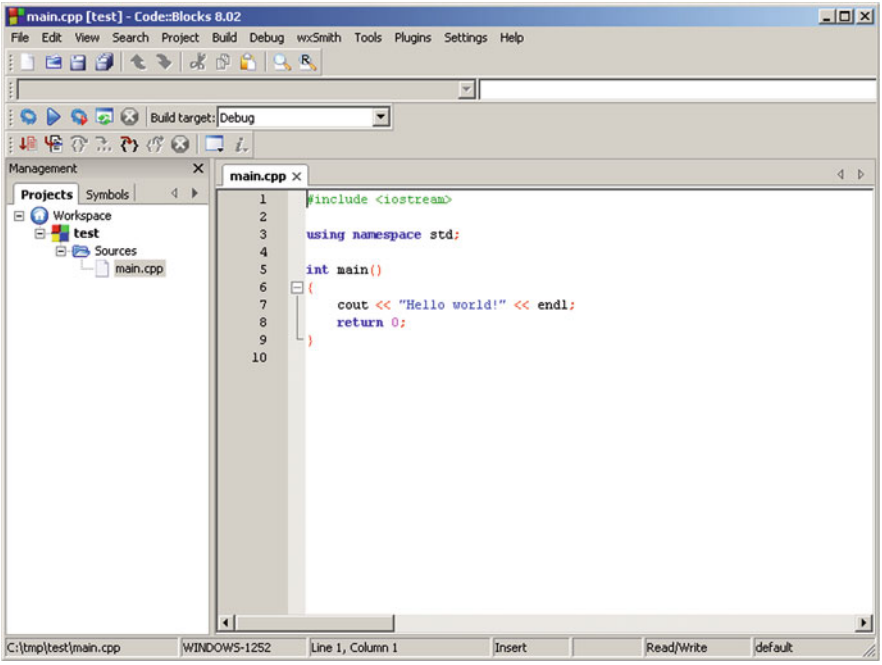
Der Code komplexer Anwendungen wird meist auf viele Programmdateien verteilt, die innerhalb eines so genannten Projektes verwaltet werden. Um zu beginnen, erstellen Sie ein neues Projekt des Typs „Konsolenanwendung“ mit dem Inhalt „Hallo Welt“-Anwendung. Am Besten machen Sie dies gleich zweimal, indem Sie ein C- und ein C++ - Projekt erstellen. Die Entwicklungssysteme bieten dazu meist eine Menüführung an, der Sie folgen können. Ändern Sie möglichst nichts an den Einstellungen des Systems, bevor Sie sich besser auskennen, da unter Umständen nach einer Änderung nichts mehr funktioniert. Wenn alles gut gegangen ist, sollten Sie etwa folgendes sehen:

---

[13] Die Empfehlung gilt später, wenn Sie die Sprache beherrschen, nur noch bedingt, denn im Gegensatz zu den Behauptungen der Theoretiker, die umfangreiche und vollständige Planungen vorschreiben, fällt vieles doch erst bei der Umsetzung und den damit verbundenen Tests auf. Doch davon später.

[14] ... für die man sich aber ein dickes Fell zulegen sollte, da man vermutlich mehrfach dafür angemacht wird, dass man bestimmte Fragen zum 397.881-sten Mal stellt.

Nach Drücken der Menuepunkte für Übersetzen und Ausführen (*finden Sie schon*) erhalten Sie ein schwarzes Fenster mit dem Text Hello world!, und das System funktioniert und Sie können mit weiteren Versuchen loslegen.



3.2 Erste Erkenntnisse

Bei diesem Prozess erhalten Sie mindestens eine Programmdatei. Der Inhalt verschiedener Programmdateien wird an ihren Dateierweiterungen erkannt:

```
*.c       C-Programmdatei
*.cpp     C++-Programmdatei (alternativ *.cc)
*.cxx     entspricht .cpp
*.h       C/C++-Datei mit Schnittstellenverein-
          barungen (alternativ *.hpp, bestimmte
          Dateien auch ohne Erweiterung)
```

Das von den meisten Entwicklungsumgebungen erzeugte Minimalprogramm sieht folgendermaßen im C++ Projekt aus:

```
int main(){
   cout << "Hello World!" << endl;
   return 1;
}
```

bzw. so im C Projekt,

```
int main(){
   prinft("Hello World!")
   return 1;
}
```

Beim Ausführen des Programms erhalten Sie ein schwarzes Fenster mit dem Text „Hello World!“ sowie der Information „Process returned 1 . . .“. Im Vorgriff auf die Vorstellung der Sprachsyntax kann man hieraus bereits folgende Schlüsse ziehen:

(a) Das Betriebssystem ruft eine Funktion mit Namen `main` auf. Die erste Zeile ist die Syntax für die Implementation einer Funktion.
(b) `cout` dient zur Ausgabe auf den Bildschirm, wobei verschiedene Ausgabeelemente durch **<<** verkettet werden können.
(c) `printf(..)` ist eine Funktion in C, die die gleiche Arbeit erledigt.
(d) Der zur Funktion gehörende Code wird durch **{** und **}** geklammert. Einzelne Anweisungen werden durch `;` abgeschlossen.
(e) `return` erlaubt die Rückgabe eines Wertes aus der Funktion.

> **Aufgabe.** Machen Sie, bevor Sie im Buch weitermachen, ruhig mal einige Versuche auf Verdacht. Geben Sie anderen Text aus, geben Sie andere Zahlen in der `return`-Anweisung zurück, schreiben Sie eine weitere Funktion `meinefunktion` durch Kopieren vor die `main`-Funktion und versuchen Sie, sie aus der `main`-Methode aufzurufen. Auch wenn nicht alles sofort funktioniert, das Eine oder Andere lässt sich intuitiv verstehen.

Als **Empfehlung** sei Ihnen noch auf den Weg gegeben, zunächst C++-Programme zu erzeugen, auch wenn Sie C-Anwendungen programmieren. In C++ stehen Ihnen alle Methoden und Möglichkeiten zur Verfügung, die auch C bietet. Der C++-Compiler ist allerdings etwas strenger als der C-Compiler, was gerade beim Anfang mithilft, Fehler schneller zu identifizieren. Außerdem stellt C++ einige Werkzeuge zur Verfügung, die etwas einfacher zu bedienen sind, wie beispielsweise `cout` anstelle von `printf`. Gerade bei Test ist man dafür oft dankbar. Wenn Sie sich später in einem fortgeschrittenen Kenntnisstadium in (systemnahe) Bereiche verirren, die tatsächlich reine C-Programmierung erfordern, ist es immer noch früh genug, den Compiler zu wechseln.

3.3 Weitere Arbeitsschritte

Da die Anweisungen in einer Programmiersprache recht knapp sind, ist es ratsam, zusätzliche Erklärungen zu hinterlegen. Diese werden durch Kommentarzeichen vom Programmkode abgesetzt:

```
/* Anfangskennung eines Kommentars. Alles bis
   zur Endekennung wird als Kommentar
   interpretiert, auch über mehrere Zeilen.
   Endekennung: */

// Kommentar bis zum Ende der Zeile, nur
// in C++-Umgebungen gültig
```

Sie werden feststellen, dass in einer Programmdatei im Editor die einzelnen Worte unterschiedlich farblich dargestellt werden. Kommentare lassen sich so leicht erkennen, ebenfalls fest definiert Worte der Programmiersprache und eigene Bezeichnungen.

Empfehlung. Verwenden Sie `/*..*/` nur in geschlossenen Kommentarblöcken außerhalb von Kodebereichen. Zur Kommentierung innerhalb von Anweisungsblöcken verwenden Sie – auch bei mehrzeiligen Kommentaren – besser **//** als Kommentartrenner. Neben einer bessere Übersichtlichkeit vermeidet man so auch Fehler bei Programmerweiterungen (*versehentliche Code-Auskommentierung*).

Anweisung werden grundsätzlich durch ein Semikolon abgeschlossen. Sehen Sie sich den „Hallo Welt“-Code an und vergleichen Sie die Struktur mit Ihren nummerierten Zeilen einer Anweisungsliste.

Schauen Sie sich nun den Beginn des Programms an, an dem weitere Zeilen zu finden sind. Programme machen oft intensiven Gebrauch von fertigen Unterliste. Um diese verfügbar zu machen, müssen die Schnittstellendateien zu Beginn einer Programmdatei genannt werden. Dies erfolgt durch den `include`-Befehl, wobei mit `<..>` geklammerte Dateien in Systemverzeichnissen, mit **“..“** geklammerte in Entwicklerverzeichnissen gesucht werden.[15] Bei der Übersetzung fügt der Compiler gewissermaßen an der Stelle der `include`-Anweisung die gesamte Datei in die aktuelle Programmdatei ein und arbeitet sie mit ab.

```
#include include <stdio.h>
#include "myinterface.h"
```

Weitere wichtige Hinweise für den Aufbau von `include`-Dateien finden Sie in den nächsten Kapiteln.

Einige Entwicklungssysteme bieten online-Hilfen an, die aktiv werden, wenn der Mauszeiger auf ein Schlüsselwort gezogen wird oder der Cursor im Schlüsselwort positioniert und dann die F1-Taste gedrückt wird. Probieren Sie dies an `cout` oder `printf(..)` aus. Im günstigen Fall erhalten Sie eine Beschreibung, wie der Begriff verwendet werden kann und welche Schnittstellendatei Sie mit dem `#include`-Befehl einbinden müssen.[16]

---

[15] Wo die Entwicklungsumgebung genau sucht, geht aus ihrer Beschreibung hervor. Manchmal muss man aber auch ein wenig in den Menues suchen oder eine der Newsgruppen durchsuchen.

[16] Ansonsten Laden Sie die C/C++-Bibliothekshandbücher aus dem Internet oder suchen Sie mit einer Suchmaschine nach weiteren Details. Das Einbeziehen dieser Informationsquellen ist ausdrücklicher Bestandteil des Buchkonzepts.

Sie können das Programm nun übersetzen und ausführen lassen. Weist das Programm Fehler auf, erhalten Sie in einem separaten Fenster eine Fehlerliste. Durch Anklicken einer Fehlermeldung wird die Programmzeile, in der der Fehler gefunden wurde, im Kodefenster angezeigt. Beseitigen Sie mit Hilfe des Textes der Fehlermeldung den Fehler und versuchen Sie es erneut mit einer Übersetzung.
Wichtig! Beseitigen Sie immer den ersten Fehler und übersetzen Sie dann erneut. Die restlichen Meldungen können Pseudofehler sein, die nur aufgrund des ersten Fehlers angezeigt werden, in Wirklichkeit aber gar keine Fehler sind. Durch solche Folgefehler sind die Fehlerlisten unvorhersehbar. Beseitigen Sie beispielsweise den ersten Fehler aus einer Liste mit acht Fehlern, kann es passieren, dass anschließend nur noch zwei Fehler übrig sind, die Fehleranzahl kann sich aber auch auch 15 erhöhen.
Hinweis. Gerne gemachte Fehler sind das Vergessen eines Semikolons, einer Klammer oder eines Anführungszeichens, so dass Anweisungen nicht geschlossen werden, Anweisungsblöcke vorzeitig enden oder Texte nicht beendet werden. Häufig passen die Fehlermeldungen des Compilers weder zu dem Fehlerbild noch beziehen sie sich auf die entsprechenden Programmzeilen. Konsultieren Sie deshalb auch 2–3 Zeilen oberhalb des angezeigten Fehlers und zählen Sie alle öffnenden und schließenden Klammern durch.

Der Debugger erlaubt (*nach erfolgreicher Übersetzung*) die schrittweise Verfolgung eines Programmablaufs und des Inhalts von Variablen. Ermitteln Sie, wie Sie Breakpoint im Programm setzen können (*das Programm läuft dann bis zu einer solchen Programmzeile durch und wird dann unterbrochen, so dass Sie sich die Zwischenergebnisse ansehen können*) und mit welchen Funktionstasten einzelne Schritte ausgeführt werden können.[17] Halten Sie den Aufwand aber in Grenzen: Debugger besitzen oft recht viele Möglichkeiten, die jeweils nur unter recht speziellen Bedingungen sinnvoll einsetzbar sind. Neben universellen Beispielen, deren Konstruktion nicht ganz einfach ist, führt das dazu, dass einige Optionen sehr selten genutzt werden und ihre Bedienung daher dem Vergessen anheim fällt und bei Bedarf von Neuem erarbeitet werden muss.

> **Aufgabe.** Freunden Sie sich mit der Entwicklungsumgebung an und spielen Sie mit den Funktionen herum. Normalerweise können Sie hier nichts Schlimmes anstellen, und notfalls löschen Sie das Projekt und erstellen ein neues.

Zugegeben, diese Einführung ist arg knapp. Andererseits gibt es zu jeder Entwicklungsumgebung kleinere oder größere Tutorials mit Beispielen, die genau auf das betreffende System zugeschnitten sind und sehr gut die Arbeitsmöglichkeiten demonstrieren. Schlimmstenfalls suchen Sie ein wenig im Internet; nach wenigen Minuten werden sie hinreichend Übungsmaterial vor sich haben. Sie müssen dies auch nicht zur Perfektion treiben. Es genügt, wenn Sie erste Erfolge sehen; der Rest kommt schon noch im Laufe der weiteren Studien.

---

[17] Oft ist bei den Entwicklungssystemen ein kurzes Tutorial vorhanden, was alles gemacht werden kann. Falls nicht, müssen Sie halt ein wenig herumexperimentieren.

4 Die Sprachelemente von C

4.1 Die Datentypen

In unseren Anweisungslisten haben wir jeweils als eine der ersten Handlungen in den verschiedenen Abschnitten Merkzettel für bestimmte Größen bereit gestellt. C stellt dem Anwender für diesen Zweck folgende Standarddatentypen zum Abspeichern von jeweils einem Wert des entsprechenden Typs zur Verfügung:

```
char  s;           /* 8-Bit-Wert, -128 .. 127 */
unsigned char us;  /* 8-Bit-wert, 0..255 */
short si;          /* 16-Bit-Wert,
                      -32.768..32.767 */
unsigned short us; /* 16-Bit-Wert, 0 .. 65.535 */
int i;             /* 32-Bit-Wert,
                      -2.147.483.648 ..
                      +2.147.583.647  */
unsigned int j;    /* dto., 0 .. 4.294.967.296 */
double r;          /* 8 Byte-Fließkommazahl mit ca.
                      16 Stellen Genauigkeit und
                      einem
                      Zahlenbereich von
                      10^(+/- 308)  */
```

Auf Ihrem System sind möglicherweise weitere Typen definiert. Informieren Sie sich im Hilfemenue Ihres Entwicklungssystems unter dem Stichwort „Datentypen“. Wir beschäftigen uns später ausführlich mit den Datentypen, so dass wir die Darstellung hier ziemlich knapp halten können.

Zur Deklaration einer Variablen eines bestimmten Typs wird die Typbezeichnung und ein Variablenname, der mit Buchstaben oder einem Unterstrich beginnen darf, eingetippt und das Ganze mit einem Semikolon abgeschlossen. Die Auswertung erfolgt größensensitiv, das heißt „hallo“ und „Hallo“ sind verschiedene zulässige Variablenbezeichnungen.

Wichtig! Das Vergessen eines Semikolons als Abschluss einer Anweisung ist einer der häufigsten Fehler. Wenn der Übersetzer (*Compiler*) Ihr Programm nicht übersetzt, so prüfen Sie auch die Zeilen vor der angezeigten Fehlerzeile, ob ein Semikolon fehlt!

Das Speichern von Daten erfolgt durch Zuweisung von Werten zu den Variablen, die auch durch Rechnungen erzeugt werden können. Vergleichen Sie folgende Anweisungen (*Semikolon nicht vergessen!*) mit den Zeilen Ihrer Anweisungslisten.

```
var i = 10, j, k;
j=15;
k=10*j+i;                    // k enthält nun den Wert 160
```

Konstante Werte können auf zwei Arten erzeugt werden: Als typisierte Konstante oder als Wertkonstante:

```
const double pi = 3.1415926;
#define PI  3.1415926

double pi_square;
pi_square=pi*PI;
```

Mit dem ersten Ausdruck wird eine Speicherstelle mit dem angegebenen Typ und Inhalt erzeugt, mit dem zweiten ein Textfeld, das der Compiler beim Übersetzen wie ein Textverarbeitungsprogramm überall dort einsetzt, wo er auf den Ausdruck `PI` stößt. Deklarierte Konstante wie `pi` können im Programm nicht verändert werden, das heißt Sie können bei der Verwendung sicher sein, dass der einmal festgelegte Inhalt noch gültig ist.

Mehrere zusammengehörende gleiche Daten wie beispielsweise die Zeichen eines Satzes können in Feldern gespeichert werden. Bei der Variablendeklaration wird die Feldgröße in eckigen Klammern mit angegeben:

```
char s[80] = "Hallo Welt!";
```

Diese Anweisung erzeugt ein Textfeld, das 79 Zeichen aufnehmen kann, in das zunächst im Rahmen einer Initialisierung aber nur die 11 Zeichen des Satzes eingetragen werden. Zeichenketten werden in C nicht mit einer Längenangabe versehen, sondern enden durch den Inhalt Null nach dem letzten Zeichen. Dies gilt aber nur für die Arbeit mit Zeichenketten (Strings). Bei anderer Verwendung der Felder ist eine Längenangabe notwendig (*Siehe Kap. 2*). Wegen der abschließenden Null können hier auch nur 79 Zeichen gespeichert werden, obwohl 80 Speicherplätze reserviert sind. Ähnlich lassen sich Felder anderer Typen erzeugen, wobei die volle Länge nutzbar ist. Auf die Unterschiede und Besonderheiten wird später ausführlich eingegangen. Der Zugriff auf ein Feld erfolgt unter Angabe des Index, wie wir es auch schon in unseren Anweisungslisten für den Zugriff auf einzelne Buchstaben eines Satzes vereinbart haben:

```
int a[10];
a[5] = 7;
```

Wichtig! Ein Feld der Größe n besitzt die gültigen Indizes $0..(n-1)$.

```
s[0] hat den Inhalt 'H'
s[1] hat den Inhalt 'a'
...
s[79] hat einen unbestimmten Inhalt, kann aber für
      die Speicherung von Werten verwendet werden
s[80] ist keine gültige Speicherstelle mehr
```

Datenfelder fangen in C grundsätzlich mit dem Index Null an und der Index n darf nicht mehr verwendet werden, weil der Speicherplatz bereits zu einer anderen Variablen oder zum Programm gehört. Sehr viele Fehler beruhen auf der Verletzung

dieser Grenzen und werden verharmlosend mit der Bezeichnung „Pufferüberlauf“ (buffer overflow) versehen.

Wird eine Feldvariable ohne eckige Klammern in einem Programm verwendet, so erhält sie die Bedeutung eines Zeigers auf den Beginn der Datentabelle. Man kann sich dies wie die Organisation einer Bibliothek vorstellen: `s[i]` bezeichnet die einzelnen Bücher in einem Regal Namens `s` . Der Regalname gibt den Ort an, an dem die Bücher zu finden sind. Neben dem reinen Namensteil von Feldern gibt es in C spezielle Ortsdatentypen, die Zeigervariablen auf Daten. Ortsdatentypen werden mit dem Typ von Daten deklariert, die an dem Ort zu finden sind, den der Zeiger bezeichnet, und entstehen aus den normalen Datentypen durch einen nachgestellten Stern:

```
int i;         /* eine Zahlenvariable */
int* pi;       /* eine Ortsvariable für eine Zahl */
```

Zur Verdeutlichung: `i` bezeichnet eine Speicherstelle im System, auf der ein ganzzahliger Zahlenwert gespeichert werden kann. `pi` bezeichnet eine Speicherstelle, an der der Ort einer Speicherstelle für ganzzahlige Zahlenwerte hinterlegt werden kann.

Feldvariablennamen und Zeigervariablen können ähnlich verwendet werden. Der Unterschied zwischen beiden ist, dass Feldvariablen auf existierenden Speicher verweisen, Zeigervariablen aber zunächst nicht. Bei folgendem Versuch sollte daher einiges schief gehen.

```
int a[10];
int* pa;
a[3]=10;      /* ok, Speicher vorhanden */
pa[3]=10;     /* Fehler, kein Speicher vorhanden */
```

Auf Zeigervariablen muss erst ein Ort, unter dem Speicher zur Verfügung steht, zugewiesen werden, bevor sie genutzt werden:

```
int a[10],i;
int* pa, * pb;
a[3]=10;      /* ok, Speicher vorhanden */
pa=a;         /* pa zeigt auf das Feld a */
pa[4]=10;     /* ok, Speicher vorhanden */
pb=&i;        /* pb zeigt nun auf den Speicher i */
pb[0]=1;      /* ok, i hat nun den Wert 1 */
pb[5]=4;      /* Fehler, i ist kein Feld ! */
```

Um von Variablen für einzelne Werte den Ort zu ermitteln, wird der Operator `&` eingesetzt. Anstelle von `pb[0]` kann auch eine andere Bezeichnung eingesetzt werden.

```
pb[0]=4;
*pb=4* *pb;    /* i hat nun den Inhalt 16 */
```

Der Stern vor dem Variablennamen hat also die gleiche Wirkung wie die Angabe des Index Null. Eine Besonderheit von Zeigern ist die Möglichkeit, mit ihnen Orte zu berechnen:

```
int a[10];
int *pa;
pa=a;
*pa=10;          /* a[0] hat nun den Inhalt 10 */
*pa = *pa + 10; /* a[0] hat nun den Inhalt 20 */
pa = pa + 5;             /* pa zeigt nun auf a[5]! */
*pa = 13;        /* pa[5] hat nun den Inhalt 13 */
```

> **Aufgabe.** Erstellen Sie nun selbst einige Beispielprogramme und probieren Sie aus, was Sie mit den Datentypen anstellen können, z.B. zuweisen von Werten, addieren, auf den Bildschirm ausgeben usw. Die Operatoren können auch geschachtelt werden. Versuchen Sie beispielsweise zu verstehen, was mit **p gemeint ist und wie man damit umgehen kann. Probieren Sie ein wenig herum und stellen Sie auch einige Versuche an, bei denen Sie bewusst gegen die Regeln verstoßen. Beobachten Sie die Wirkung (*manchmal wird nichts geschehen, aber das ist eher Zufall und sollte nicht beruhigen*).

Hilfe. Bei der einen oder anderen Anweisung wird Sie der Compiler darauf hinweisen, dass das so nicht geht. Sie können ihn jedoch mit einem kleinen Trick manchmal überreden, doch etwas zu akzeptieren:

```
int* pi;
int i;
// i=pi;                    // Fehlermeldung
i = (int) pi;               // so gehts
```

Schreiben Sie einfach in Klammern den Datentyp auf der rechten Seite einer Zuweisung vor die Variable, auf die der Compiler abbilden soll. Was genau dabei passiert, diskutieren wir späten; hier läuft erst mal der Compiler weiter und Sie können etwas sehen.

Hat ihr Compiler alles übersetzt, kann es passieren, dass sich das Programm während der Laufzeit verabschiedet. Beobachten Sie die Ausführung der Befehle beispielsweise im Debugger oder durch Ausgabe von Hilfszeilen und stellen Sie fest, an welcher Stelle der Laufzeitfehler geschieht. Drucken Sie mittels `cout` oder `printf(..)` alles mögliche aus, um die Wirkung ihrer Anweisungen zu studieren.[18]

Wichtig! Die Zeiger machen C zu einer recht vertrackten Angelegenheit, da vom Compiler relativ wenig Kontrollen durchgeführt werden und bei Unachtsamkeit entsprechend viel Unfug angerichtet werden kann. Einige Regeln die in den weiteren

[18] Wie `printf(..)` genau funktioniert, entnehmen Sie bitte dem C-Bibliothekshandbuch, das Sie sich aus dem Internet bereits beschafft haben sollten.

Kapiteln des Buches formuliert werden, helfen, hier die Übersicht zu behalten und korrekte Programme zu entwerfen.

Der sorglose Umgang mit Zeigern und der nachfolgende Frust, wenn die Anwendungen nicht das machen, was sie eigentlich sollen, hat C unrichtigerweise zu dem Ruf verholfen, man könne in dieser Sprache nicht sauber programmieren. Leute, die sich einen 400 PS-Wagen kaufen und nur die Gaspedalstellungen „Leerlauf“ und „Vollgas“ kennen, können sich aber im Stadtverkehr auch nicht richtig bewegen, und das liegt bestimmt nicht am Fahrzeug.

Die Arbeit mit Typen wird in den späteren Kapiteln noch ausführlich vertieft. Versuchen Sie aber, durch diese Übung ein Gefühl für Grundtypen, Zeiger und Felder zu bekommen.

4.2 Die Schnittstellendefinitionen

In einer Anweisungsliste haben wir zunächst den Namen und die Schnittstelle zwischen Programmierer und Nutzer festgelegt. Dies erfolgt in C durch Funktionsköpfe, wobei wir die diskutierten Datentypen einsetzen. Einen Funktionskopf haben Sie ja mit der Funktion `main()`, die vom Betriebssystem beim Programmstart aufgerufen wird, je bereits kennen gelernt.

Die Anweisungsliste zum Sortieren der Buchstaben eines Satzes benötigt die Ortsangabe des Satzes, an dem der Satz zu finden ist, und gibt außer dem sortierten Satz nichts zurück. Eine passende Schnittstelle unter Verwendung des zusätzlichen Datentyps `void`, der etwa „hier sind keine Daten“ bedeutet, sieht dann so aus:

```
void sort(char* input)
```

Hätten wir vereinbart, dass die Rückgabe auf einem getrennten Blatt erfolgt, kämen folgende Schnittstellen in Frage

```
/* Die Sortierfunktion stellt das neue Blatt zur
   Verfügung */
char* sort_1(const char* input)

/* Der Nutzer stellt der Sortierfunktion ein leeres
   Blatt zur Verfügung */
void sort_2(const char* input, char* output)
```

Beachten Sie die Vereinbarungen `char*` und `const char*` in der Parameterliste, je nachdem, ob der Inhalt verändert werden darf oder nicht. Das Schlüsselwort `const` erfüllt gleich zwei Funktionen: zum Einen signalisiert es dem Anwender, dass der Inhalt der übergebenen Variablen in der Funktion nicht verändert wird, zum Anderen sorgt es dafür, dass der Programmierer im Inneren der Funktion auch keine Anweisungen verwenden kann, die dies tun. Wir werden uns mit dieser Materie später noch ausführlich auseinander setzen.

Ein Funktionskopf besteht somit aus einem eindeutigen Funktionsnamen (*siehe Variablenbezeichnungen*), eine Liste von Übergabenparametern für die Ein- und

Ausgabe, die vom Nutzer zur Verfügung gestellt werden, und gegebenenfalls einem Rückgabewert, der von der Funktion erzeugt wird und im rufenden Programm auf einem Speicherplatz hinterlegt werden sollte.

```
char s[100]="Hallo Welt";
char* t;
t=sort_1(s);
```

In diesem Aufruf wird der Inhalt von **s** sortiert und auf einem neuen Ort gespeichert, der auf einer Ortsvariable gesichert wird.

Funktionen können in Programmen verwendet werden, so bald sie deklariert sind, das heißt der komplette Kopf mit einem Semikolon genannt wird. Jede Funktion benötigt eine Implementation, die ebenfalls mit dem vollständigen Kopf beginnt, nun aber in geschwungenen Klammern den Programmkode enthält.

```
void foo();        /* Deklaration der Funktion */
...
void foo(){...}    /* Implementation der Funktion */
```

Zu beachten sind folgende Regeln:

(a) Funktionsnamen müssen eindeutig sein, das heißt es darf in einem Programm höchsten eine Deklaration und eine Implementation geben.

(b) Deklaration und Implementation sind immer außerhalb der Kodeblöcke anderer Funktionen auf die Haupttextebene zu schreiben.

```
void foo(){              /* ok */
   ...
   void foo_2() { .. }   /* nicht erlaubt */
   ...
}//end function

void foo_3(){ ...        /* ok */
```

(c) Variablenbezeichnungen sind nur innerhalb des Kodeblocks und gegebenenfalls eingeschlossener Kodeblöcke gültig, in denen sie deklariert sind.

```
int i;         /* Hauptebene, globale Variable */

void foo(){
   int i;      /* möglicher logischer Konflikt */
   int j;      /* ok */
   ...
   {
      j=5;     /* ok, j bekannt */
   }
}//end function

void foo_2(){
   j=5;        /* Fehler, j unbekannt */
}//end function
```

```
void foo_3(){
   int j;      /* ok, unabhängig von foo() */
}//end function
```

Sie können auf der Haupttextebene deklariert werden, sind dann in allen folgenden Funktionen bekannt und bezeichnen immer den gleichen Speicherplatz. `i` ist eine solche globale Variable, auf die die Funktionen `foo_2` und `foo_3` zurückgreifen können. `foo` deklariert selbst eine eigene Variable des Namens `i`, die mit dem ersten `i` nichts zu tun hat.

Bei der Deklaration innerhalb einer Funktion verlieren sie ihre Gültigkeit am Funktionsende. Die Namen können mit neuen Speicherplatz in weiteren Funktionen wiederverwendet werden, wobei auch dann keine Konflikte entstehen, wenn sich die Funktionen gegenseitig aufrufen.

> **Aufgabe.** Implementieren Sie eine einfache Funktion mit einem Übergabe- und einem Rückgabetyp. Testen Sie die Wirkung des Schlüsselwortes const in den verschiedenen Typvereinbarungen im Funktionskopf und im allgemeinen Variablenteil. Untersuchen Sie auch die Verwendung von Feldern und Zeigern. Erstellen Sie eine Liste der von Ihnen getesteten Kombinationen und kommentieren Sie die Compiler- und Laufzeitergebnisse.

4.3 Bibliotheksfunktionen

In C sind relativ wenige Vorgänge direkt in der Sprache verankert. Die meisten Standardoperationen werden über Bibliotheksfunktionen abgewickelt, zu deren Bekanntmachung die so genannte Header-Dateien einzubinden sind. Normalerweise benötigt werden

```
#include <stdlib.h>
#include <stdio.h>
```

Wenn ein Compiler Funktionen nicht findet und Fehlermeldungen produziert, liegt dies oft daran, dass die notwendigen Headerdateien nicht eingebunden sind.

Welche Funktionen angeboten werden, ist teilweise systemspezifisch. Eine Reihe von Funktionen gehören zu der C-Standardbibliothek, aber auch diese können aber auf unterschiedlichen Systemen leicht abgewandelte Namen aufweisen (`write(..)` *oder* `_write(..)`) oder in einer anderen Headerdatei zu finden sein.[19] In den meisten Fällen lassen sich Funktionen für bestimmte Zwecke in einer

[19] Normen sind oft eine Garantie dafür, dass irgendwo 5 cm fehlen. Eine für Bauingenieure gültige DIN-Norm legt zum Beispiel die lichte Fensterhöhe in Küchen auf 80 cm fest, während die ebenfalls gültige DIN-Norm für Küchenbauer eine Arbeitsplattenhöhe von 85 cm vorschreibt. Wenn Sie ein Opfer dieses Widerspruchs geworden sind, ist ein Gang zum Anwalt sinnlos, denn jeder hat nach Norm gebaut, und das bereits seit jeweils mehr als 15 Jahren. Der letzte Fakt drückt das

Übersicht finden und durch eine abgestimmte Suche auch einbinden und verwenden. Andere Funktionen gehören nicht zum Standard, sind aber auch auf jedem System vorhanden, da ohne sie eine Arbeitsfähigkeit gar nicht gegeben wäre. Sie sucht man in den offiziellen Listen oft vergebens und sie sind auf unterschiedlichen Systemen auch unter verschiedenen Namen anzutreffen (*beispielsweise Funktionen für die Netzwerk- oder Dateiarbeit*). Ich spare mir deshalb hier längere und nur für bestimmte Systeme gültige Listen und verweise auf die Hilfesysteme Ihrer Entwicklungsumgebung bzw. die aus dem Internet ladbaren Bibliothekshandbücher.

Zwei Funktionen, von denen Sie mindestens eine bereits genutzt haben, sollen noch kurz andiskutiert werden, um die Orientierung im Hilfesystem zu erleichtern (*es sind Hilfen von Experten für Experten; Anfänger bedürfen zumindest einer Gewöhnungs- und Experimentierphase*).

```
/* Ein- und Ausgabe */
scanf ("%f",&r);
printf ("Das Quadrat von %f ist %f",r,r*r);
```

Die Funktion `scanf(..)` dient zur Eingabe von Werten mit Hilfe der Tastatur. Der erste Parameter ist ein Formatstring, der normalen Text sowie Anweisungen, wie Daten auf Variablen abzulegen sind, enthält. **%** `f` im Formatstring gibt an, dass die auf der Tastatur eingegebenen Daten als Fließkommazahlen zu interpretieren sind. Die Auswertung erfolgt vom ersten Zeichen, das kein Leerzeichen ist, bis zum nächsten Leerzeichen oder einem Zeilenvorschub, falls dieser vorher beobachtet wird. In der Parameterliste sind die Variablen genannt, auf denen die eingegebenen Werte abgespeichert werden sollen. Es können beliebig viele Parameter angegeben werden, jeder Parameter muss aber eine Formatangabe der Form **%** .. im Formatstring besitzen und die Formatangabe muss zum Typ der Variable passen. Die Variablen werden als Zeiger angegeben, da die Werte innerhalb der Funktion gelesen werden und dann in das rufende Programm zu transportieren sind.

Die Funktion `printf(..)` ist genauso aufgebaut und dient zur Ausgabe auf den Bildschirm. Da die Werte hier in der Funktion nicht äußerlich sichtbar verändert werden, erfolgt die Übergabe in diesem Fall als Wert und nicht als Zeiger.

Wichtig! Die meisten fehlerhaften Ein- und Ausgaben entstehen durch falsche Verwendung der Zeiger- oder Wertübergaben sowie durch falsche Formatangaben. Leseversuche mit Werten statt mit Zeigern können nicht zum Erfolg führen, eine Stringformatierung für eine Fließkommavariable führt auch zu merkwürdigen Ergebnissen. Sehen Sie lieber zweimal im Hilfesystem nach, bevor Sie eine Formatierung verwenden, von der Sie nur glauben, dass sie richtig ist.

Anmerkung. Es ist eine gängige Marotte in Anfängerkursen, längere `scanf`- und `printf`-Gymnastiken zu veranstalten, um den Anwender übertrieben höflich aufzufordern, doch nun bitte ein Zahl einzugeben. Halten Sie sich möglichst davon

---

typisch Deutsche aus: Jeder weiß mittlerweile, dass die beidseitige Befolgung der Vorschrift absoluter Schwachsinn ist, aber es ist ja nun mal Vorschrift, schwachsinnig vorzugehen und Müll zu produzieren.

fern und schreiben Sie Algorithmen, die über Initialisierungen mit Testdaten versorgt werden. Durch das Entwerfen primitiver Anwenderschnittstellen vergeuden Sie nur Zeit und lernen fast nichts. In der Praxis wird so etwas nicht benötigt.

Eine weitere Funktionengruppe ist für den Umgang mit Zeigervariablen vorgesehen. Speicherbereich für Zeiger kann alternativ zu Verweisen auf deklarierte Variable auch zur Laufzeit vom Betriebssystem geordert wer-den. Da geborgte Sachen nach Gebrauch auch wieder zurückgegeben wer-den sollten, bietet die Bibliothek zwei spezielle Funktionen an:

```
int* pi;
pi=(int*) malloc (12);// Speicher für drei ganze
                      // Zahlen wird vom Betriebs-
                     // system angefordert
pi[0]=1; pi[1]=2; pi[2]=3;
...
free (pi);            /* Speicher zurückgeben */
```

Ohne diese Technik wären die meisten Anwendungen kaum in der Lage, mehr als ein paar einfache Anwendungsfälle zu bearbeiten, die unzureichende Beherrschung dieser Technik ist aber auch eine Garantie dafür, Anwendungen zu schreiben, die besser nie das Licht der Welt erblickt hätten. Entsprechend viel Wert wird später auf die Vertiefung dieses Themas gelegt.

Für das fehlerfreie Übersetzen ist meist ein so genanntes Type-Casting, also eine manuelle Typzuweisung erforderlich. Die Funktion mallocbesitzt den Rückgabetyp `void*`, was soviel bedeutet wie „*Zeiger auf irgendetwas nicht näher Definiertes*“. Das ist natürlich etwas anderes als „*Zeiger auf einen ganzzahligen Wert*“, und damit der Compiler sicher sein kann, dass Sie das tatsächlich auch so meinen, müssen Sie das in der Form (`int*`) bestätigen. Durch diese vorangestellte Bestätigung wird der Typ manuell von `void*` auf `int*` geändert. Auch dazu später noch ausführliche Ergänzungen.

4.4 Weitere Teile des Programmkode

Bei Start eines Programms wird vom Betriebssystem die Funktion maingewissermaßen als Haupt-Anweisungsliste aufgerufen. Der Minimalkode für diese Funktion ist

```
int main(int args, char** argc){
    return 1;
}/*end function*/
```

Die Schnittstelle besteht aus einem als Wert übergebenen Zahlenwert sowie einem Zeiger auf einen Zeiger des Typs `char`. Da mit `char*` Zeichenketten (*Strings*) deklariert werden, handelt es sich insgesamt um ein Feld von Zeichenketten. Der erste Parameter gibt die Anzahl der Zeichenketten an.

Der Funktionskörper wird durch die Klammern gebildet und enthält zunächst nur die Rückgabe des Wertes Eins an das Betriebssystem. Wir können ihn nun durch weitere Anweisungen füllen, beispielsweise durch einen Ausgabebefehl zur Anzeige des ersten Parameters. Mit Hilfe der Bibliotheksbeschreibung finden wir

```
#include <stdio.h>
int main(int args, char** argc){
    printf("args hat den Wert %i\n",args);
    return 1;
}/*end function*/
```

Ein Probelauf des Programms liefert den Wert Eins; rufen wir das Programm von einem Kommandozeile mit mehreren Parametern auf, so ist der angezeigte Wert immer im eins höher als die Anzahl der Parameter. Zum Bearbeiten einer unbestimmten Anzahl von Objekten haben wir in den Anweisungslisten eine Schleife mit einem Hilfsmerker verwendet. Dies sieht in der Programmiersprache nun so aus:

```
#include <stdio.h>
int main(int args, char** argc){
    int i;     /* Hilfsmerker zum Zählen */
    printf("args hat den Wert %i\n",args);
    for(i=1;i<args;i=i+1){
        printf("Parameter %i hat den Wert %s\n",
                                        i,argc[i]);
    }/*endfor*/
    return 1;
}/*end function*/
```

Die Variablendeklaration der Zählvariable – hier `int i` – erfolgt zu Beginn eines Funktionsprogrammblocks. Die Variable ist in dem Block, in dem sie deklariert wird, und in allen Blöcken, die in diesem Block enthalten sind bekannt, hier also auch in dem von der Schleife gebildeten Block.

Die Schleife wird durch die `for`–Anweisung implementiert, die die von ihr abhängigen Anweisungen als eigenen Block enthält. Die `for`-Anweisung enthält drei Parameter:

```
for( <Initialisierung der Zählvariable> ;
     <Bedingung für den Abbruch der Schleife> ;
     <Verändern der Zählvariable> )
{ <Anweisungen> }
```

Bezüglich der Indizierung der Werte erinnern Sie sich an die Definition von Feldern in C: sie beginnen immer mit dem Index Null und enden einen Index vor der Feldgröße!

Wichtig! Viele Programmierfehler beruhen auf vergessenen Klammern. Meist wird ein Block durch „{“ geöffnet, aber nicht mehr durch „}“ geschlossen. Der

Compiler reagiert darauf mit den merkwürdigsten Fehlermeldungen. Wenn Sie Fehlermeldung an Stellen im Programm erhalten, an denen Sie absolut nichts finden können, kontrollieren Sie oberhalb dieser Stelle zunächst alle Blockbildungen auf korrekten Abschluss.

Bei dem Ausdruck stellen Sie fest, dass außer den Parametern auch der Programmname ausgedruckt worden ist. Wir ändern das Programm nun so, dass nur noch die Parameter ausgedruckt werden. Zusätzlich soll eine Meldung erscheinen, wenn kein Parameter angegeben ist, das heißt wir benötigen nun eine Verzweigung:

```
#include <stdio.h>
int main(int args, char** argc){
    int i;     /* Hilfsmerker zum Zählen */
    if(args>1){
        printf("Die Parameteranzahl ist %i\n",
                                          args-1);
        for(i=0;i<args;i=i+1){
           printf("Parameter %i hat den Wert %s\n",
                  i,argc[i]);
        }/*endfor*/
    }else{
        printf("Keine Parameter vorhanden.\n");
    }/*endif*/
    return 1;
}/*end function*/
```

Die Verzweigung `if(..)` in den Formen

```
if( <Bedingung> ){...}            /*oder*/
if( <Bedingung> ){...}else{...}
```

führt den ersten Anweisungsblock aus, wenn die angegebene Bedingung logisch Wahr ist. Im ersten Fall wird mit der nächsten Anweisung nach dem Block fortgesetzt, das heißt die folgenden Anweisungen werden unabhängig von der Bedingung immer ausgeführt. Im zweiten Fall wird der auf `else` folgende Block nur dann ausgeführt, wenn die Bedingung nicht erfüllt ist. Wir haben dadurch die Möglichkeit, nicht nur zusätzlichen Code auszuführen, sondern auch alternativen Code.

Zu Schleifen und Verzweigungen existieren neben `for` noch weitere Syntaxelemente:

```
i=0;                        i=0;
while(i<args){              do{
    ...                        ...
    i=i+1;                     i=i+1;
}/*endwhile*/               }while(i<args);
```

Die linke Konstruktion ist mit der `for`–Schleife identisch, die rechte nicht, da bei ihr der Programmblock mindestens einmal durchlaufen wird, weil die Prüfung der Abbruchbedingung erst am Ende stattfindet.

Wenn mehrere Bedingungen abhängig (!) voneinander auszuwerten sind, sind Konstruktionen wie `if(..){..}else if(..){..}` oft ziemlich umständlich zu implementieren. Einen Ausweg unter bestimmten Nebenbdegungen bietet

```
switch(args){
    case 0: ...
            break;
    case 1: ...
            break;
    default: ...
}
```

`switch` ist eine Erweiterung von `if(..)` und erlaubt die Verzweigung in beliebige viele, durch Zahlen bezeichnete Fälle. Das Argument von `switch()` muss ein Zahlentyp sein (`char, int`, nicht `double`!). Die mit der zugehörenden `case`–Deklaration beginnenden Anweisungen werden ausgeführt. Die Anweisung `break` führt dazu, dass hinter den `switch`–Block gesprungen wird. Fehlt sie, werden auch die Anweisungen der folgenden `case`–Blöcke ausgeführt.

Aufgabe. Sie sollten nun versuchen, die Anweisungslisten aus dem ersten Teil in kleine Programme mit Funktionen umzusetzen. Spielen Sie mit den hier aufgezählten Syntaxelementen und ändern Sie bewusst den Code durch Weglassen oder Hinzufügen von Schlüsselbegriffen. Beobachten Sie den Ablauf des Programms und die Werte der Variablen mit Hilfe des Debuggers. Mit ein wenig Experimentierfreude werden Sie die Verwendung der Begriffe schneller lernen, als wenn ich diesen Text um viele langatmig (*und -weilige*) Beschreibungen erweitert hätte. Es sollte Ihnen damit auch gelingen, Programme zu entwickeln, die auf dem Bildschirm das gewünschte Ergebnis produzieren.

Aufgabe. Beachten Sie das Ausrufezeichen am Wort „abhängig“ im Absatz über switch und erläutern Sie den Unterschied zwischen folgenden Kodeteilen:

```
Kodeteil 1:
if(a<b){   ...   }
if(a<c){   ...   }

Kodeteil 2:
if(a<b){   ...   }
else if(a<c){   ...   }
```

an einem Beispiel.

Sie dürfen nun mit Recht darauf stolz sein, dass es Ihnen in kurzer Zeit gelungen ist, funktionsfähige kleine Programme zu erstellen, auch wenn Ihnen diese nach dem Studium der weiteren Teile des Buches vermutlich nicht veröffentlichungsreif erscheinen. Beginnen Sie nun auch mit dem Studium der weiterführenden Kapitel. Auch wenn Ihre Programme bereits das gewünschte Ergebnis produzieren, ist die Wahrscheinlichkeit recht groß, dass noch einige grundsätzliche Fehler darin stecken, die Versuche, etwas komplexere Anwendungen entstehen zu lassen, zum Scheitern

verurteilen. Das wird sich aber schnell ändern, da Sie nun hinreichende Kenntnisse besitzen, den tiefer gehenden Ausführungen zu folgen beziehungsweise durch kleine Versuche jeweils Klarheit zu schaffen.

Aufgabe. Entwickeln Sie Funktionen für

- die Berechnung eines Skalarproduktes zweier Vektoren,
- die Berechnung der Nullstellen einer quadratischen Gleichung,
- die Berechnung von Funktionswerten von Polynomen.

Stellen Sie zuerst die mathematischen Formeln zusammen. Achten Sie auf korrekte Datentypen.[20]

4.5 Eigene Datentypen

In der Funktion `main(..)` werden zwei Parameter übergeben, die nicht getrennt werden dürfen. Dieser Fall zusammengehörender Daten tritt in der Praxis häufiger auf. Die Speicherung in eigenständigen Variablen erfordert einiges an Disziplin beim Programmierer bei der Verwendung und macht die Parameterlisten in Funktionsköpfen durch ihre Länge recht unübersichtlich. Spezielle Syntaxelemente erlauben die gemeinsame Gruppierung zusammengehörender Variablen und die Einführung anderer Namen für Datentypen zur leichteren Erkennung. Die beiden f `main(..)` lassen sich folgendermaßen fest aneinander binden.

```
typedef int     Argumentzahl;
typedef char**  Argumentfeld;

struct Argumente{
    Argumentzahl na;
    Argumentfeld af;
};
```
[21]

[20] Wer jetzt etwas düpiert ist ob der Mathematik, sollte seine Haltung überdenken. In vielen Bereichen der Informatik und der Programmierung läuft wenig ohne einen mathematischen Hintergrund. Außerdem haben mathematische Programmieraufgaben den Vorteil, dass Sie mit Lösen der Aufgabe beides intensiv verstanden haben: das Programmieren und die Mathematik.

[21] Achten Sie auf dieses Semikolon! An den geschweiften Klammern hinter **struct**-Definitionen MUSS ein Semikolon stehen! Man kann nämlich an eine **struct**-Definition auch direkt eine Variablendeklaration anschließen, was der Compiler aber nur dann korrekt auswerten kann, wenn alles durch ein Semikolon abgeschlossen wird. Nicht wenige Programmiereraspiranten verfallen bei einem Vergessen intensiven Selbstmordgedanken, blinder Zerstörungswut gegenüber dem System oder einem Berufswechsel zum Würstebräter in der Frittenbude, da der Compiler einen Fehler oft erst eine ganze Reihe Zeilen später lokalisiert und dort absolut nichts Unkorrektes zu finden ist.

Der Reihenfolge nach: das Schlüsselwort `typedef` erlaubt die Vergabe von Alias-Namen für bestehende Typen. Ein `int`-Typ bleibt auch mit dem Aliasnamen ein `int`-Typ, aus der Deklaration einer Variablen lässt sich so aber leichter ablesen, wofür sie verwendet wird. Auch Fehler durch erneutes Eingeben sehr komplexer Typen lassen sich so reduzieren.

Im Typ `struct` können mehrere Datentypen zu einem Globaltyp vereinigt werden. Der Globaltyp bekommt einen eigenen Namen, die inneren Bestandteile werden mit ihrem Typ und ebenfalls einem eigenen Namen aufgeführt. Innerhalb eines Typs müssen die Namen eindeutig sein; sie dürfen natürlich jederzeit an anderer Stelle wiederverwendet werden. Die Definition ist rekursiv, d.h. die inneren Bestandteile einer `struct` dürfen ebenfalls wieder vom Typ `struct` sein.

Damit können wir nun eine Ausgabefunktion implementieren, die einfacher aussieht als `main()`:

```
void print(struct Argumente a){
    int i;
    for(i=0;i<a.na;i=i+1){
        printf("Parameter %i hat den Wert %s\n",
                                    i,a.af[i]);
    }/*endfor*/
}/*end function*/

int main(int args, char** argc){
    struct Argumente arg;
    arg.na=args;
    arg.af=argc;
    print(arg);
    return 1;
}/*end function*/
```

Die Deklaration einer `struct`-Variable unterscheidet sich nicht von der Deklaration von Variablen anderer Typen. Der Zugriff auf die Teile einer `struct` im Programmkode erfolgt durch Angabe des Variablennamens und des Namens der Teilvariablen in der Struktur, getrennt durch einen Punkt. Sehen wir uns zum Abschluss noch die Syntax der Zeigerarbeit mit Strukuren an:

```
struct Argumente ar;
struct Argumente *arg;
arg=&ar;
(*arg).na=args;
arg->af=argc;
print(*arg);
```

Wenn Sie auf unerklärliche Fehlermeldungen stoßen, kontrollieren Sie Ihre **struct**-Vereinbarungen auf Vorliegen des Semikolons.

Der Zugriff auf die inneren Teile einer Struktur erfolgt bei einem Zeiger wahlweise durch (⋆arg). oder arg->, im Funktionsaufruf muss nun folgerichtig ⋆arg verwendet werden, da die Funktion einen Wert und keinen Zeiger als Parameter erwartet.

Strukturen und ihre Erweiterungen in C++, die Klassen, sind durchgängiges Thema im Hauptteil des Buches, so dass hier der Verweis auf die Syntax genügen mag.

Aufgabe. Implementieren Sie die Kodestücke und fügen Sie sie zu einem kleinen Programm zusammen. Spielen Sie ein wenig mit den Syntaxelementen herum und beobachten Sie die Fehlermeldungen des Compilers.

5 Die Sprachelemente von C++

C++ unterscheidet sich auf den ersten Blick recht wenig von C. Alle Syntaxelemente von C treten hier auch wieder auf, so dass man sich beim Übergang leicht zurecht findet. Allerdings werden Sie schnell feststellen, dass ein C++ – Compiler wesentlich empfindlicher reagiert als ein C- Compiler (*falls Sie in den Übungen zum letzten Kapitel nicht bereits mit dem C++ – Compiler gearbeitet haben, weil Ihre Dateien die Erweiterung .cpp statt .c aufwiesen*). In der weiteren Arbeit werden Sie feststellen, dass C++ eine eigenständige Sprache ist, die mit C einige gemeinsame Syntaxelemente besitzt.[22]

5.1 Überladen von Funktionen

Anders als in C dürfen in C++ Funktionsnamen mit anderen Parametern wiederverwendet werden. Betrachten wir dazu als Beispiel eine Funktion zum Quadrieren einer Zahl, die wir für Ganze Zahlen und ein weiteres Mal für Fließkommazahlen implementieren.

```
int square(int arg) { return arg⋆arg; }
double square(double arg) { return arg⋆arg; }
```

In C könnten wir nur eine der Funktionen implementieren, und wäre das zufällig die erste gewesen, so würde bei einem Einsatz ungefragt eine Fließkommazahl in eine Ganze Zahl umgewandelt, diese quadriert und das Ergebnis wieder in eine Fließkommazahl zurücktransformiert werden. Prüfen wir das mit den Zahlen 2, 5 und 7, so fällt nichts auf, aber wehe, das versucht jemand mit

```
r=square(sqrt(2.000));
```

[22] Man kann allerdings alles, was nicht die Ausführung echter Compileralgorithmen erfordert, komplett und sogar recht elegant auch in C implementieren – natürlich nicht in der Klarheit oder mit den Sicherheiten von C++.

C++ erlaubt die Implementation verschiedener Versionen der gleichen Funktion (*Überladen*). Beim Übersetzen sucht sich der Compiler die passende Funktion automatisch heraus. Existiert die zweite Methode nicht, so gibt der Compiler zumindest Warnungen heraus. Diese automatische Anpassung an die Erfordernisse werden wir später bei den Templates noch genauer untersuchen.

5.2 Überladen von Operatoren

Eine der großen Stärken von C++ ist die Möglichkeit, Operatoren zu überladen. Operatoren sind Verknüpfungen zwischen Variablen, also z.B.

```
=  +  -  *  /  %  +=  -=  *=  /=  &  &&
|  ||  <  <<  <=  .  ,  →  *
```

und weitere. Der Compiler weiß bei `a+b` natürlich, was er zu tun hat, wenn `a` und `b` vom Typ `int` sind, aber was ist beispielsweise mit `struct Argumente` aus einem der vorhergehenden Beispiele? Klar, weiß er nicht, aber er fasst die Addition implizit als Funktion mit einem speziellen Namen auf:

```
int a,b;
...a+b... ruft die Funktion int operator+(int,int) auf
```

Man braucht daher nur eine Funktion

```
Argumente operator+(Argumente,Argumente)
```

zu implementieren, und schon kommt der Compiler mit

```
Argumente a,b;
... a+b ...
```

zurecht. Bei der Ausführung eines Algorithmus mit einem anderen Datentyp muss man in C++ daher nur den Datentyp selbst austauschen und nicht den ganzen Programmcode wie in anderen Sprachen.

Natürlich sind hier einige Randbedingungen einzuhalten, damit dies auch korrekt funktioniert. Wir belassen es aber hier bei dieser Andeutung und werden dies genauer im Zusammenhang mit der Anwendungsprogrammierung, bei der dies gewinnbringend eingesetzt werden kann, diskutieren.

5.3 Namensbereiche

C++ berücksichtigt von vornherein die Entwicklung sehr großer Anwendungen. Hierbei kann es leicht passieren, dass Namen in einem anderen Zusammenhang wiederverwendet werden. Denken Sie an eine Datenbankanwendung und einen

komplizierten Algorithmus, die im Fehlerfall eine Funktion namens `Error()` aufrufen. Welche ist gemeint?

Hierzu gibt es das Schlüsselwort `namespace`, mit dem gewisse Anwendungsbereiche abgesteckt werden können.:

```
namespace MeinBereich {
   ... Funktionen, Datentypen, ...
}//end namespace
```

Innerhalb des `namespace`-Blockes können beliebige Funktionen und Datentypen abgelegt werden (*die main-Funktion steht in der Regel nicht in einem* `namespace`). Alle `namespace`-Blöcke mit dem gleichen Namen werden vom Compiler logisch zu einem Block zusammengefasst.

Um Funktionen oder Typen eines `namespace` verwenden zu können, muss man den `namespace` im nutzenden Programmbereich angeben. Dies kann auf zwei Arten erfolgen:

```
MeinBereich::foo();  // jeder Aufruf wird einzeln
                     // mit dem Namensbereich
                     // verknüpft
using namespace foo; // alle folgenden Namen und
                     // Typen werden mit dem
                     // Namensbereich verknüpft
```

Die globale Anweisung `using namespace ...` gilt für alle nach dieser Anweisungszeile folgenden Zeilen. Jeder Name wird überprüft, ob er im normalen Bereich oder im betreffenden `namespace` deklariert ist. Lässt sich das nicht mehr widerspruchsfrei auflösen, was etwa der Fall ist, wenn mehrere `using namespace`-Anweisungen aufeinander folgen und in den `namespace` gleiche Namen auftreten, muss die individuelle Auflösung erfolgen.

Wichtig! Die C++-Bibliotheken verwenden den `namespace std`. Auch wenn Sie Namensbereiche in Ihren Anwendungen (*noch*) nicht benötigen, müssen Sie bei der Verwendung von Bibkliotheken ein `using namespace std;` voraussstellen, damit der Compiler Ihr Programm übersetzen kann.

Namensbereiche lassen auch schachteln, so dass Sie auch eigene Bibliotheken aufbauen können, die mehrere Teile enthalten:

```
namespace Brands {
    namespace System {
```

Das Dereferenzieren wird dabei natürlich auch etwas aufwändiger

```
using namespace Brands::System
...
Brands::System::foo();
```

5.4 Klassen, Konstruktor, Destruktor

Das `struct`–Konzept wird in C++ erweitert, in dem zusammengehörende Daten zusätzlich mit Funktionen, die für ihre Bearbeitung zuständig sind, gruppiert und Zugriffsrechte auf die einzelnen Teile vergeben werden. Da-bei müssen zwei Funktionen zwangsweise immer aufgeführt werden, die für die korrekte Einrichtung einer Struktur und deren Zerstörung zuständig sind:

```
struct A {
    A();              /* Konstruktor */
    ~A();             /* Destruktor */
    char * s;
};//end struct
```

Die Zwangsfunktionen „Konstruktor“ und „Destruktor“ zeichnen sich dadurch aus, dass sie den gleichen Namen wie die Klasse selbst besitzen (*der Destruktor zusätzlich ein vorangestelltes* $\sim$) und keine Rückgabewerte aufweisen. Der Konstruktor dient zur Einrichtung der Attribute:

```
A::A(){
    s=strdup("Hallo Welt");
}//end constructor
```

Die Implementation der Methoden erfolgt durch Nennen der Klassenbezeichnung und des Methodennamens, getrennt durch `::`. Hier wird Speicher vom Betriebssystem besorgt, der im Destruktor wieder freigegeben wird

```
A::~A(){
    free(s);
}//end destructor
```

Der Aufruf von Konstruktoren und Destruktoren erfolgt automatisch durch das System, so dass immer dafür gesorgt ist, dass der Speicher ordnungsgemäß zugewiesen und auch wieder freigegeben wird.

```
void main(){
    A a;
    cout << a.s << endl;
}//end main
```

> **Aufgabe.** Implementieren Sie das Beispiel und verfolgen Sie den Programmablauf mit Hilfe des Debuggers und der Option „Verzweigen in Unterprogramme“. Ermitteln Sie, an welchen Stellen der Konstruktor oder der Destruktor ausgeführt wird, obwohl im Programmkode gar kein Aufruf dafür vorgesehen ist. An einigen Positionen sollte man besser nicht noch tiefer verzweigen, aber das finden Sie schon experimentell heraus.

Wie Sie bereits in den ersten Versuchen festgetstellt haben, übernimmt die Rolle der C-Ausgabefunktion `printf` in C++ die Ausgabeklasse `cout`. Statt sich mit

Formatstrings herum zu schlagen, werden alle auszugebenden Variablen oder Konstanten durch „<<“ getrennt hintereinander geschrieben. Hinter den Operatoren **<<** verbergen sich überladene Funktionen für die verschiedenen Standarddatentypen (*Funktionen für nicht-Standardtypen müssen selbst implementiert werden*), von der vom Compiler automatisch die jeweils benötigte in der richtigen Reihenfolge eingebunden wird. Wie das im Einzelnen geschieht, werden wir noch diskutieren. Die notwendigen `include`-Anweisungen für die Einbindung der Bibliotheken entnehmen Sie bitte dem Hilfesystem Ihrer Entwicklungsumgebung. Vergessen Sie nicht, die `using namespace`-Anweisung vor Ihr Programm zu schreiben.

In dieser Konstruktion könnte während der Lebensdauer des Objektes `a` jemand den String verändern und dadurch Fehler verursachen. Bei kritischen Variablen empfiehlt es sich daher, den fbieten, was in C++ durch folgende Klassendefinition möglich ist

```
class A {
public:
    A();              /* Konstruktor */
    ~A();             /* Destruktor */
    char const* str(){ return s;};
private:
    char * s;
};//end struct
```

Nur die unter `public` notierten Methoden oder Attribute können außerhalb der Klassenmethoden verwendet werden. Der String **s** ist nun privat, und der Compiler lässt einen Zugriff von anderen Stellen im Programm nicht zu. Mit der Funktion `str()` können wir auf den String zugreifen, aber nur in Anweisungen, die den String nicht verändern. Der String ist nun gekapselt.

> **Aufgabe.** Testen Sie auch dieses durch Implementieren der Klasse. Probieren Sie insbesondere verbotene Zugriffe aus.

Eine Klasse darf auch mehrere Konstruktoren enthalten (*aber immer nur einen Destruktor*), die das Objekt auf verschiedene Weise initialisieren. Beispielsweise kann damit ein anderer String in das Objekt eingesetzt werden:

```
class A {
public:
    A(){...}
    A(char const* c){...}
     ...
};

A a, b("neuer Text");
```

erzeugt nun eine Variable `a` mit dem alten Inhalt und eine Variable `b` mit einem anderen. Der Compiler sucht sich jeweils den passenden Konstruktion heraus und führt ihn aus.

Neben dem Standardkonstruktor `A()` wird in den meisten Fällen ein Kopier-Konstruktor benötigt, der eine Kopie eine Objektes herstellen kann. In unserem Fall sieht dies folgendermaßen aus (*der Kopf hat generell dieses Aussehen*).

```
A::A(A const& a){
    s=strdup(a.s);
}//end constructor
```

Aufgabe. Implementieren Sie die Konstruktoren sowie die folgende Funktion:

```
void foo(A a){
   cout << a.str() << endl;
}//end function
```

Verfolgen Sie mittels des Debuggers den Aufruf von Konstruktoren und Destruktoren, wenn diese Funktion aufgerufen wird.

Anmerkung. Viele Systeme „denken“ sich selbst Standard- und Kopierkonstruktoren sowie Zuweisungsoperatoren aus, wenn Sie nichts weiter angeben. In den meisten Fällen wird das auch gut gehen, nämlich dann, wenn für die Attribute ebenfalls gut definierte Konstruktoren oder Zuweisungsoperatoren existieren. Bei Pointer-Attribute geht das jedoch gnadenlos schief. Die grundsätzliche Empfehlung lautet daher, diese Elemente auf jeden Fall selbst anzulegen und nicht dem System zu überlassen (notfalls spart man sich zunächst die Implementierung auf). Das ist etwas mehr Arbeit, die sich aber bei größeren Projekten häufig später wieder bezahlt macht.

In einigen Fällen kann es unerwünscht sein, dass bestimmte Konstruktoren oder Operatoren dem Anwendungsprogrammierer zugänglich sind oder überhaupt existieren. In diesem Fall verschiebt man die Definition einfach in den privaten Teil der Klasse:

```
class A {
public:
   A(..);
...
private:
   A();
   ...
};
```

Wenn man anschließend die Implementation des Kodes „vergisst“, ist ein Aufruf des Standardkonstruktors nirgendwo mehr möglich.

5.5 Vererbung

Wenn in weiteren Klassen die Eigenschaften der Klasse A erneut benötigt werden, müssen die Teile von A nicht erneut implementiert, sondern können auf weitere Klassen durch „Vererbung“ übertragen werden:

```
class B: public A {
public:
    B();
    ~B();
    void print();
private:
    char* t;
};
```

Bei Deklaration einer Variable des Typs `B` wird nun zunächst der Konstruktor `A ()` durchgeführt, anschließend der Konstruktor `B` beim Abbau eines Objektes werden die Destruktoren in umgekehrter Reihenfolge durchlaufen. Um der erbenden Klasse `B` Zugriff auf das Attribut `s` der Klasse `A` zu verschaffen, ändern wir die Klassendefinition ein wenig:

```
class A {
public:
    A();                /* Konstruktor 1 */
    A(const char* c); /* Konstruktor 2 */
    ~A();              /* Destruktor */
protected:
    char * s;
};//end struct
// Implementation des Konstruktors  B
B::B(): A("Hallo Welt") {...}
```

Wie Sie experimentell unschwer verifizieren können, kann nun in den Methoden von `B` auf den String `s` zugegriffen werden. Außerdem haben wir über einen weiteren Konstruktor die Möglichkeit geschaffen, auch andere Strings in der Klasse unterzubringen, die sich damit zum Konstantenspeicher eignet. Grundsätzlich dürfen beliebig viele verschiedene Konstruktoren überladen werden, auch mit unterschiedlichen Zuordnungen zu den Schlüsselbegriffen `public, private` oder `protected`. Es bleibt aber immer bei einem einzigen Destruktor. Sind in einer Klasse mehrere Konstruktoren deklariert, so muss in den Konstruktoren erbender Klassen jeweils angegeben werden, welcher der Konstruktoren der vererbenden Klasse zu verwenden ist.

Methoden werden mitvererbt, können aber auch überschrieben werden. Das folgende Beispiel demonstriert dies:

```
class B: public A {
public:
   void f1();
   void f2();
...};
```

```
class A {
public:
```

```
   void f1();
   void f3();
...};

A a; B b;
a.f1();     // Methode A::f1
b.f1();     // Methode B::f1
a.f2();     // ungültig, f1 in A nicht bekannt
b.f2();     // OK
a.f3();     // Methode A::f3
b.f3()      // Methode A::f3
```

Aufgabe. Implementieren Sie ein Beispiel, in dem Sie die angegebenen Beziehungen durch entsprechende Ausdrucke auf dem Bildschirm kontrollieren können.

5.6 Zeigervariablen in C++

Aufgabe. Den Inhalt dieses Kapitels sollten Sie sorgfältig und detailliert während des Lesens implementieren und mittels Kontrollausgaben und Debugger testen. Das Life-Erlebnis sagt hier mehr als viele Worte.

Für die Arbeit mit Zeigern stellt C++ zwei neue Operatoren zur Verfügung, die die Funktionen `malloc(..)` und `free(..)` aus C ersetzen. Das ist notwendig, weil ja nicht nur der Speicherplatz für eine Zeigervariable verwaltet, sondern auch der Konstruktor und der Destruktor richtig aufgerufen werden muss, wovon die beiden C-Funktionen nichts wissen.

```
A* a;
a = new A("Hallo Welt");
...
a->print();
...
delete a;
```

Im Gegensatz zur `malloc`–Funktion von C weiß `new` auch ohne weitere Angaben, wie viel Speicherplatz die Variable benötigt, und ruft gleichzeitig den angegebenen Konstruktor auf. `delete` sorgt in gleicher Weise für den Aufruf des Destruktors, was `free(..)` alleine nicht könnte.

Von diesen Operatoren gibt es noch Sonderformen, auf die wir später kommen werden. Da Operatoren letztendlich nichts anderes als spezielle Funktionen mit festgelegten Aufrufschemata sind, können sie auch überschrieben werden. Doch auch davon später.

Nehmen wir nun an, in einer Anwendung werde in Anhängigkeit von äußeren Bedingungen entweder ein Objekt von `A` oder von `B` benötigt. Da Objekte von `B` mindestens die Eigenschaften von `A` aufweisen, ist dies mit Zeigern so realisierbar:

```
A * a;
if(...)
   a=new A();
else
   a=new B();
...
a->print();
delete a;
```

Zeigervariablen können immer mit Variablen erbender Klassen versehen werden. Das Umgekehrte – eine Zeigervariable des Typs

```
B* = new A();
```

mit einem Objekt der Elternklasse zu belegen – stößt natürlich auf den Unmut des Compilers und funktioniert nicht.[23] Die Verwendung der Zeigervariable ist natürlich auf das begrenzt, was in der Klasse `A` zur Verfügung steht, d.h. Funktionen, die erst in `B` definiert werden, können natürlich nicht aufgerufen werden. Die aufgerufenen Methoden stammen ebenfalls sämtlich aus `A`, was Anlass zum nächsten Teilkapitel gibt.

5.7 Virtuelle Vererbung

Diese Implementierung weist nämlich eine Unschönheit und einen echten Fehler auf. Trotzdem `new B()` aufgerufen wurde, wird die `print`-Methode der Klasse `A` ausgeführt, und auch beim Destruktor kommt nur der `A`-Destruktor zur Anwendung, was ein schwerer Fehler ist, denn von `B` angeforderter Speicherplatz wird nun nicht zurückgegeben. Um richtig zu funktionieren, muss in der Variablen Buch darüber geführt werden, von welcher Klasse sie abstammt. Wir korrigieren dies durch folgende Implementation von `A` :

```
class A {
public:
    A();              /* Konstruktor 1 */
    A(const char* c); /* Konstruktor 2 */
    virtual ~A();     /* Destruktor */
    virtual void print();
protected:
    char * s;
};//end struct
```

[23] Außer durch die bereits aus dem C-Teil bekannte explizite Typzuweisung (Casting) auf einen beliebigen (unpassenden) Typ vor dem Zuweisungsoperator. Die verantwortung dafür hat natürlich der Programmierer. Doch auch davon später mehr.

Normalerweise weiß der Compiler aus der Variablendeklaration, zu welchem Typ eine Variable (*auch eine Zeigervariable*) gehört, und setzt bei der Übersetzung direkt die Funktionsadressen ein (*dieses Verhalten haben wir zuvor beobachtet und in diesem Fall als unpassend erkannt*). Das Schlüsselwort `virtual` sorgt für ein anderes Verhalten:

Während des Konstruktoraufrufs wird für den Anwender unsichtbar neben den Attributen auch ein Zeiger auf eine Methodentabelle angelegt, der hier je nach verwendetem Konstruktor auf die Methodentabelle der Klasse `A` oder `B` verweist. In der Tabelle werden die als `virtual` deklarierten Funktionsadressen gesammelt. Statt nun beim Übersetzen bereits die Methode festzulegen, prüft das Programm bei virtuellen Methoden erst beim Funktionsaufruf während der Ausführung, welche Methodentabelle für das Objekt zuständig ist, und ruft dort die entsprechende Methode auf. Nicht als `virtual` deklarierte Methoden werden weiterhin während des Übersetzungsvorgangs ausgewählt und fest im Code eingetragen.

Die `virtual`-Deklaration muss nur bei der ersten Klasse, ab der virtuelle Methodenauswahl verwendet werden soll, angegeben werden und gilt dann automatisch für alle erbenden Klassen (und kann nicht mehr deaktiviert werden).

> **Aufgabe.** Die Methode `sizeof(..)` gibt die Speichergröße eines Objektes oder einer Klasse an. Testen Sie die Ausgabe für `sizeof(B)` mit und ohne virtuelle Methoden und interpretieren Sie das Ergebnis hinsichtlich der angesprochenen Methodentabelle.[24]

Wichtig! Bei `virtual`-Deklarationen ist der Destruktor **immer** mit einzubeziehen. Testen Sie die virtuelle Vererbung an einigen Beispielen mit Hilfe des Debuggers. Anwendungen werden wir in den weiteren Kapiteln diskutieren.

Mit der `virtual`-Deklaration kann auch eine bestimmte Programmierung erzwungen werden. Die Deklaration

```
class A {
   ...
   virtual void foo()=0;
   ...
```

erzeugt eine so genannte virtuelle Klasse. Hierbei handelt es sich um Basisklassen, mit deren Typnamen in Anwendungen gearbeitet wird, hinter denen aber noch gar kein Code vorhanden ist. Objekte der Klasse `A` können also vom Compiler gar nicht erzeugt werden. Programmierer müssen, um die Anwendungen verwenden zu können, von `A` erbende Klassen erzeugen und in diesen die Methode `foo()` implementieren. Zeigerobjekte dieser Klasse können dann wie oben beschrieben einem Objekt des Typs `A`$\star$ zugewiesen und bearbeitet werden.

---

[24] `sizeof(..)` ist keine Laufzeit-, sondern eine Compilzeitmethode, d.h. der Rückgabewert ist der statische Wert im Programmkode. `sizeof(..)` kann beispielsweise nicht auf das Ergebnis von `malloc(..)` angewandt werden. In Compileralgorithmen spielt `sizeof(..)` eine wichtige Rolle.

C++ bietet dem Klassenprogrammierer somit über die virtuelle Vererbung eine exakte Steuerungsmöglichkeit, welche Methoden bei einem Aufruf verwendet werden sollen.

```
A → B → C
B* b=new C(); A* a=b;
a->non_virtual_function();  // Funktion aus Klasse A
b->non_virtual_function();  // Funktion aus Klasse B
b->A::virtual_function();   // Funktion aus Klasse A
```

Die letzte Anweisung bietet über den so genannten Scope-Operator auch die Möglichkeit, das Schema bewusst zu umgehen.

Wo liegt der Sinn in dieser Differenzierung zwischen virtuellen und nicht virtuellen Methoden? Der Aufruf von nichtvirtuellen Methoden kann schneller erfolgen. Der Compiler kann bei der Programmübersetzung direkt die Sprungadresse der Methode einsetzen und muss es nicht dem laufenden Programm überlassen, diese erst aus einer Tabelle zu ermitteln. Bei optimiertem Programmkode ist der Compiler sogar in der Lage, den Funktionsaufruf komplett zu eliminieren, so dass die Programme sehr schnell werden können. Im Vergleich dazu existiert in Java ausschließlich die virtuelle Vererbung, d.h. eine Geschwindigkeitsoptimierung durch geschickte Definition von Funktionen ist nicht möglich.[25]

5.8 Mehrfachvererbung

C++ bietet darüber hinaus das Konzept der Mehrfachvererbung:

```
class A { ... };
class B: public A { ... };
class C: public A { ... };
class D: public B, public C { ... };
```

Die Klasse `D` erbt also sowohl die Eigenschaften der Klasse `B` als auch der Klasse `C`. Implementieren beide eine Funktion `foo()`, so muss dieser Konflikt durch explizite Benennung aufgelöst werden:

```
D d;
d.B::foo();
```

Im unserem Konstruktionsfall kommt noch ein weiteres Problem hinzu, denn `D` erbt gleich zweimal von `A`, wenn auch indirekt. Die Attribute von `A` sind gleich zweimal in der Klasse vorhanden, was ja durchaus auch sinnvoll sein kann, und bei einer

[25] Was ein wenig den Hype unverständlich macht, Java inzwischen auch in Bereichen, die sich durch hohen Rechenaufwand auszeichnen, zu verwenden. Ein nicht geringer Teil der Hardwarebeschleunigung wird dadurch aufgefressen.

Ansprache muss wie oben differenziert werden, ob man den aus `B` oder aus `C` stammenden Anteil von `A` meint. Soll `A` nur einmal in `D` instanziiert werden, so ist wieder das Schlüsselwort `virtual` dafür zuständig:

```
class A { ... };
class B: virtual public A { ... };
class C: virtual public A { ... };
class D: public B, public C { ... };
```

Mittels dieses Vererbungskonzeptes lassen sich recht komplexe Beziehungen erstellen.[26] Üben sie auch das zunächst einmal formal ein; wir werden später sehen, wie diese Konzepte in Anwendungen zu nutzen sind.

5.9 Referenzen

Neben Zeigern existiert in C++ noch das Konzept der Referenz, das beispielsweise anstelle eines Zeigers in einem Funktionsaufruf verwendet werden kann

```
void foo(A* a){
  *a=...
}
...
A a;
foo(&a);
...
```

```
void foo(A& a){
   a=...
}
...
A a;
foo(a);
...
```

Referenzen haben eine Reihe anderer Eigenschaften als Zeiger, die sich aber erst bei einem genaueren Studium erschließen und gewissen Parallelen zu Feldern/Zeigern in C aufweisen. Beispielsweise müssen Referenzen immer mit einer Instanz verknüpft sein und können nicht leer (*Nullzeiger*) implementiert werden. Im Zusammenhang mit Funktionen haben sie den Vorteil, wie Zeiger zu funktionieren, ohne dass man die Variablen auch wie Zeiger behandeln muss (*das gilt für den Aufruf und die Nutzung*). Auch hiermit werden wir uns in den weiteren Kapiteln intensiv beschäftigen.

5.10 Templates

Die Template-Technik hat sich zur eigentlichen zentralen Komponente von C++ entwickelt. Da sich der Hauptteil des Buches zu einem überwiegenden Teil mit Templates auseinandersetzt, halten wir uns hier wieder sehr kurz. Die Funktion zum Quadrieren sieht für alle Objekttypen, ob es sich nun um Zahlen oder Matrizen oder

[26] C++ geht hier weit über das hinaus, was die meisten anderen objektorientierten Sprachen zu bieten haben.

sonst was handelt, immer gleich aus. Es bietet sich an, sie typunabhängig als Muster zu implementieren.

```
template <class T>
T square(T t) { return t*t; }
```

Der Compiler braucht nun während der Übersetzung eines Aufrufs der Funktion nur zu überprüfen, ob für den Datentyp `T` die Multiplikation definiert ist, und kann sich nun eine spezialisierte Version der Vorlagenfunktion „ausdenken“ und übersetzen.

```
double r;
r=16.0;
r=square(r);
```

Der Compiler stellt hier fest, dass `r` vom Typ `double` ist, und setzt überall in der Implementation der Template-Funktion für `T double` ein. Er generiert unsichtbar für den Programmentwickler

```
double square(double r) { return r*r; }
```

Da die Multiplikation für Fließkommazahlen implementiert ist, kann der die Funktion korrekt übersetzen.

Neben Funktionstemplates sind auch Klassentemplates zulässig. Das folgende Beispiel erzeugt eine Struktur mit zwei Attributen, deren Typen über Template-Parameter festgelegt werden

```
template <class T1, class T2> class pair{
public:
   T1 first;
   T2 second;
};

pair<int,double> pid;
```

Zwischen Funktions- und Klassentemplates existieren eine Reihe wichtiger Unterschiede, die später ausführlich diskutiert werden.

Die Implementation muss jedoch nicht für alle Datentypen in dieser Form richtig oder optimal sein, weshalb das Templatekonzept Spezialisierungen erlaubt. Eine Spezialisierung besteht im einfachsten Fall darin, einen Templateparameter fortzulassen und dafür einen festen Typen in der Definition anzugeben:

```
template <class T1> class pair<T1,int>{
   T1 first;
   int second;
};
```

Bei Spezialisierungen spielt die Reihenfolge eine Rolle, in der der Compiler auf die Templates stößt.

```
template <class T> T square(const T& t);
int square<int>(int i);
template <class T> T square(const vector<T>& v);
```

Der Compiler arbeitet diese Liste von unten nach oben ab und verwendet die Implementation, die als erste mit der nutzenden Kodezeile vereinbar sind. Es können so beliebige Spezialisierungen implementiert werden. Durch Fortlassen der Implementierung für einen speziellen Typ lässt sich auch verhindern, dass der Algorithmus mit dem Typ verwendet werden kann.

Wie wir noch sehen werden, sind Templates ein sehr mächtiges Werkzeug, da der Compiler unsichtbar für den Programmentwickler sehr viele Prüfungen durchführt und die beste Möglichkeit auswählt beziehungsweise bei Nichterfüllbarkeit einer Anweisung eine Fehlermeldung generiert. Darüber hinaus bietet das Template-Konzept die Möglichkeit, Compiler-Algorithmen zu implementieren, d.h. nicht erst zur Laufzeit wird ein Algorithmus abgewickelt, der mit Daten umgeht, sondern bereits zur Übersetzungszeit können Algorithmen ablaufen, die Datentypen, Code und Daten generieren.[27]

6 Zur Arbeitsweise

Halten wir zum Abschluss ein paar Regeln fest, nach denen Sie bei der Entwicklung von Software vorgehen sollten:

(a) Sie entwickeln Funktionen oder Methoden, die von anderen Anwendungsteilen benötigt und aufgerufen werden – Sie entwickeln keine Programme!

    Achten Sie also darauf, dass andere Programmteile Ihren Code verstehen und verwenden können, nicht ein fiktiver Anwender. Niemand benötigt ein Programm, dass beispielsweise die Lösungen einer quadratischen Gleichung berechnet – ein Taschenrechner ist da um Größenordnungen günstiger – aber ein anderer Programmteil kann diese Daten mit kompletter Hintergrundinformation (*reelle NS, komplexe NS, usw.*) benötigen.

(b) Formulieren Sie exakt die Aufgabenstellung. Achten Sie auf Details und denken Sie nicht, dass etwas so oder so gemeint sein könnte – stellen Sie sicher, wie es gemeint ist! Schriftlich (*wie alles weitere*), nicht nur „im Kopf"!

(c) Stellen Sie eine Liste der grundsätzlichen theoretischen Ausdrücke zusammen, und zwar so, wie sie in der allgemeinen Theorie verwendet werden. Ein Polynom wird beispielsweise in der Form

$$P(x) = \sum_{k=0}^{n} a_k * x^k$$

[27] Mit Templates in den einfachsten Formen verstanden die Compiler schon Ende der 80er Jahre umzugehen. Nach Fertigstellen der endgültigen Standards Ende der 90er Jahre dauerte es allerdings nochmals etwa 4 Jahre, bis die ersten Compiler tatsächlich den Standard auch erfüllen können. In diesem Zusammenhang sollte man die „das können wir jetzt auch"-Argumente anderer Sprachen bewerten, die gar nicht erst für so etwas gebaut worden sind.

angegeben und nicht als

$$P(x) = a + b * x + c * x^2 + \ldots + i * x^n$$

Analysieren Sie die theoretischen Ausdrücke hinsichtlich der Informationen, die in Ihre Funktion hineingesteckt werden müssen, und der Informationen, die wieder heraus kommen.

(d) Stellen Sie die Schnittstellendaten der Funktion zusammen, und zwar mit den Datentypen, die die Theorie vorsieht! Vektoren besitzen in der Regel reelle Komponenten und nicht ganzzahlige!

Implementieren Sie die Schnittstelle (*Funktionskopf*) einschließlich des Rahmens für den Körper (*geschwungene Klammern. Grundsätzlich gilt: Öffnende und schließende Klammer editieren, bevor die erste Anweisung zwischen sie geschrieben wird. Immer auf korrektes Ein- und Ausrücken achten*).

(e) Stellen Sie Testfälle zusammen einschließlich der dazugehörenden Lösungen.

(f) Implementieren Sie die vollständige Testumgebung vor der Zielfunktion. Sie muss laufen und Ihnen zu jedem Zeitpunkt einen Testlauf zur Kontrolle der bis dahin realisierten Ergebnisse zu ermöglichen. Die Zielimplementation kann so in jedem Teil einzelnen getestet werden und Sie erhalten nicht am Schluss ein einzelnes unübersichtliches Ergebnis.

Die Testumgebung soll Kontrolldaten liefern und nicht höflich sein! Arbeiten Sie mit minimalem Aufwand.

(g) Arbeiten Sie nun erst schrittweise die Algorithmen aus. Implementieren Sie die Schritte einzeln und lassen Sie zwischendurch Tests ablaufen, ob der bis dahin entwickelte Code das richtige Ergebnis liefert.

Hierbei kann es natürlich passieren, dass der eine oder andere neue Gesichtspunkt auftaucht, den Sie in c) oder d) noch nicht berücksichtigt haben. Ergänzen Sie Ihre Theorie, gehen Sie zu den entsprechenden Entwicklungsschritten zurück und arbeiten Sie sich erneut nach vorne durch (*Zusammenstellung von Testdaten*, *Wiederholung der Tests für jeden einzelnen Schritt usw.*).

(h) Wenn alles fertig ist: Freuen Sie sich, trinken Sie ein Bier, einen Kaffee oder machen Sie sonst was, nach dem Ihnen ist, und gehen Sie anschließend an die nächste Aufgabe.

Seien Sie kreativ. Probieren Sie neue Konstrukte oder Ideen aus, nehmen Sie die Hilfen des Systems ausgiebig in Anspruch und nutzen Sie, wenn möglich, auch das Internet. Kurze geschickte Anfragen an eine Suchmaschine können Lösungsansätze liefern. Öffnen Sie neben Ihrem Arbeitsprojekt auch ein Testprojekt, in dem Sie alles möglich ausprobieren und wieder löschen können, ohne Unheil anzurichten.

Lösen Sie Ihre Aufgaben kurz und pragmatisch. Wenn die Zukunftsperspektive Ihrer Anwendung nicht ausdrücklich solche Umstände erkennen lässt, ist es kontraproduktiv, wenn bildlich gesprochen die Buchung einer Zugverbindung von Köln nach Düsseldorf über Frankfurt, Dresden und Hamburg führt und in Duisburg endet und Sie zusehen müssen, ob Sie von da einen Bus erwischen, und das nur, weil

sich die Bausteine eines von Ihnen verwendeten Framworks nur in dieser Weise kombinieren lassen.

Anmerkung. Je nach Projekt muss der eine oder andere Punkt dieser Liste nicht immer 100%-ig passen, aber meist stimmt die Vorgehensweise. Sie können natürlich auch was anderes machen, sollten dann aber auf Beschwerden verzichten, falls es nicht so gut klappt.