HANSER

Leseprobe

zu

„C++ programmieren“

von Ulrich Breymann

Print-ISBN: 978-3-446-46386-8
E-Book-ISBN: 978-3-446-46551-0
E-Pub-ISBN: 978-3-446-46470-4

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-46386-8
sowie im Buchhandel

Inhalt

Vorwort **23**

Teil I: Einführung in C++ **25**

1 Es geht los! **27**
1.1 Historisches 27
1.2 Arten der Programmierung 28
1.3 Werkzeuge zum Programmieren 29
1.4 Das erste Programm 30
1.5 Integrierte Entwicklungsumgebung 36
1.5.1 Das erste C++-Projekt mit Code::Blocks 37
1.5.2 Xcode 39
1.6 Einfache Datentypen und Operatoren 40
1.6.1 Ausdruck 40
1.6.2 Namenskonventionen 41
1.6.3 Ganze Zahlen 41
1.6.4 Reelle Zahlen 49
1.6.5 Konstanten 53
1.6.6 Zeichen 54
1.6.7 Logischer Datentyp bool 57
1.6.8 Regeln zum Bilden von Ausdrücken 59
1.6.9 Standard-Typumwandlungen 60
1.7 Gültigkeitsbereich und Sichtbarkeit 62

1.7.1 Namespace std ... 63
1.8 Kontrollstrukturen ... 64
1.8.1 Anweisungen ... 64
1.8.2 Sequenz (Reihung) ... 66
1.8.3 Auswahl (Selektion, Verzweigung) ... 66
1.8.4 Fallunterscheidungen mit switch ... 72
1.8.5 Wiederholungen ... 75
1.8.6 Kontrolle mit break und continue ... 82
1.9 Benutzerdefinierte und zusammengesetzte Datentypen ... 84
1.9.1 Aufzählungstypen ... 85
1.9.2 Strukturen ... 87
1.9.3 Der C++-Standardtyp vector ... 89
1.9.4 Zeichenketten: der C++-Standardtyp string ... 94
1.9.5 Container und Schleifen ... 97
1.9.6 Typermittlung mit auto ... 99
1.9.7 Deklaration einer strukturierten Bindung mit auto ... 101
1.9.8 Bitfeld und Union ... 101
1.10 Einfache Ein- und Ausgabe ... 104
1.10.1 Standardein- und -ausgabe ... 104
1.10.2 Ein- und Ausgabe mit Dateien ... 107
1.11 Guter Programmierstil ... 111
1.11.1 Programmierrichtlinien – die C++ Core Guidelines ... 112
1.11.2 Werkzeug zur Code-Formatierung ... 112

2 Programmstrukturierung ... 113
2.1 Funktionen ... 114
2.1.1 Aufbau und Prototypen ... 114
2.1.2 Gültigkeitsbereiche und Sichtbarkeit in Funktionen ... 116
2.1.3 Lokale static-Variable: Funktion mit Gedächtnis ... 117
2.2 Schnittstellen zum Datentransfer ... 118
2.2.1 Übergabe per Wert ... 119
2.2.2 Übergabe per Referenz ... 121
2.2.3 Gefahren bei der Rückgabe von Referenzen ... 123
2.2.4 Vorgegebene Parameterwerte und unterschiedliche Parameterzahl ... 124
2.2.5 Überladen von Funktionen ... 125
2.2.6 Funktion main() ... 127
2.2.7 Beispiel Taschenrechnersimulation ... 128

2.2.8 Spezifikation von Funktionen ... 133
2.2.9 Reihenfolge der Auswertung von Argumenten ... 133
2.3 Makros ... 133
2.3.1 #include ... 134
2.3.2 #define, #if, #ifdef, #ifndef, #elif, #else, #endif ... 134
2.3.3 Vermeiden mehrfacher Inkludierung ... 135
2.3.4 __has_include ... 136
2.3.5 Textersetzung mit #define ... 137
2.3.6 Umwandlung von Parametern in Zeichenketten ... 139
2.3.7 Verifizieren logischer Annahmen zur Laufzeit ... 139
2.3.8 Verifizieren logischer Annahmen zur Compilationszeit ... 140
2.3.9 Fehler ohne Programmabbruch lokalisieren ... 140
2.4 Modulare Programmgestaltung ... 141
2.4.1 Projekt: Mehrere cpp-Dateien bilden ein Programm ... 142
2.4.2 Übersetzungseinheit, Deklaration, Definition ... 144
2.4.3 Dateiübergreifende Gültigkeit und Sichtbarkeit ... 146
2.5 Namensräume ... 147
2.5.1 Gültigkeitsbereich auf Datei beschränken ... 150
2.6 inline-Funktionen und -Variablen ... 151
2.7 constexpr-Funktionen ... 152
2.7.1 Berechnung zur Compilationszeit mit consteval ... 154
2.8 Rückgabetyp auto ... 155
2.9 Funktions-Templates ... 156
2.9.1 Spezialisierung von Templates ... 158
2.9.2 Einbinden von Templates ... 159
2.10 C++-Header ... 162
2.10.1 Einbinden von C-Funktionen ... 163
2.11 Module ... 164

3 Objektorientierung 1 ... 167
3.1 Datentyp und Objekt ... 169
3.2 Abstrakter Datentyp ... 169
3.3 Klassen ... 171
3.3.1 const-Objekte und Methoden ... 174
3.3.2 inline-Elementfunktionen ... 175
3.4 Initialisierung und Konstruktoren ... 176
3.4.1 Standardkonstruktor ... 176

3.4.2 Direkte Initialisierung der Attribute ... 177
3.4.3 Allgemeine Konstruktoren ... 178
3.4.4 Kopierkonstruktor ... 181
3.4.5 Typumwandlungskonstruktor ... 184
3.4.6 Konstruktor und mehr vorgeben oder verbieten ... 186
3.4.7 Einheitliche Initialisierung und Sequenzkonstruktor ... 187
3.4.8 Delegierender Konstruktor ... 189
3.4.9 constexpr-Konstruktor und -Methoden ... 190
3.5 Beispiel Rationale Zahlen ... 194
3.5.1 Aufgabenstellung ... 194
3.5.2 Entwurf ... 195
3.5.3 Implementation ... 198
3.6 Destruktoren ... 203
3.7 Wie kommt man zu Klassen und Objekten? Ein Beispiel ... 205
3.8 Gegenseitige Abhängigkeit von Klassen ... 210

4 Zeiger ... 213
4.1 Zeiger und Adressen ... 214
4.2 C-Arrays ... 218
4.2.1 C-Array, std::size() und sizeof ... 219
4.2.2 Initialisierung von C-Arrays ... 220
4.2.3 Zeigerarithmetik ... 220
4.2.4 Indexoperator bei C-Arrays ... 222
4.2.5 C-Array iterieren ... 222
4.3 C-Zeichenketten ... 223
4.3.1 Schleifen und C-Strings ... 226
4.4 Dynamische Datenobjekte ... 229
4.4.1 Freigeben dynamischer Objekte ... 232
4.5 Zeiger und Funktionen ... 235
4.5.1 Parameterübergabe mit Zeigern ... 235
4.5.2 C-Array als Funktionsparameter ... 236
4.5.3 const und Zeiger-Parameter ... 238
4.5.4 Parameter des main-Programms ... 239
4.5.5 Gefahren bei der Rückgabe von Zeigern ... 239
4.6 this-Zeiger ... 240
4.7 Mehrdimensionale C-Arrays ... 242
4.7.1 Statische mehrdimensionale C-Arrays ... 242

4.7.2 Mehrdimensionales C-Array als Funktionsparameter ... 243
4.7.3 Dynamisch erzeugte mehrdimensionale C-Arrays ... 246
4.7.4 Klasse für dynamische zweidimensionale Arrays ... 249
4.8 Binäre Ein-/Ausgabe ... 255
4.9 Zeiger auf Funktionen ... 258
4.10 Typumwandlungen für Zeiger ... 262
4.11 Zeiger auf Elementfunktionen und -daten ... 263
4.12 Komplexe Deklarationen lesen ... 264
4.12.1 Lesbarkeit mit typedef und using verbessern ... 265
4.13 Alternative zu rohen Zeigern, new und delete ... 267

5 Objektorientierung 2 ... 271
5.1 Eine String-Klasse ... 271
5.1.1 friend-Funktionen ... 277
5.2 String-Ansicht ... 278
5.3 Typbestimmung mit decltype und declval ... 280
5.4 Klassenspezifische Daten und Funktionen ... 284
5.4.1 Klassenspezifische Konstante ... 288
5.5 Klassen-Templates ... 291
5.5.1 Ein Stack-Template ... 291
5.5.2 Stack mit statisch festgelegter Größe ... 293
5.6 Code Bloat bei der Instanziierung von Templates vermeiden ... 295
5.6.1 extern-Template ... 295

6 Vererbung ... 297
6.1 Vererbung und Initialisierung ... 302
6.2 Zugriffsschutz ... 303
6.3 Typbeziehung zwischen Ober- und Unterklasse ... 306
6.4 Oberklassen-Schnittstelle verwenden ... 307
6.4.1 Konstruktor erben ... 308
6.5 Überschreiben von Funktionen in abgeleiteten Klassen ... 310
6.5.1 Virtuelle Funktionen ... 311
6.5.2 Abstrakte Klassen ... 315
6.5.3 Virtueller Destruktor ... 320
6.5.4 Vererbung verbieten ... 322
6.5.5 Private virtuelle Funktionen ... 324
6.6 Probleme der Modellierung mit Vererbung ... 325
6.7 Mehrfachvererbung ... 328

6.7.1 Namenskonflikte ... 331
6.7.2 Virtuelle Basisklassen ... 332
6.8 Standard-Typumwandlungsoperatoren ... 335
6.9 Typinformationen zur Laufzeit ... 338
6.10 Private- und Protected-Vererbung ... 340

7 Fehlerbehandlung ... 343
7.1 Ausnahmebehandlung ... 345
7.1.1 Exception-Spezifikation in Deklarationen ... 348
7.1.2 Exception-Hierarchie in C++ ... 349
7.1.3 Besondere Fehlerbehandlungsfunktionen ... 351
7.1.4 Arithmetische Fehler/Division durch 0 ... 352
7.2 Speicherbeschaffung mit new ... 354
7.3 Exception-Sicherheit ... 355

8 Überladen von Operatoren ... 357
8.1 Rationale Zahlen – noch einmal ... 359
8.1.1 Arithmetische Operatoren ... 359
8.1.2 Ausgabeoperator << ... 361
8.2 Eine Klasse für Vektoren ... 363
8.2.1 Index-Operator [ ] ... 366
8.2.2 Zuweisungsoperator = ... 369
8.2.3 Mathematische Vektoren ... 371
8.2.4 Multiplikationsoperator ... 372
8.3 Inkrement-Operator ++ ... 374
8.4 Typumwandlungsoperator ... 378
8.5 Smart Pointer: Operatoren -> und * ... 380
8.5.1 Smart Pointer und die C++-Standardbibliothek ... 385
8.6 Objekt als Funktion ... 386
8.7 Spaceship-Operator <=> ... 388
8.7.1 Ordnungen in C++ ... 389
8.7.2 Automatische Erzeugung der Vergleichsoperatoren ... 390
8.7.3 Klassenspezifische Sortierung ... 391
8.7.4 Freie Funktionen statt Elementfunktionen ... 392
8.8 new und delete überladen ... 393
8.8.1 Unterscheidung zwischen Heap- und Stack-Objekten ... 397
8.8.2 Empfehlungen im Umgang mit new und delete ... 399
8.9 Operatoren für Literale ... 399

8.9.1 Stringliterale ... 400
8.9.2 Benutzerdefinierte Literale ... 401
8.10 Operatoren [] und () für Matrizen ... 403
8.10.1 Zweidimensionale Matrix als Vektor von Vektoren ... 404
8.10.2 Zweidimensionale Matrix mit zusammenhängendem Speicher ... 406
8.11 Zuweisung und Vergleich bei Vererbung ... 409
8.11.1 Polymorpher Vergleich ... 410
8.11.2 Kopie mit clone()-Methode erzeugen ... 410
8.11.3 Kopie mit Zuweisungsoperator erzeugen ... 410

9 Dateien und Ströme ... 413
9.1 Eingabe ... 415
9.2 Ausgabe ... 417
9.3 Formatierung mit std::format ... 419
9.3.1 Syntax für Platzhalter ... 419
9.3.2 Formatierung eigener Datentypen ... 423
9.4 Formatierung mit Flags ... 424
9.5 Formatierung mit Manipulatoren ... 427
9.5.1 Eigene Manipulatoren ... 433
9.6 Fehlerbehandlung ... 434
9.7 Typumwandlung von Dateiobjekten nach bool ... 436
9.8 Arbeit mit Dateien ... 437
9.8.1 Positionierung in Dateien ... 438
9.8.2 Lesen und Schreiben in derselben Datei ... 438
9.9 Umleitung auf Strings ... 439
9.10 Formatierte Daten lesen ... 440
9.10.1 Eingabe benutzerdefinierter Typen ... 440
9.11 Blockweise lesen und schreiben ... 442
9.11.1 vector-Objekt binär lesen und schreiben ... 442
9.11.2 array-Objekt binär lesen und schreiben ... 444
9.11.3 Matrix binär lesen und schreiben ... 445

10 Die Standard Template Library (STL) ... 447
10.1 Container, Iteratoren, Algorithmen ... 448
10.2 Iteratoren im Detail ... 453
10.3 Beispiel verkettete Liste ... 455
10.4 Bereiche (ranges) ... 459

Teil II: Fortgeschrittene Themen 463

11 Performance, Wert- und Referenzsemantik 465
11.1 Performanceproblem Wertsemantik 467
11.1.1 Auslassen der Kopie 467
11.1.2 Temporäre Objekte bei der Zuweisung 468
11.2 Referenzsemantik für R-Werte 469
11.2.1 Kategorien von Ausdrücken 469
11.2.2 Referenzen auf R- und L-Werte 470
11.2.3 Auswertung von Referenzen auf R-Werte 471
11.3 Optimierung durch Referenzsemantik für R-Werte 472
11.3.1 Bewegender Konstruktor 475
11.3.2 Bewegender Zuweisungsoperator 475
11.4 Die move()-Funktion 476
11.5 Referenzen auf R-Werte und Template-Parameter 478
11.5.1 Auswertung von Template-Parametern – ein Überblick 479
11.6 Ein effizienter binärer Plusoperator 480
11.6.1 Eliminieren auch des bewegenden Konstruktors 481
11.6.2 Kopien temporärer Objekte eliminieren 482
11.7 Rule of three/five/zero 483
11.7.1 Rule of three 483
11.7.2 Rule of five 483
11.7.3 Rule of zero 484

12 Lambda-Funktionen 485
12.1 Eigenschaften 486
12.1.1 Äquivalenz zum Funktionszeiger 487
12.1.2 Lambda-Funktion und Klasse 488
12.2 Generische Lambda-Funktionen 489
12.2.1 Generische Lambda-Funktionen mit Templates 490
12.3 Parametererfassung mit [] 491

13 Metaprogrammierung mit Templates 493
13.1 Grundlagen 494
13.2 Variadic Templates: Templates mit variabler Parameterzahl 497
13.2.1 Ablauf der Auswertung durch den Compiler 497
13.2.2 Anzahl der Parameter 499
13.2.3 Parameterexpansion 499

13.3 Fold-Expressions ... 501
13.3.1 Weitere Varianten ... 502
13.3.2 Fold-Expression mit Kommaoperator ... 503
13.4 Klassen-Template mit variabler Stelligkeit ... 505
13.5 Type Traits ... 505
13.5.1 Wie funktionieren Type Traits? – ein Beispiel ... 506
13.5.2 Abfrage von Eigenschaften ... 509
13.5.3 Abfrage numerischer Eigenschaften ... 511
13.5.4 Typumwandlungen ... 511
13.5.5 Auswahl weiterer Traits ... 512
13.6 Concepts ... 514

14 Reguläre Ausdrücke ... 519
14.1 Elemente regulärer Ausdrücke ... 520
14.1.1 Greedy oder lazy? ... 522
14.2 Interaktive Auswertung ... 524
14.3 Auszug der regex-Schnittstelle ... 527
14.4 Verarbeitung von \n ... 528
14.5 Anwendungen ... 530

15 Threads und Coroutinen ... 531
15.1 Zeit und Dauer ... 532
15.2 Threads ... 533
15.2.1 Automatisch join() ... 537
15.3 Die Klasse jthread ... 538
15.3.1 Übergabe eines Funktors ... 540
15.3.2 Thread-Group ... 542
15.4 Synchronisation kritischer Abschnitte ... 543
15.4.1 Data Race erkennen ... 546
15.5 Thread-Steuerung: Pausieren, Fortsetzen, Beenden ... 546
15.6 Warten auf Ereignisse ... 550
15.7 Atomare Veränderung von Variablen ... 556
15.8 Asynchrone verteilte Bearbeitung einer Aufgabe ... 559
15.8.1 future und async ... 559
15.8.2 packaged_task und future ... 561
15.8.3 promise und future ... 562
15.9 Thread-Sicherheit ... 563
15.10 Coroutinen ... 564

16 Grafische Benutzungsschnittstellen 569
16.1 Ereignisgesteuerte Programmierung 570
16.2 GUI-Programmierung mit Qt 571
16.2.1 Installation und Einsatz 571
16.2.2 Meta-Objektsystem 572
16.2.3 Der Programmablauf 573
16.2.4 Ereignis abfragen 574
16.3 Signale, Slots und Widgets 575
16.4 Dialog 584
16.5 Qt oder Standard-C++? 587
16.5.1 Threads 587
16.5.2 Verzeichnisbaum durchwandern 589

17 Internet-Anbindung 591
17.1 Protokolle 592
17.2 Adressen 592
17.3 Socket 595
17.3.1 Bidirektionale Kommunikation 598
17.3.2 UDP-Sockets 600
17.3.3 Atomuhr mit UDP abfragen 601
17.4 HTTP 604
17.4.1 Verbindung mit GET 605
17.4.2 Verbindung mit POST 610
17.5 Mini-Webserver 611

18 Datenbankanbindung 621
18.1 C++-Interface 622
18.2 Anwendungsbeispiel 625

Teil III: Ausgewählte Methoden und Werkzeuge der Softwareentwicklung 631

19 Effiziente Programmerzeugung mit make 633
19.1 Wirkungsweise 634
19.2 Variablen und Muster 636
19.3 Universelles Makefile für einfache Projekte 638
19.4 Automatische Ermittlung von Abhängigkeiten 639
19.4.1 Makefiles für verschiedene Betriebssysteme und Compiler 641

19.4.2 Getrennte Verzeichnisse: src, obj, bin 642
19.5 Makefile für Verzeichnisbäume 643
19.5.1 Nur ein Makefile auf Projektebene 645
19.5.2 Rekursive Make-Aufrufe 646
19.6 Erzeugen von Bibliotheken 648
19.6.1 Statische Bibliotheksmodule 648
19.6.2 Dynamische Bibliotheksmodule 650
19.7 Weitere Build-Tools 653

20 Unit-Test 655
20.1 Werkzeuge 656
20.2 Test Driven Development 657
20.3 Boost Unit Test Framework 658
20.3.1 Beispiel: Testgetriebene Entwicklung einer Operatorfunktion 660
20.3.2 Fixture 665
20.3.3 Testprotokoll und Log-Level 666
20.3.4 Prüf-Makros 667
20.3.5 Kommandozeilen-Optionen 671

Teil IV: Das C++-Rezeptbuch: Tipps und Lösungen für typische Aufgaben 673

21 Sichere Programmentwicklung 675
21.1 Regeln zum Design von Methoden 675
21.2 Defensive Programmierung 677
21.2.1 double- und float-Werte richtig vergleichen 678
21.2.2 const und constexpr verwenden 679
21.2.3 Anweisungen nach for/if/while einklammern 679
21.2.4 int und unsigned/size_t nicht mischen 679
21.2.5 size_t oder auto statt unsigned int verwenden 679
21.2.6 Postfix++ mit Präfix++ implementieren 680
21.2.7 Ein Destruktor darf keine Exception werfen 681
21.2.8 explicit-Typumwandlungsoperator bevorzugen 681
21.2.9 explicit-Konstruktor für eine Typumwandlung bevorzugen 681
21.2.10 Leere Standardkonstruktoren vermeiden 681
21.2.11 Mit override Schreibfehler reduzieren 681
21.2.12 Kopieren und Zuweisung verbieten 682

21.2.13 Vererbung verbieten 682
21.2.14 Überschreiben einer virtuellen Methode verhindern 682
21.2.15 »Rule of zero« beachten 683
21.2.16 One Definition Rule 683
21.2.17 Defensiv Objekte löschen 683
21.2.18 Speicherbeschaffung und -freigabe kapseln 684
21.2.19 Programmierrichtlinien einhalten 684
21.3 Exception-sichere Beschaffung von Ressourcen 684
21.3.1 Sichere Verwendung von unique_ptr und shared_ptr 684
21.3.2 So vermeiden Sie new und delete! 685
21.3.3 shared_ptr für Arrays korrekt verwenden 685
21.3.4 unique_ptr für Arrays korrekt verwenden 687
21.3.5 Exception-sichere Funktion 687
21.3.6 Exception-sicherer Konstruktor 688
21.3.7 Exception-sichere Zuweisung 689
21.4 Empfehlungen zur Thread-Programmierung 689
21.4.1 Warten auf die Freigabe von Ressourcen 689
21.4.2 Deadlock-Vermeidung 690
21.4.3 notify_all oder notify_one? 691
21.4.4 Performance mit Threads verbessern? 691

22 Von der UML nach C++ 693
22.1 Vererbung 694
22.2 Interface anbieten und nutzen 694
22.3 Assoziation 696
22.3.1 Aggregation 699
22.3.2 Komposition 700

23 Algorithmen für verschiedene Aufgaben 701
23.1 Algorithmen mit Strings 702
23.1.1 String splitten 702
23.1.2 String in Zahl umwandeln 703
23.1.3 Zahl in String umwandeln 706
23.1.4 Strings sprachlich richtig sortieren 706
23.1.5 Umwandlung in Klein- bzw. Großschreibung 708
23.1.6 Strings sprachlich richtig vergleichen 709
23.1.7 Von der Groß-/Kleinschreibung unabhängiger Zeichenvergleich 710
23.1.8 Von der Groß-/Kleinschreibung unabhängige Suche 710

23.2 Textverarbeitung ... 712
23.2.1 Datei durchsuchen ... 712
23.2.2 Ersetzungen in einer Datei ... 713
23.2.3 Lines of Code (LOC) ermitteln ... 715
23.2.4 Zeilen, Wörter und Zeichen einer Datei zählen ... 717
23.2.5 CSV-Datei lesen ... 717
23.2.6 Kreuzreferenzliste ... 718
23.3 Operationen auf Folgen ... 720
23.3.1 Vereinfachungen ... 720
23.3.2 Folge mit gleichen Werten initialisieren ... 722
23.3.3 Folge mit Werten eines Generators initialisieren ... 723
23.3.4 Folge mit fortlaufenden Werten initialisieren ... 723
23.3.5 Summe und Produkt ... 724
23.3.6 Mittelwert und Standardabweichung ... 725
23.3.7 Skalarprodukt ... 726
23.3.8 Folge der Teilsummen oder -produkte ... 727
23.3.9 Folge der Differenzen ... 728
23.3.10 Kleinstes und größtes Element finden ... 730
23.3.11 Elemente rotieren ... 731
23.3.12 Elemente verwürfeln ... 732
23.3.13 Dubletten entfernen ... 733
23.3.14 Reihenfolge umdrehen ... 735
23.3.15 Stichprobe ... 736
23.3.16 Anzahl der Elemente, die einer Bedingung genügen ... 737
23.3.17 Gilt ein Prädikat für alle, kein oder wenigstens ein Element einer Folge? ... 738
23.3.18 Permutationen ... 739
23.3.19 Lexikografischer Vergleich ... 742
23.4 Sortieren und Verwandtes ... 744
23.4.1 Partitionieren ... 744
23.4.2 Sortieren ... 747
23.4.3 Stabiles Sortieren ... 747
23.4.4 Partielles Sortieren ... 749
23.4.5 Das n.-größte oder n.-kleinste Element finden ... 750
23.4.6 Verschmelzen (merge) ... 752
23.5 Suchen und Finden ... 754
23.5.1 Element finden ... 754
23.5.2 Element einer Menge in der Folge finden ... 755

23.5.3 Teilfolge finden ... 756
23.5.4 Teilfolge mit speziellem Algorithmus finden ... 758
23.5.5 Bestimmte benachbarte Elemente finden ... 759
23.5.6 Bestimmte aufeinanderfolgende Werte finden ... 760
23.5.7 Binäre Suche ... 761
23.6 Mengenoperationen auf sortierten Strukturen ... 764
23.6.1 Teilmengenrelation ... 764
23.6.2 Vereinigung ... 765
23.6.3 Schnittmenge ... 766
23.6.4 Differenz ... 767
23.6.5 Symmetrische Differenz ... 768
23.7 Heap-Algorithmen ... 768
23.7.1 pop_heap ... 770
23.7.2 push_heap ... 771
23.7.3 make_heap ... 771
23.7.4 sort_heap ... 772
23.7.5 is_heap ... 772
23.8 Vergleich von Containern auch ungleichen Typs ... 773
23.8.1 Unterschiedliche Elemente finden ... 773
23.8.2 Prüfung auf gleiche Inhalte ... 775
23.9 Rechnen mit komplexen Zahlen: Der C++-Standardtyp complex ... 776
23.10 Vermischtes ... 778
23.10.1 Erkennung eines Datums ... 778
23.10.2 Erkennung einer IPv4-Adresse ... 780
23.10.3 Erzeugen von Zufallszahlen ... 781
23.10.4 for_each – auf jedem Element eine Funktion ausführen ... 786
23.10.5 Verschiedene Möglichkeiten, Container-Bereiche zu kopieren ... 787
23.10.6 Vertauschen von Elementen, Bereichen und Containern ... 790
23.10.7 Elemente transformieren ... 791
23.10.8 Ersetzen und Varianten ... 792
23.10.9 Elemente herausfiltern ... 794
23.10.10 Grenzwerte von Zahltypen ... 795
23.10.11 Minimum und Maximum ... 796
23.10.12 Wert begrenzen ... 798
23.10.13 ggT, kgV und Mitte ... 799
23.11 Parallelisierbare Algorithmen ... 800

24 Datei- und Verzeichnisoperationen 803
24.1 Übersicht 804
24.2 Pfadoperationen 805
24.3 Datei oder Verzeichnis löschen 806
24.4 Datei oder Verzeichnis kopieren 808
24.5 Verzeichnis anlegen 809
24.6 Datei oder Verzeichnis umbenennen 810
24.7 Verzeichnis anzeigen 811
24.8 Verzeichnisbaum anzeigen 812

Teil V: Die C++-Standardbibliothek 813

25 Aufbau und Übersicht 815
25.1 Auslassungen 817
25.2 Beispiele des Buchs und die C++-Standardbibliothek 819

26 Hilfsfunktionen und -klassen 821
26.1 Unterstützung der Referenzsemantik für R-Werte 821
26.2 Paare 823
26.3 Tupel 825
26.4 bitset 827
26.5 Indexfolgen 830
26.6 variant statt union 831
26.7 Funktionsobjekte 832
26.7.1 Arithmetische, vergleichende und logische Operationen 832
26.7.2 Binden von Argumentwerten 833
26.7.3 Funktionen in Objekte umwandeln 835
26.8 Templates für rationale Zahlen 837
26.9 Hüllklasse für Referenzen 838
26.10 Optionale Objekte 839

27 Container 841
27.1 Gemeinsame Eigenschaften 843
27.1.1 Reversible Container 845
27.1.2 Initialisierungsliste (initializer_list) 846
27.1.3 Konstruktion an Ort und Stelle 847
27.2 Sequenzen 848
27.2.1 vector 849

27.2.2 vector<bool> 850
27.2.3 array 851
27.2.4 list 854
27.2.5 deque 856
27.3 Container-Adapter 858
27.3.1 stack 858
27.3.2 queue 859
27.3.3 priority_queue 860
27.4 Assoziative Container 862
27.4.1 Sortierte assoziative Container 864
27.4.2 Hash-Container 872
27.5 Sicht auf Container (span) 879

28 Iteratoren 881
28.1 Iterator-Kategorien 882
28.1.1 Anwendung von Traits 884
28.2 Abstand und Bewegen 887
28.3 Zugriff auf Anfang und Ende 888
28.3.1 Reverse-Iteratoren 889
28.4 Insert-Iteratoren 890
28.5 Stream-Iteratoren 892

29 Algorithmen 895
29.1 Algorithmen mit Prädikat 896
29.2 Übersicht 897

30 Nationale Besonderheiten 901
30.1 Sprachumgebung festlegen und ändern 902
30.1.1 Die locale-Funktionen 903
30.2 Zeichensätze und -codierung 904
30.3 Zeichenklassifizierung und -umwandlung 909
30.4 Kategorien 910
30.4.1 collate 910
30.4.2 ctype 911
30.4.3 numeric 912
30.4.4 monetary 914
30.4.5 time 917
30.4.6 messages 919

30.5 Konstruktion eigener Facetten 921

31 String 923
31.1 string_view für String-Literale 933

32 Speichermanagement 935
32.1 unique_ptr 935
32.2 shared_ptr 938
32.3 weak_ptr 940
32.4 new mit Speicherortangabe 941

33 Ausgewählte C-Header 943
33.1 <cassert> 943
33.2 <cctype> 944
33.3 <cmath> 945
33.4 <cstddef> 946
33.5 <cstdlib> 946
33.6 <cstring> 947
33.7 <ctime> 949

A Anhang 951
A.1 ASCII-Tabelle 951
A.2 C++-Schlüsselwörter 953
A.3 Compilerbefehle 954
A.4 Rangfolge der Operatoren 955
A.5 C++-Attribute für den Compiler 957
A.6 Lösungen zu den Übungsaufgaben 957
A.7 Änderungen in der 6. Auflage 967

Glossar 969

Literaturverzeichnis 979

Register 983

Vorwort

Diese Auflage unterscheidet sich von der vorherigen durch eine gründliche Überarbeitung und die Umstellung auf den 2020 von der zuständigen ISO/IEC-Arbeitsgruppe verabschiedeten C++-Standard.[1] Das Buch ist konform zum C++20-Standard, ohne den Anspruch auf Vollständigkeit zu erheben – das Standard-Dokument [ISOC++] umfasst allein 1841 Seiten. Sie finden in diesem Buch eine verständliche und mit vielen Beispielen angereicherte Einführung in die Sprache, unabhängig vom Betriebssystem.

Für wen ist dieses Buch geschrieben?

Es ist für alle geschrieben, die einen kompakten und gleichzeitig in die Tiefe gehenden Einstieg in die Programmierung mit C++ suchen. Es ist für Interessierte ohne Programmiererfahrung gedacht und für andere, die diese Programmiersprache kennenlernen möchten. Beiden Gruppen dient das Buch als Lehrbuch und Nachschlagewerk.

Ein umfassendes Handbuch

Die ersten zehn Kapitel führen in die Sprache ein, die folgenden behandeln fortgeschrittene Themen. Die sofortige praktische Umsetzung des Gelernten anhand von leicht nachvollziehbaren Beispielen steht im Vordergrund. Klassen und Objekte, Templates und Exceptions sind Ihnen bald keine Fremdworte mehr. Es gibt 101 Übungsaufgaben – mit Musterlösungen im Anhang und zum Download. Durch das Studium dieser Kapitel werden aus Neulingen bald Fortgeschrittene – und mithilfe der weiteren Kapitel Experten.

C++ in praktischen Anwendungen

Sie finden kurze Einführungen in die Themen Programmierung paralleler Abläufe, Netzwerk-Programmierung einschließlich eines kleinen Webservers, Datenbankanbindung und grafische Benutzungsoberflächen. Dabei wird durch Einsatz der Boost-Library und des Qt-Frameworks größtmögliche Portabilität erreicht.

---

[1] Wer C++-Vorkenntnisse hat, kann einen Eindruck von den Änderungen mit einem Blick in den Anhang A.7 gewinnen.

Softwareentwicklung ist nicht nur Programmierung

Sie lernen die Automatisierung der Programmerzeugung mit Make kennen. Das Programmdesign wird durch konkrete Umsetzungen von Design-Mustern nach C++ unterstützt. Das Kapitel über Unit-Tests zeigt, wie Programme getestet werden können. Das integrierte »C++-Rezeptbuch« mit mehr als 150 praktischen Lösungen, der Teil über die C++-Standardbibliothek, das umfangreiche Register und das detaillierte Inhaltsverzeichnis machen das Buch zu einem praktischen Nachschlagewerk für alle, die sich mit der Softwareentwicklung in C++ beschäftigen.

Moderne Programmiermethodik

Sie möchten Programme schreiben, die hohen Qualitätsansprüchen gerecht werden. Dazu gehört das Know-how, C++ richtig einzusetzen. Dass ein Programm läuft, reicht nicht. Es soll auch gut entworfen sein, möglichst wenige Fehler enthalten, selbst mit Fehlern in Daten umgehen können, verständlich geschrieben und schnell in der Ausführung sein. Deshalb liegt ein Schwerpunkt des Buchs auf guter Codierpraxis entsprechend den »C++ Core Guidelines«. Die Umsetzung wird an vielen Beispielen gezeigt.

Wie benutzen Sie dieses Buch am besten?

Es eignet sich zum Selbststudium oder als Begleitbuch zu einem Kurs oder einer Vorlesung. Man lernt am besten durch eigenes Tun! Dabei hilft es, die Beispiele herunterzuladen, sie zu studieren und zu modifizieren (*http://www.cppbuch.de/*). Auch wird empfohlen, die Übungsaufgaben zu lösen. Um sowohl Anfängern als auch Fortgeschrittenen gerecht zu werden, gibt es einfache, aber auch schwerere Aufgaben. Wenn Ihnen eine Lösung nicht gelingt – einfach bei den Lösungen im Anhang nachsehen bzw. im Verzeichnis *cppbuch/loesungen* der downloadbaren Beispiele. Und dann versuchen, die Lösungen nachzuvollziehen.

Wo finden Sie was?

Bei der Programmentwicklung wird häufig das Problem auftauchen, etwas nachschlagen zu müssen. Es gibt die folgenden Hilfen: Erklärungen zu Begriffen sind im *Glossar* aufgeführt. Es gibt ein umfangreiches *Stichwortverzeichnis* und ein detailliertes *Inhaltsverzeichnis*. Der Anhang enthält unter anderem verschiedene hilfreiche Tabellen und die Lösungen der Übungsaufgaben. Auf der Webseite *http://www.cppbuch.de/* finden Sie die Software zu diesem Buch. Sie enthält alle Programmbeispiele und die Lösungen zu den Aufgaben. Sie finden dort auch weitere Hinweise, Errata und nützliche Internet-Links.

Zu guter Letzt

Allen Menschen, die dieses Buch durch Hinweise und Anregungen verbessern halfen, sei an dieser Stelle herzlich gedankt. Insbesondere danke ich Herrn Dr. Daniel Krügler für viele hilfreiche Kommentare und Frau Irene Weilhart vom Hanser Verlag für die mittlerweile jahrzehntelange sehr gute Zusammenarbeit.

Bremen, im August 2020

Ulrich Breymann

1

Es geht los!

Dieses Kapitel behandelt die folgenden Themen:

- Entstehung und Entwicklung der Programmiersprache C++
- Objektorientierte Programmierung – erste Grundlagen
- Wie schreibe ich ein Programm und bringe es zum Laufen?
- Einfache Datentypen und Operationen
- Ablauf innerhalb eines Programms steuern
- Erste Definition eigener Datentypen
- Standarddatentypen vector und string
- Einfache Ein- und Ausgabe
- Guter Programmierstil

1.1 Historisches

C++ wurde etwa ab 1980 von Bjarne Stroustrup als die Programmiersprache »C with classes« (englisch *C mit Klassen*), die Objektorientierung stark unterstützt, auf der Basis der Programmiersprache C entwickelt. Später wurde die neue Sprache in C++ umbenannt. ++ ist ein Operator der Programmiersprache C, der den Wert einer Größe um 1 erhöht. Insofern spiegelt der Name die Eigenschaft »Nachfolger von C«. 1998 wurde C++ erstmals von der ISO (International Organization for Standardization) und der IEC (International Electrotechnical Commission) standardisiert. Diesem Standard haben sich nationale Standardisierungsgremien wie ANSI (USA) und DIN (Deutschland) angeschlossen. Die Anforderungen an C++ sind gewachsen, auch zeigte sich, dass manches fehlte und anderes überflüssig oder fehlerhaft war. Das C++-Standardkomitee hat kontinuierlich an der Verbesserung von C++ gearbeitet. Seit 2011 werden im Abstand von drei

Jahren neue Versionen des Standards herausgegeben. Die Kurznamen sind entsprechend den Jahreszahlen C++11, C++14, C++17 und C++20. C++20 wurde von der zuständigen ISO/IEC-Arbeitsgruppe JTC1/SC22/WG21 verabschiedet und bei der ISO zur Veröffentlichung eingereicht.

1.2 Arten der Programmierung

Es gibt viele verschiedene Arten der Programmierung. C++ unterstützt im Wesentlichen die folgenden:

Imperative Programmierung

Dabei werden die im Programmcode festgelegten Schritte der Reihe nach ausgeführt. Es geht also darum, einem Rechner mitzuteilen, *was* er tun soll und *wie* es zu tun ist. Ein Programm ist ein in einer Programmiersprache formulierter Algorithmus oder anders ausgedrückt eine Folge von Anweisungen, die der Reihe nach auszuführen sind, ähnlich einem Kochrezept. Der Algorithmus wird in einer besonderen Sprache, die der Rechner »versteht«, geschrieben. Der Schwerpunkt dieser Betrachtungsweise liegt auf den einzelnen Schritten oder Anweisungen an den Rechner, die zur Lösung einer Aufgabe abzuarbeiten sind. Mit sogenannten Kontrollstrukturen wird der Programmablauf gesteuert. So können Teile wiederholt ausgeführt oder übersprungen werden.

Objektorientierte Programmierung

Bei der imperativen Programmierung wird ein Aspekt eher stiefmütterlich behandelt: Der Rechner muss »wissen«, *womit* er etwas tun soll. Zum Beispiel soll er

- eine bestimmte Summe Geld von einem Konto auf ein anderes transferieren;
- eine Ampelanlage steuern;
- ein Rechteck auf dem Bildschirm zeichnen.

Häufig, wie in den ersten beiden Fällen, werden Objekte der realen Welt (Konten, Ampelanlage ...) *simuliert*, das heißt im Rechner abgebildet. Die abgebildeten Objekte haben eine *Identität*. Das *Was* und das *Womit* gehören stets zusammen. Beide sind also Eigenschaften eines Objekts und sollen daher nicht getrennt werden. Ein Konto kann schließlich nicht auf Gelb geschaltet werden, und eine Überweisung an eine Ampel ist nicht vorstellbar. Ein *objektorientiertes Programm* kann man sich als Abbildung von Objekten der realen Welt in Software vorstellen. Die Abbildungen werden selbst wieder Objekte genannt. Klassen sind Beschreibungen von Objekten.

Generische Programmierung

Die generische Programmierung ermöglicht es, Klassen und Algorithmen für verschiedene Datentypen mit nur einem Schema zu modellieren. Das gilt für die imperative Programmierung ebenso wie für die objektorientierte Programmierung. Andere Arten der Programmierung sind für C++ von geringerer Bedeutung.

1.3 Werkzeuge zum Programmieren

Um Programme schreiben und ausführen zu können, brauchen Sie nicht viel: einen Computer mit einem Editor und einem C++-Compiler.

Der Editor

Ein Editor ist ein Programm, mit dem man Texte schreiben kann. Dabei darf ein Text keine versteckten Sonderzeichen enthalten, weswegen LibreOffice oder Word nicht geeignet sind. Ein für Windows passender Texteditor ist *Notepad++*[1]. Den Editor *Atom*[2] gibt es für Windows, Linux und macOS. Mac-Benutzer können auch den Editor der Entwicklungsumgebung *Xcode* nehmen.

Der Compiler

Der Compiler ist ein Programm, das Ihren Programmtext in eine für den Computer verarbeitbare Form übersetzen kann. Von Menschen geschriebener und für Menschen lesbarer Programmtext kann nicht vom Computer so verstanden werden, wie wir einen Text verstehen. Das vom Compiler erzeugte Ergebnis der Übersetzung kann der Computer aber ausführen. Das Erlernen einer Programmiersprache ohne eigenes praktisches Ausprobieren ist kaum sinnvoll. Nutzen Sie daher die Dienste des Compilers möglichst bald anhand der Beispiele. Wie, zeigt Ihnen der Abschnitt direkt nach der Vorstellung des ersten Programms.

GNU C++ Compiler: g++

Die am meisten verbreiteten Compiler sind der GNU C++-Compiler [GCC] *g++*, der zu Microsofts Visual Studio gehörende *cl* und seit einiger Zeit auch *clang++* von LLVM[3]. Um C++ zu lernen, ist es letztlich egal, welchen Compiler Sie nehmen. In diesem Buch liegt der Schwerpunkt auf dem *g++*-Compiler, weil er Open-Source ist und auf den wichtigsten Betriebssystemen läuft. *g++* wie auch *clang++* werden am schnellsten an die Entwicklungen des C++-Standards angepasst.

Windows

Installationshinweise finden Sie auf der Internetseite *http://cppbuch.de/downloads.html*. Im Folgenden wird davon ausgegangen, dass Sie die Installation wie dort beschrieben vorgenommen haben. Auf derselben Internetseite liegen auch die Beispiele zum Download bereit.

Linux

Bei den meisten Linux-Systemen ist der GNU C++-Compiler enthalten. Wenn nicht, finden Sie Hinweise zur Installation unter *http://www.cppbuch.de/swinstallation.html*.

---

[1] *https://notepad-plus-plus.org/*
[2] *https://atom.io*
[3] *http://llvm.org*

LLVM C++ Compiler: clang++

clang++ hat mit wenigen Ausnahmen dieselben Optionen wie *g++*. In diesem Buch wird daher nicht weiter auf *clang++* eingegangen. Auf einem Mac sollten Sie die Entwicklungsumgebung Xcode mit dem Clang++-Compiler installieren. Hinweise dazu finden Sie unter *http://www.cppbuch.de/swinstallation.html*. Der Aufruf von *g++* wird vom System durch einen Aufruf von *clang++* ersetzt.

Microsofts Visual C++ Compiler: cl

Wenn Sie die Entwicklungsumgebung Visual Studio herunterladen (Tipp: Community Edition), bekommen Sie auch den zugehörigen Compiler. Um weitgehend unabhängig vom Betriebssystem zu bleiben, spielt er hier nur eine geringfügige Rolle.

Wenn Sie etwas Erfahrung mit C++ gewonnen haben, bietet sich eine integrierte Entwicklungumgebung (englisch *Integrated Development Environment, IDE*) an, um die Beispiele korrekt zu übersetzen. Abschnitt 1.5 geht darauf ein.

Tipp

Um für den Anfang die Einarbeitung in eine Integrierte Entwicklungsumgebung zu vermeiden, nehmen Sie einen Texteditor. Das Programm kann dann in der Konsole oder im Linux- bzw. macOS-Terminal mit den weiter unten gezeigten Kommandos compiliert werden. Compilieren heißt, den Programmtext in eine ausführbare Form umzuwandeln, die die Maschine »versteht«.

1.4 Das erste Programm

Das klassische erste Programm ist ein Mini-Programm, das einfach nur »Hello World!« ausgibt. Das Listing 1.1. zeigt den Programmcode.

Listing 1.1: Hello World-Programm *(cppbuch/k1/hello.cpp)*

```
#include <iostream>

int main()
{
  std::cout << "Hello␣World!\n";
}
```

Die Entwicklung eines einfachen Programms lernen Sie hier an einer nur unwesentlich schwierigeren Aufgabe kennen: Es sollen zwei Zahlen addiert werden. Dabei wird Ihnen zunächst das Programm vorgestellt und gleich danach erfahren Sie, wie Sie es eingeben und zum Laufen bringen können. Der erste Schritt besteht in der Formulierung der Aufgabe. Sie lautet: »Lies zwei Zahlen a und b von der Tastatur ein. Berechne die Summe beider Zahlen und zeige das Ergebnis auf dem Bildschirm an.« Die Aufgabe ist so einfach,

wie sie sich anhört! Im zweiten Schritt wird die Aufgabe in die Teilaufgaben »Eingabe«, »Berechnung« und »Ausgabe« zerlegt:

Listing 1.2: Programmentwurf

```
int main()                  // Noch tut dieses Programm nichts!
{
    // Lies zwei Zahlen ein
    /* Berechne die Summe beider
       Zahlen  */
    // Zeige das Ergebnis auf dem Bildschirm an
}
```

Sie sehen einen einfachen Entwurf, der gleichzeitig ein C++-Programm ist. Es tut allerdings noch nichts. Es bedeuten:

`int`	ganze Zahl zur Rückgabe
`main`	Name der Funktion, mit der jedes Programm beginnt.
`()`	Innerhalb dieser Klammern können der Funktion Informationen mitgegeben werden.
`{ }`	Block
`/* ... */`	Kommentar, der über mehrere Zeilen gehen kann
`// ...`	Kommentar bis Zeilenende

Ein durch `{` und `}` begrenzter *Block* enthält die Anweisungen an den Rechner. Der *Compiler* übersetzt den Programmtext in eine rechnerverständliche Form. Im Programm sind lediglich *Kommentare* enthalten und noch keine Anweisungen an den Computer, sodass unser Programm deswegen nichts tut.

Kommentare werden einschließlich der Kennungen vom Compiler vollständig ignoriert. Ein Kommentar, der mit `/*` beginnt, ist mit der ersten `*/`-Zeichenkombination beendet, auch wenn er sich über mehrere Zeilen erstreckt. Ein mit `//` beginnender Kommentar endet am Ende der Zeile. Auch wenn Kommentare vom Compiler ignoriert werden, sind sie doch sinnvoll für alle, die ein Programm lesen. Die Anweisungen zu erläutern hilft denjenigen, die Ihre Nachfolge antreten, weil Sie befördert worden sind oder die Firma verlassen haben. Kommentare sind auch wichtig für den Autor eines Programms, der nach einem halben Jahr nicht mehr weiß, warum er gerade diese oder jene komplizierte Anweisung geschrieben hat. Sie sehen:

Ein Programm ist »nur« ein Text!

- Der Text hat eine Struktur entsprechend den C++-Sprachregeln: Es gibt Wörter wie hier das Schlüsselwort `main` (in C++ werden alle Schlüsselwörter kleingeschrieben). Es gibt weiterhin Zeilen, Satzzeichen und Kommentare.
- Die Bedeutung des Textes wird durch die Zeilenstruktur nicht beeinflusst. Mit `\` und folgendem (ENTER) ist eine Worttrennung am Zeilenende möglich. Das Zeichen `\` wird »Backslash« genannt. Mit dem Symbol (ENTER) ist hier und im Folgenden die Betätigung der großen Taste (↵) rechts auf der Tastatur gemeint.
- Groß- und Kleinschreibung werden unterschieden! `main()` ist nicht dasselbe wie `Main()`.

Weil die Zeilenstruktur für den Rechner keine Rolle spielt, kann der Programmtext nach Gesichtspunkten der Lesbarkeit gestaltet werden. Im dritten Schritt müssen »nur« noch die Inhalte der Kommentare als C++-Anweisungen formuliert werden. Dabei bleiben die Kommentare zur Dokumentation stehen, wie im Beispielprogramm unten zu sehen ist.

Hinweis

Alle Programmbeispiele sind von der Internet-Seite *http://cppbuch.de/* herunterladbar. In den Listings finden Sie den zugehörigen Dateinamen in der Überschrift oder in der ersten Zeile des Listings.

Listing 1.3: Summe zweier Zahlen berechnen *(cppbuch/k1/summe.cpp)*

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

int main()
{
  int summand1;
  int summand2;

  // Lies zwei Zahlen ein
  cout << "␣Zwei␣ganze␣Zahlen␣eingeben:";
  cin >> summand1 >> summand2;
  /* Berechne die Summe beider Zahlen
  */
  int summe = summand1 + summand2;

  // Zeige das Ergebnis auf dem Bildschirm an
  cout << "Summe=" << summe << '\n';
  return 0;
}
```

Es sind einige neue Worte dazugekommen, die hier kurz erklärt werden. Machen Sie sich keine Sorgen, wenn Sie nicht alles auf Anhieb verstehen! Alles wird im Verlauf des Buchs wieder aufgegriffen und vertieft. Wie das Programm zum Laufen gebracht wird, werden Sie bald erfahren.

`#include<iostream>` Einbindung der Ein-/Ausgabefunktionen. Diese Zeile muss in jedem Programm stehen, das Eingaben von der Tastatur erwartet oder Ausgaben auf den Bildschirm bringt. Sie können sich vorstellen, dass der Compiler beim Übersetzen des Programms an dieser Stelle erst alle zur Ein- und Ausgabe notwendigen Informationen liest. Details folgen in Abschnitt 2.3.

`using namespace std;` Der Namensraum `std` wird benutzt. Schreiben Sie es einfach in jedes Programm an diese Stelle und haben Sie Geduld: Erklärungen folgen auf den Seiten 63 und 147.

`int main()` `main()` ist die Funktion, mit der jedes Programm beginnt (es gibt auch andere Funktionen). Der zu `main()` gehörende Programmcode wird durch die geschweiften Klammern `{` und `}` eingeschlossen.

	Ein mit { und } begrenzter Bereich heißt *Block*. Mit `int` ist gemeint, dass die `main()`-Funktion nach Beendigung eine Zahl vom Typ `int` (= ganze Zahl) an das Betriebssystem zurückgibt. Dazu dient die unten beschriebene `return`-Anweisung. Normalerweise – das heißt bei ordnungsgemäßem Programmablauf – wird die Zahl 0 zurückgegeben. Andere Zahlen können über das Betriebssystem einem folgenden Programm einen Fehler signalisieren.

`int summand1;`
`int summand2;`
`int summe = ...`
	Deklaration von Objekten: Mitteilung an den Compiler, der entsprechend Speicherplatz bereitstellt und ab jetzt die Namen `summand1`, `summand2` und `summe` innerhalb des Blocks `{ }` kennt. Es gibt verschiedene Zahlentypen in C++. Mit `int` sind ganze Zahlen gemeint: `summe`, `summand1`, `summand2` sind ganze Zahlen. Oft ist es sinnvoll, einen Anfangswert festzulegen, etwa 0. Dazu später mehr.

`;`	Ein Semikolon beendet jede Deklaration und jede Anweisung (aber keine Verbundanweisung, siehe weiter unten).
`cout`	Ausgabe: `cout` (Abkürzung für *character out* oder *console out*) ist die Standardausgabe. Der Doppelpfeil deutet an, dass alles, was rechts davon steht, zur Ausgabe `cout` gesendet wird, zum Beispiel `cout << summand1;`. Wenn mehrere Dinge ausgegeben werden sollen, sind sie durch << zu trennen.
`cin`	Eingabe: Der Doppelpfeil zeigt hier in Richtung des Objekts, das ja von der Tastatur einen neuen Wert aufnehmen soll. Die Information fließt von der Eingabe `cin` zum Objekt `summand1` beziehungsweise zum Objekt `summand2`.
`=`	Zuweisung: Der Variablen auf der linken Seite des Gleichheitszeichens wird das Ergebnis des Ausdrucks auf der rechten Seite zugewiesen.
`"Text"`	beliebige Zeichenkette, die die Anführungszeichen selbst nicht enthalten darf, weil sie als Anfangs- beziehungsweise Endemarkierung einer Zeichenfolge dienen. Wenn die Zeichenfolge die Anführungszeichen enthalten soll, sind diese als \" zu schreiben: `cout << "\"C++\" ist der Nachfolger von \"C\"!";` erzeugt die Bildschirmausgabe *"C++" ist der Nachfolger von "C"!*.
`'\n'`	die Ausgabe des Zeichens `\n` bewirkt eine neue Zeile.
`return 0;`	Unser Programm läuft einwandfrei, es gibt daher 0 an das Betriebssystem zurück. Diese Anweisung darf in der `main()`-Funktion fehlen, dann wird automatisch 0 zurückgegeben.

`<iostream>` ist ein Header. Dieser aus dem Englischen stammende Begriff (head = dt. Kopf) drückt aus, dass Zeilen dieser Art am Anfang eines Programmtextes stehen. Der Begriff wird im Folgenden verwendet, weil es zurzeit keine gängige deutsche Entsprechung gibt. Einen Header mit einem Dateinamen gleichzusetzen, ist meistens richtig, nach dem C++-Standard aber nicht zwingend.

summand1, summand2 und summe sind veränderliche Daten und heißen Variablen. Sie sind Objekte eines vordefinierten Grunddatentyps für ganze Zahlen (int), mit denen die üblichen Ganzzahloperationen wie +, - und = durchgeführt werden können. Der Begriff »Variable« wird für ein veränderliches Objekt gebraucht. Für Variablen gilt:

- Sie müssen deklariert werden. int summe; ist eine Deklaration, wobei int der *Datentyp* des Objekts summe ist, der die Eigenschaften beschreibt. Entsprechendes gilt für summand1 und summand2. Die Objektnamen sind frei wählbar im Rahmen der unten angegebenen Konventionen. Unter *Deklaration* wird verstanden, dass der Name dem Compiler bekannt gemacht wird. Wenn dieser Name danach im Programm versehentlich falsch geschrieben wird, kennt der Compiler den falschen Namen nicht und gibt eine Fehlermeldung aus. Somit dienen Deklarationen der Programmsicherheit.
- Objektnamen bezeichnen Bereiche im Speicher des Computers, deren Inhalte verändert werden können. Die Namen sind symbolische Adressen, unter denen der Wert gefunden wird. Über den Namen kann dann auf den aktuellen Wert zugegriffen werden (siehe Abbildung 1.1).

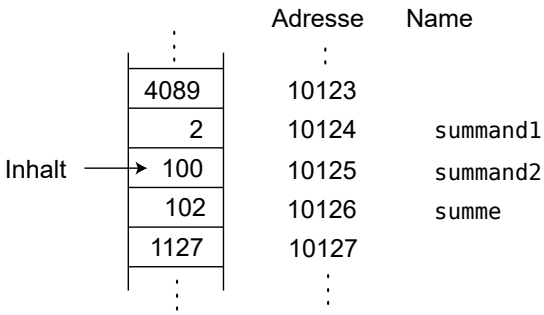

Abbildung 1.1: Speicherbereiche mit Adressen

Der Speicherplatz wird vom Compiler reserviert. Man spricht dann von der *Definition* der Objekte. Definition und Deklaration werden unterschieden, weil es auch Deklarationen ohne gleichzeitige Definition gibt, doch davon später mehr. Zunächst sind die Deklarationen zugleich Definitionen. Abbildung 1.2 zeigt den Ablauf der Erzeugung eines lauffähigen Programms. Ein Programm ist ein Text, von Menschenhand geschrieben (über Programmgeneratoren soll hier nicht gesprochen werden) und dem Rechner unverständlich. Um dieses Programm auszuführen, muss es erst vom Compiler in eine für den Computer verständliche Form übersetzt werden. Der Compiler ist selbst ein Programm, das bereits in maschinenverständlicher Form vorliegt und speziell für diese Übersetzung zuständig ist. Nach Eingabe des Programmtextes mit dem Editor können Sie den Compiler starten.

Ein Programmtext wird auch »Quelltext« oder »Quellcode« (englisch *source code*) genannt. Der Compiler erzeugt aus dem Quellcode den Objektcode, der noch nicht ausführbar ist. Hinter den einfachen Anweisungen cin >> ... und cout << ... verbergen sich eine Reihe von Aktivitäten wie die Abfrage der Tastatur und die Ansteuerung des Bildschirms, die nicht speziell programmiert werden müssen, weil sie schon in vorübersetzter Form in Bibliotheksdateien vorliegen. Die Aufrufe dieser Aktivitäten im Programm müssen mit den dafür vorgesehenen Algorithmen in den Bibliotheksdateien zusammengebun-

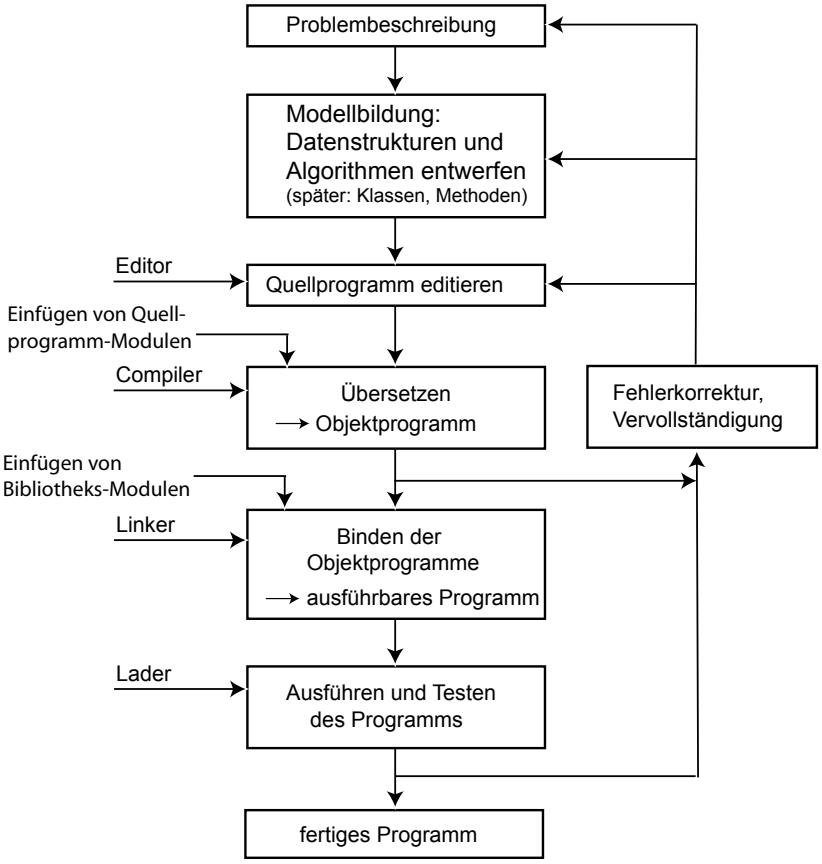

Abbildung 1.2: Erzeugung eines lauffähigen Programms

den werden, eine Aufgabe, die der *Linker* übernimmt, auch *Binder* genannt. Der Linker bindet Ihren Objektcode mit dem Objektcode der Bibliotheksdateien zusammen und erzeugt daraus ein ausführbares Programm, das nun gestartet werden kann. Der Aufruf des Programms bewirkt, dass der *Lader*, eine Funktion des Betriebssystems, das Programm in den Rechnerspeicher lädt und startet. Diese Schritte werden stets ausgeführt, auch wenn sie in den Programmentwicklungsumgebungen verborgen ablaufen. Bibliotheksmodule können auch während der Programmausführung geladen werden (nicht im Bild dargestellt).

Wie bekomme ich ein Programm zum Laufen?

Nachdem Sie den Programmtext mit einem Editor geschrieben haben, speichern Sie ihn als Datei *summe.cpp* ab. Öffnen Sie nun unter Windows eine MinGW-Konsole[4], bzw. ein Terminal unter Linux oder macOS, und wechseln mit `cd` in das Verzeichnis, wo Sie *summe.cpp* abgespeichert haben. Die Übersetzung, auch Compilation genannt, wird mit

```
g++ -o summe.exe summe.cpp
```

[4] Voraussetzung dazu ist die Installation des Compilers gemäß *http://cppbuch.de/downloads.html*.

gestartet. Das Programm wird durch Eintippen von *summe.exe* (oder *./summe.exe* unter Linux/macOS, wenn das aktuelle Verzeichnis nicht im Pfad ist) gestartet. Eigentlich verbergen sich hinter dem Aufruf des Compilers zwei Schritte:

`g++ -c summe.cpp`	compilieren (*summe.o* wird erzeugt)
`g++ -o summe.exe summe.o`	linken
`g++ -o summe.exe summe.cpp`	beide Schritte zusammengefasst.

Die Objektdateien können je nach System die Endung *.o* oder *.obj* tragen. Beim macOS verbirgt sich hinter g++ der Aufruf des zu Xcode gehörenden clang++-Compilers. Wenn `g++ -std=c++20 -o summe.exe summe.cpp` geschrieben wird, soll der Compiler den C++-Standard 2020 verwenden. Das ist bei diesem einfachen Programm nicht notwendig, weil es keine der neueren Eigenschaften nutzt.

Mit Visual Studio C++ unter Windows 10 funktioniert es ganz ähnlich. Zum Übersetzen reicht eine normale Konsole nicht aus, auch nicht eine Powershell. Sie brauchen einen »Developer Command Prompt«, eine Konsole, in der der Pfad für den Compiler und dazugehörige Programme gesetzt ist. Um den Developer Command Prompt zu öffnen, klicken Sie links unten im Windows-Hauptfenster auf das Start-Symbol. Tippen Sie »Develo« und schon wird Ihnen die Developer-Command-Prompt-App angeboten. Wenn Sie diese öffnen, erscheint eine geeignete Konsole. In dieser wechseln Sie in das Verzeichnis mit Ihrem Programm *summe.cpp*. Dieses wird so übersetzt:

`cl /c /std:c++latest /EHsc summe.cpp`	compilieren (*summe.obj* wird erzeugt)
`link summe.obj`	linken. Beide Schritte zusammengefasst:
`cl /std:c++latest /EHsc summe.cpp`	(wie oben, aber ohne /c)

Die Option `/std:c++latest` informiert den Compiler, die aktuellste C++-Version zu nehmen. Das Programm rufen Sie auf, indem Sie `summe.exe` in die Konsole tippen.

Übungen

1.1 Schreiben Sie mit einem ASCII-Editor (z.B. Notepad++ unter Windows) das Programm zur Summenberechnung ab, speichern es und bringen Sie es zum Laufen.

1.2 Schreiben Sie nunmehr an einer Stelle `caut` statt `cout` und lesen Sie die Fehlermeldungen des Compilers. Vermutlich erscheinen sie Ihnen etwas rätselhaft, aber das wird sich ändern.

1.5 Integrierte Entwicklungsumgebung

Integrierte Entwicklungsumgebungen (abgekürzt IDE für Integrated Development Environment) haben einen speziell auf Programmierzwecke zugeschnittenen Editor, der darüber hinaus auf Tastendruck oder Mausklick die Übersetzung anstößt. Es gibt einige davon. Ein paar Beispiele:

- Code::Blocks. Diese IDE ist Open Source. Es gibt sie für Windows, Linux und macOS (für macOS leider nur in einer älteren Version). Sie ist einfach zu konfigurieren und zu benutzen.
- Visual Studio C++ von Microsoft. Diese IDE gibt es nur für Windows. Die Community-Edition ist kostenlos.
- Visual Studio Code von Microsoft. Diese IDE gibt es für Windows, Linux und macOS. Sie ist Open Source und kostenlos, aber leider nicht ganz so einfach wie Code::Blocks zu konfigurieren.
- Eclipse und NetBeans sind auch beliebt. Sie haben die Programmiersprache Java als Schwerpunkt, es gibt aber jeweils ein Zusatzmodul (»Plug-in«) für C++.
- Xcode, die von Mac-Entwicklern bevorzugte IDE.

Die Arbeitsweisen der Entwicklungsumgebungen ähneln sich. Weil die IDE Code::Blocks wohl die einfachste ist und nicht auf ein Betriebssystem beschränkt ist, habe ich mich entschlossen, sie für eine Kurzvorstellung auszuwählen. Die anderen genannten Entwicklungsumgebungen sind deutlich mächtiger, aber der größere Funktionsumfang wird für den Einstieg nicht gebraucht.

In den Verzeichnissen der herunterladbaren Beispiele gibt es viele Dateien, die einzeln für sich ein Programm darstellen, wie das oben genannte Programm *summe.cpp*. Eine IDE ist zur Übersetzung vieler einzelner Programme nicht geeignet. Das erledigt besser das Programm make, das Sie in so einem Verzeichnis aufrufen können. Dass es viele einzelne Programmbeispiele gibt, liegt in der Natur eines Lehrbuchs über C++. Die industrielle Wirklichkeit sieht anders aus. Da besteht ein Programm aus vielen, manchmal Hunderten von Dateien. Das ist der Anwendungsbereich von Entwicklungsumgebungen: Sie verwalten viele zu einem Programm gehörende Dateien als sogenanntes Projekt. Im Folgenden wird am Beispiel die Erzeugung und Bearbeitung des einfachst möglichen Projekts gezeigt, nämlich eines Projekts, das nur aus einer *cpp*-Datei besteht.

1.5.1 Das erste C++-Projekt mit Code::Blocks

Im Folgenden wird angenommen, dass Sie die Entwicklungumgebung gemäß der Anleitung auf *http://cppbuch.de/downloads.html* installiert und entsprechend den Hinweisen zur Einrichtung konfiguriert haben. Um das erste C++-Projekt mit Code::Blocks zu starten, rufen Sie Code::Blocks auf und klicken »Create a new project« im erscheinenden Fenster an und dann »Console application« , »Next«, und »C++«. Geben Sie dann einen Projekttitel an, z.B. »ErstesProgramm« und den Ordner, in dem es angelegt werden soll. Das Ergebnis sieht dann ungefähr aus wie in Abbildung 1.3.

Im nächsten Fenster machen Sie ein Häkchen bei Debug, Release oder beidem. »Debug« bedeutet, dass im ausführbaren Programm Informationen enthalten sind, die eine Fehlersuche mit einem Debugger (= ein Programm zur Fehlersuche) erleichtern. In einer Release-Variante fehlen diese Informationen, weswegen das Programm weniger Platz beansprucht. Fürs Erste reicht Debug. Dann klicken Sie oben links auf das kleine Kreuz neben »Sources« und dann auf das erscheinende »main.cpp«. Nun erscheint eine Vorlage für das Hauptprogramm, die Sie modifizieren können. Ein Klick auf das Diskettensymbol oben links sichert die Datei. Um bei dem bekannten Beispiel zu bleiben, löschen Sie den angezeigten Text und tippen das bekannte Programm *summe.cpp* ein – oder Sie kopie-

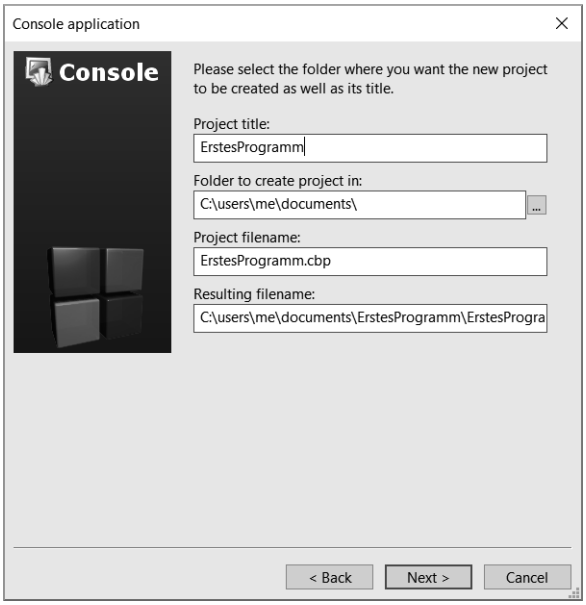

Abbildung 1.3: Erstes Projekt mit Code::Blocks anlegen

ren es. Mit der Taste F9 wird das Programm übersetzt und ausgeführt. Es erscheint ein Terminal mit der Ausgabe des Programms. Dort geben Sie nun zwei ganze Zahlen ein. Ein Druck auf die ENTER-Taste zeigt das Ergebnis an. Mit einem weiteren Tastendruck beenden Sie das Terminal. Wenn Sie im Fenster unten »Build log« anklicken, sehen Sie, was Code::Blocks getan hat: den Compiler mit den gewählten Optionen aufrufen und das Ergebnis im Verzeichnis *bin/Debug* ablegen (Abbildung 1.4). Falls Sie das Build Log nicht sehen: Mit F2 wird die Anzeige an- und ausgeschaltet.

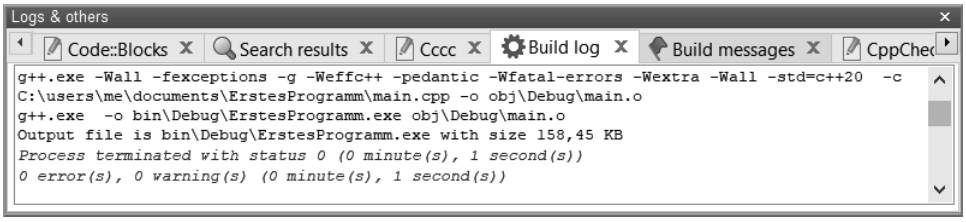

Abbildung 1.4: Build-Log

Um zu zeigen, wie sich Fehler auswirken, ändern Sie bei der Eingabezeile mit `cin` den Namen der Variablen `summe2` in `summex` ab und drücken wieder F9. Die Abbildung 1.5 zeigt den vom Compiler entdeckten Fehler an.

Nun korrigieren Sie den Fehler und drücken wieder F9. Jetzt wird das Programm wieder erfolgreich übersetzt und ausgeführt. Bei manchen Entwicklungsumgebungen schließt sich das Fenster zu schnell. Man kann dann eine Pause erzwingen. Dazu werden am Programmende die folgenden Zeilen hinzugefügt:

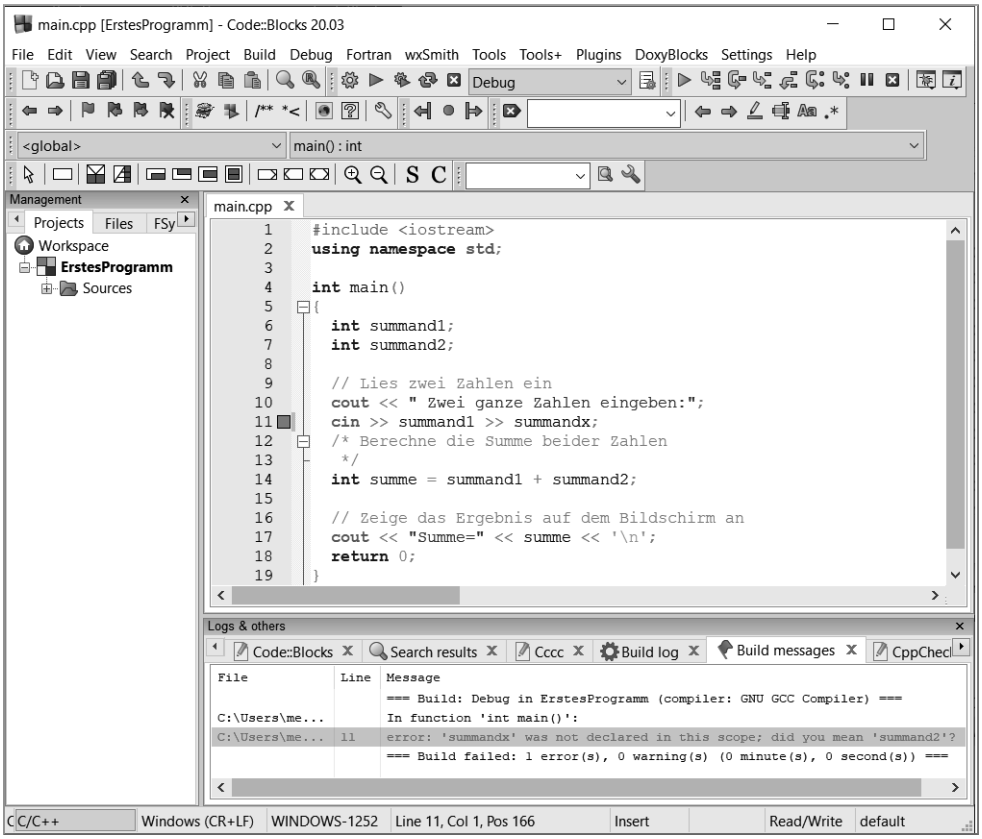

Abbildung 1.5: Code::Blocks zeigt einen Fehler an

```
cout << "Bitte␣drücken␣Sie␣Enter␣zum␣Beenden␣des␣Programms\n";
cin.ignore(1000, '\n');      // löscht alle Zeichen bis zum Zeilenende, aber max. 1000
cin.get();
```

Übungen

1.3 Bauen Sie verschiedene andere Fehler ein und versuchen Sie, die Fehlermeldungen zu verstehen.

1.4 Schreiben Sie mit dem Editor oder der IDE ein Programm, das eine kompliziertere Rechnung mit drei Variablen ausführt, zum Beispiel *c = (a1 + a2) * a3*. Bei der Eingabe können Sie pro Variable eine Zeile verwenden, etwa `cin << var1;`. Die Deklarationen dürfen nicht vergessen werden. *Hinweis*: Lösungen zu den meisten Aufgaben finden Sie im Anhang.

1.5.2 Xcode

Auf *http://www.cppbuch.de/swinstallation.html* finden Sie Hinweise zur Installation des Xcode-Compilers. Um ein Xcode-Projekt anzulegen, klicken Sie auf »Create a new Xcode

```
{
  auto x{zahl*10};              // Beginn des lokalen Gültigkeitsbereichs.
  cout << x << '\n';
}                               // Die Gültigkeit von x endet hier.
break;
```

Das Weglassen der geschweiften Klammern führt zu einer Fehlermeldung des Compilers. Wie bei der `if`-Anweisung ist es möglich, eine lokale Variable für die `switch()`-Anweisung anzulegen. Die Variable wird in den runden Klammern so initialisiert, wie Sie das auf Seite 69 bei der `if`-Anweisung sehen, also etwa `switch(int x=1; auswahl)`.

1.8.5 Wiederholungen

Häufig muss die gleiche Teilaufgabe oft wiederholt werden. Denken Sie nur an die Summation von Tabellenspalten in der Buchführung oder an das Suchen einer bestimmten Textstelle in einem Buch. In C++ gibt es zur Wiederholung von Anweisungen drei verschiedene Arten von Schleifen. In einer Schleife wird nach Abarbeitung einer Teilaufgabe (zum Beispiel Addition einer Zahl) wieder an den Anfang zurückgekehrt, um die gleiche Aufgabe noch einmal auszuführen (Addition der nächsten Zahl). Durch bestimmte Bedingungen gesteuert, zum Beispiel Ende der Tabelle, bricht irgendwann die Schleife ab.

Schleifen mit while

Abbildung 1.9 zeigt die Syntax von `while`-Schleifen.

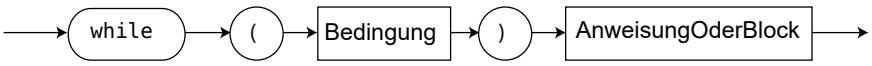

Abbildung 1.9: Syntaxdiagramm einer `while`-Schleife

AnweisungOderBlock ist wie auf Seite 67 definiert. Die Bedeutung einer `while`-Schleife ist: Solange die Bedingung wahr ist, die Auswertung also ein Ergebnis `true` oder ungleich 0 liefert, wird die Anweisung bzw. der Block ausgeführt. Die Bedingung wird auf jeden Fall zuerst geprüft. Wenn die Bedingung von vornherein unwahr ist, wird die Anweisung gar nicht erst ausgeführt (siehe Abbildung 1.10).

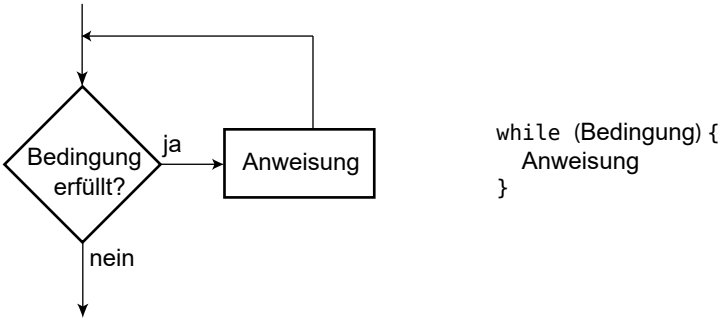

Abbildung 1.10: Flussdiagramm für eine `while`-Anweisung

Die Anweisung oder der Block innerhalb der Schleife heißt *Schleifenkörper*. Schleifen können wie `if`-Anweisungen beliebig geschachtelt werden.

```
while (Bedingung1)       // geschachtelte Schleifen, ohne und mit geschweiften Klammern
  while (Bedingung2) {
      .....
    while (Bedingung3) {
          .....
    }
  }
```

Beispiele

- Unendliche Schleife:

```
while (true)
   Anweisung
```

- Anweisung wird nie ausgeführt (unerreichbarer Programmcode):

```
while (false)
   Anweisung
```

- Summation der Zahlen 1 bis 99:

```
int sum {0};
constexpr int n {1};
constexpr int grenze {99};
while (n <= grenze) {
  sum += n++;
}
```

- Berechnung des größten gemeinsamen Teilers ggT(x, y) für zwei natürliche Zahlen x und y nach Euklid. Es gilt:
  - ggT(x, x), also x == y: Das Resultat ist x.
  - ggT(x, y) bleibt unverändert, falls die größere der beiden Zahlen durch die Differenz ersetzt wird, also ggT(x, y) == ggT(x, y-x), falls x < y.

  Das Ersetzen der Differenz geschieht im folgenden Beispiel iterativ, also durch eine Schleife.

Listing 1.15: Beispiel für `while`-Schleife *(cppbuch/k1/ggt.cpp)*

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

int main()
{
  int x {0};
  int y {0};
  cout << "2␣Zahlen␣>␣0␣eingeben␣:";
  cin >> x >> y;
  cout << "Der␣GGT␣von␣" << x << "␣und␣" << y << "␣ist␣";
  while (x != y) {
    if (x > y) {
```

```
      x -= y;
    }
    else {
      y -= x;
    }
  }
  cout << x << '\n';
}
```

Innerhalb einer Schleife muss es eine Veränderung derart geben, dass die Bedingung irgendwann einmal unwahr wird, sodass die Schleife abbricht (man sagt auch *terminiert*). Unbeabsichtigte »unendliche« Schleifen sind ein häufiger Programmierfehler. Im ggT-Beispiel ist leicht erkennbar, dass die Schleife irgendwann beendet sein *muss*:

1. Bei jedem Durchlauf wird mindestens eine der beiden Zahlen kleiner.
2. Die Zahl 0 kann nicht erreicht werden, da immer eine kleinere von einer größeren Zahl subtrahiert wird. Die `while`-Bedingung schließt die Subtraktion gleich großer Zahlen aus, und nur die könnte 0 ergeben.

Daraus allein ergibt sich, dass die Schleife beendet wird, und zwar in weniger als `x` Schritten, wenn `x` die anfangs größere Zahl war. Im Allgemeinen sind es erheblich weniger, wie eine genauere Analyse ergibt.

Tipp

Die Anweisungen zur Veränderung der Bedingung sollen möglichst an das Ende des Schleifenkörpers gestellt werden, um sie leicht finden zu können.

Schleifen mit do while

Abbildung 1.11 zeigt die Syntax einer `do while`-Schleife.

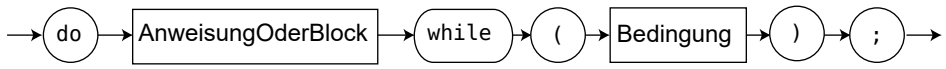

Abbildung 1.11: Syntaxdiagramm einer `do while`-Schleife

AnweisungOderBlock ist wie auf Seite 67 definiert. Die Anweisung oder der Block einer `do while`-Schleife wird ausgeführt, und *erst anschließend* wird die Bedingung geprüft. Ist sie wahr, wird die Anweisung ein weiteres Mal ausgeführt usw. Die Anweisung wird also mindestens einmal ausgeführt. Im Flussdiagramm ist die Anweisung ein Block (siehe rechts in der Abbildung 1.12).

`do while`-Schleifen eignen sich unter anderem gut zur sicheren Abfrage von Daten, indem die Abfrage so lange wiederholt wird, bis die abgefragten Daten in einem plausiblen Bereich liegen, wie im Primzahlprogramm unten zu sehen ist. Es empfiehlt sich zur besseren Lesbarkeit, `do while`-Schleifen strukturiert zu schreiben. Die schließende geschweifte Klammer soll genau unter dem ersten Zeichen der Zeile stehen, die die öffnende geschweifte Klammer enthält. Dadurch und durch Einrücken des dazwischen stehenden Textes ist sofort der Schleifenkörper erkennbar.

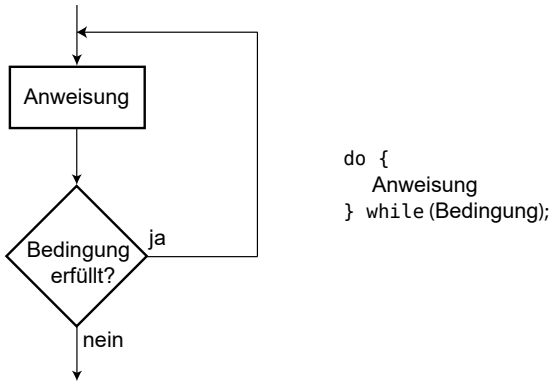

Abbildung 1.12: Flussdiagramm für eine `do while`-Anweisung

```
do {
    Anweisungen
} while (Bedingung);
```

Das *direkt hinter* die abschließende geschweifte Klammer geschriebene `while` macht unmittelbar deutlich, dass dieses `while` zu einem `do` gehört. Das ist besonders wichtig, wenn der Schleifenkörper in einer Programmliste über die Seitengrenze ragt. Eine `do while`-Schleife kann stets in eine `while`-Schleife umgeformt werden (und umgekehrt).

Listing 1.16: Berechnen einer Primzahl mit `do while` *(cppbuch/k1/primzahl.cpp)*

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

int main()
{
  cout << "Berechnung␣der␣ersten␣Primzahl,␣die␣>="
          "␣der␣eingegebenen␣Zahl␣ist\n";
  // Hinweis: Mehrere, durch " getrennte Texte ergeben eine lange Zeile in der Ausgabe.
  long zahl {0L};
  // do while-Schleife zur Eingabe und Plausibilitätskontrolle
  do {
    // Abfrage, solange zahl ≤ 3 ist
    cout << "Zahl␣>␣3␣eingeben␣:";
    cin >> zahl;
  } while (zahl <= 3);

  if (zahl % 2 == 0) {   // Falls zahl gerade ist, wird die nächste
                         // ungerade Zahl als Startwert genommen.
    ++zahl;
  }
  bool gefunden {false};
  do {
    // limit = Grenze, bis zu der gerechnet werden muss.
    // sqrt() arbeitet mit double, daher wird der Typ explizit umgewandelt.
    const long limit {1 + static_cast<long>(sqrt(static_cast<double>(zahl)))};
    long rest {0L};
```

```
    long teiler {1L};
    do {                        // Kandidat zahl durch alle ungeraden Teiler dividieren
      teiler += 2;
      rest = zahl % teiler;
    } while (rest > 0 && teiler < limit);

    if (rest > 0 && teiler >= limit) {
      gefunden = true;
    }
    else {                      // sonst nächste ungerade Zahl untersuchen
      zahl += 2;
    }
  } while (!gefunden);
  cout << "Die␣nächste␣Primzahl␣ist␣" << zahl << '\n';
}
```

Schleifen mit for

Die letzte Art von Schleifen ist die `for`-Schleife. Sie wird häufig eingesetzt, wenn die Anzahl der Wiederholungen vorher feststeht, aber das muss durchaus nicht so sein. Abbildung 1.13 zeigt die Syntax einer `for`-Schleife.

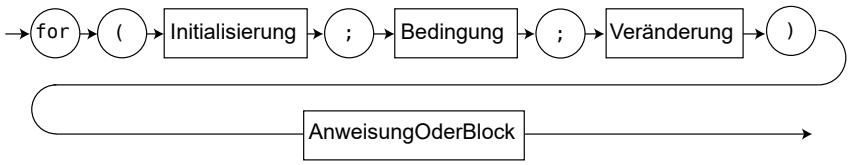

Abbildung 1.13: Syntaxdiagramm einer `for`-Schleife

Der zu wiederholende Teil (Anweisung oder Block) wird auch Schleifenkörper genannt. Beispiel: ASCII-Tabelle im Bereich 65 ... 69 ausgeben

```
for (int i = 65; i <= 69; ++i) {
    cout << i << "␣" << static_cast<char>(i) << '\n';
}
```

Bei der Abarbeitung werden die folgenden Schritte durchlaufen:

1. Durchführung der Initialisierung, zum Beispiel Startwert für eine Laufvariable festlegen. Eine Laufvariable wird wie `i` in der Beispielschleife als Zähler benutzt.
2. Prüfen der Bedingung.
3. Falls die Bedingung wahr ist, zuerst die Anweisung und dann die Veränderung ausführen.

Die Laufvariable `i` kann auch außerhalb der runden Klammern deklariert werden, dies gilt aber als schlechter Stil. Der Unterschied besteht darin, dass außerhalb der Klammern deklarierte Laufvariablen noch über die Schleife hinaus gültig sind.

```
int i;                                          // nicht empfohlen
for (i = 0; i < 100; ++i) {
    // Programmcode, i ist hier bekannt
```

```
}
// i ist weiterhin bekannt ...
```

Im Fall der Deklaration innerhalb der runden Klammern bleibt die Gültigkeit auf den Schleifenkörper beschränkt:

```
for (int i = 0; i < 100; ++i) {                    // empfohlen
    // Programmcode, i ist hier bekannt
}
// i ist hier nicht mehr bekannt
```

Die zweite Art erlaubt es, `for`-Schleifen als selbstständige Programmteile hinzuzufügen oder zu entfernen, ohne Deklarationen in anderen Schleifen ändern zu müssen. Derselbe Mechanismus gilt für Deklarationen in den runden Klammern von `if`-, `while`- und `switch`-Anweisungen.

Listing 1.17: Beispiel für `for`-Schleife *(cppbuch/k1/fakultaet.cpp)*

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

int main()
{
  cout << "Fakultät␣berechnen.␣Zahl␣>=␣0?␣:";
  int n {0};
  cin >> n;

  long fak {1L};
  for (int i = 2; i <= n; ++i) {
    fak *= i;
  }
  cout << n << "!␣␣␣=␣␣␣" << fak << '\n';
}
```

Verändern Sie niemals die Laufvariable innerhalb des Schleifenkörpers! Das Auffinden von Fehlern würde durch die Änderung erschwert.

```
for (int i = 65; i < 70; ++i) {
    // eine Seite Programmcode
    --i;  // irgendwo dazwischen erzeugt eine unendliche Schleife
    // noch mehr Programmcode
}
```

Auch wenn der Schleifenkörper nur aus einer Anweisung besteht, wird empfohlen, ihn in geschweiften Klammern `{ }` einzuschließen.

Äquivalenz von for und while

Eine `for`-Schleife entspricht direkt einer `while`-Schleife, sie ist im Grunde nur eine Umformulierung, solange nicht `continue` vorkommt (das im folgenden Abschnitt beschrieben wird):

```
for (Initialisierung; Bedingung; Veraenderung)
  Anweisung
```

ist äquivalent zu:

Objekt 1 wird erzeugt.
neuer Block
Objekt 2 wird erzeugt.
Block wird verlassen
Objekt 2 wird zerstört.
main wird verlassen
Objekt 1 wird zerstört.
Objekt 0 wird zerstört.

Der Destruktor von Objekten mit statischer Lebensdauer (`static` oder globale Objekte) wird nicht nur beim Verlassen eines Programms mit `return`, sondern auch bei Verlassen mit `exit()` aufgerufen. Im Gegensatz zum normalen Verlassen eines Blocks wird der Speicherplatz bei `exit()` jedoch nicht freigegeben.

3.7 Wie kommt man zu Klassen und Objekten? Ein Beispiel

Es kann hier keine allgemeine Methode gezeigt werden, wie man von einer Aufgabe zu Klassen und Objekten kommt. Es wird jedoch anhand eines Beispiels ein erster Eindruck vermittelt, wie der Weg von einer Problemstellung zum objektorientierten Programm aussehen kann.

Es geht hier um ein Programm, das zu einer gegebenen Personalnummer den Namen heraussucht. Ähnlichkeiten mit der Aufgabe 1.25 von Seite 111 sind beabsichtigt. Gegeben sei eine Datei *daten.txt* mit den Namen und den Personalnummern der Mitarbeiter. Dabei folgt auf eine Zeile mit dem Namen eine Zeile mit der Personalnummer. Das #-Zeichen ist die Endekennung. Der Inhalt der Datei ist:

```
Hans Nerd
06325927
Juliane Hacker
19236353
Michael Ueberflieger
73643563
#
```

Einige Analyse-Überlegungen

Um die Problemstellung zu verdeutlichen, wird sie aus verschiedenen Blickwinkeln betrachtet. Es handelt sich dabei nur um *Möglichkeiten*, nicht um den einzig wahren Lösungsansatz (den es nicht gibt).

1. In der Analyse geht es zunächst einmal darum, den typischen Anwendungsfall (englisch *use case*) *in der Sprache des (späteren Programm-)Anwenders* zu beschreiben.

Ein ganz konkreter Anwendungsfall, Szenario genannt, ist ein weiteres Hilfsmittel zum Verständnis dessen, was das Programm tun soll.

2. Im zweiten Schritt wird versucht, beteiligte Objekte, ihr Verhalten und ihr Zusammenwirken zu identifizieren.

Anwendungsfall (use case)

Das Programm wird gestartet. Alle Namen und Personalnummern werden zur Kontrolle ausgegeben (weil es hier nur wenige sind). Anschließend erfragt das Programm eine Personalnummer und gibt daraufhin den zugehörigen Namen aus oder aber die Meldung, dass der Name nicht gefunden wurde. Die Abfrage soll beliebig oft möglich sein. Wird X oder x eingegeben, beendet sich das Programm.

Für einen konkreten Anwendungsfall (= Szenario) wird die oben dargestellte Datei *daten.txt* verwendet.

Szenario

Das Programm wird gestartet und gibt aus:

Hans Nerd 06325927
Juliane Hacker 19236353
Michael Ueberflieger 73643563

Anschließend erfragt das Programm eine Personalnummer. Die Person vor dem Bildschirm (Benutzer/User) gibt 19236353 ein. Das Programm gibt »Juliane Hacker« aus und fragt wieder nach einer Personalnummer. Jetzt wird 99999 eingegeben. Das Programm meldet »nicht gefunden!« und fragt wieder nach einer Personalnummer. Jetzt wird X eingegeben. Das Programm beendet sich.

Objekte und Operationen identifizieren

Im nächsten Schritt wird versucht, die beteiligten Objekte und damit ihre Klassen zu identifizieren und eine Beschreibung ihres Verhaltens zu finden.

In der nicht-objektorientierten Lösung zur Vorläuferaufgabe 1.25 werden alle Aktivitäten in `main()` abgehandelt. Das ist unvorteilhaft, weil die Funktionalität damit nicht einfach in ein anderes Programm transportiert werden kann. Deswegen bietet es sich an, die Aktivitäten in ein eigens dafür geschaffenes Objekt zu verlegen. Die Klasse dazu sei hier etwas hochtrabend `Personalverwaltung` genannt. Was müsste so ein Objekt tun?

1. Die Datei *daten.txt* lesen und die gelesenen Daten speichern. Der Einfachheit halber wird hier angenommen, dass keine andere Datei zur Auswahl steht.
2. Die Daten auf dem Bildschirm *ausgeben*.
3. Einen *Dialog* mit dem Benutzer *führen*, in dem nach der Personalnummer gefragt wird.

Diese drei Punkte und die Kenntnis der Datei führen zu entsprechenden Schlussfolgerungen. Dabei sind im ersten Schritt die Substantive (Hauptworte) als Kandidaten für Klassen zu sehen und Verben (Tätigkeitsworte) als Methoden. Passivkonstruktionen sollen dabei

vorher stets in Aktivkonstruktionen verwandelt werden, d.h. *ausgeben* ist besser als *die Ausgabe erfolgt*.

1. Eine Auswahl der Datei ist hier nicht vorgesehen. Ein Objekt der Klasse `Personalverwaltung` soll daher schon beim Anlegen die Datei einlesen und die Daten speichern. Das übernimmt am besten der Konstruktor, dem der Dateiname übergeben wird.
   - Die gelesenen Daten gehören zu Personen. Jede *Person* hat einen Namen und eine Personalnummer. Es bietet sich an, Name und Personalnummer in einer Klasse `Person` zu kapseln. Aus Gründen der Einfachheit sollen Vor- und Nachname nicht getrennt gehalten werden; ein Name genügt.
   - Die Personalnummer soll nicht als `int` vorliegen, sondern als `string`, damit nicht führende Nullen (siehe Datei oben) beim Einlesen verschluckt werden oder zu einer Interpretation als Oktalzahl führen. Außerdem könnte es Nummernsysteme mit Buchstaben und Zahlen geben.
   - Die Klasse `Personalverwaltung` soll die Daten speichern. Dafür bietet sich ein `vector<Person>` als Attribut an.
2. Das Tätigkeitswort *ausgeben* legt nahe, eine gleichnamige Methode `ausgeben()` vorzusehen. In der Methode werden Name und Personalnummer einer Person ausgegeben. Es muss also entsprechende Methoden in der Klasse `Person` geben, etwa `getName()` und `getPersonalnummer()`. Diese Methoden würden innerhalb der Funktion `ausgeben()` aufgerufen werden.
3. *Dialog führen* legt nahe, eine Methode `dialogfuehren()` oder kurz `dialog()` vorzusehen.

Weil nur ein erster Eindruck vermittelt werden soll und die Problemstellung einfach ist, wird auf eine vollständige objektorientierte Analyse (OOA) und ein entsprechendes Design (OOD) verzichtet und auf die Literatur verwiesen, die die OOA/D-Thematik behandelt, zum Beispiel [Oe]. In diesem einfachen Fall konzentrieren wir uns gleich auf eine Lösung mit C++. Ein mögliches `main()`-Programm könnte wie folgt aussehen:

Listing 3.35: `main`-Programm zur Personalverwaltung *(cppbuch/k3/personalverwaltung/main.cpp)*

```
#include "Personalverwaltung.h"
#include <iostream>
using namespace std;

int main()
{
  Personalverwaltung personalverwaltung("daten.txt");
  cout << "Gelesene Namen und Personalnummern:\n";
  personalverwaltung.ausgeben();
  personalverwaltung.dialog();
  cout << "Programmende\n";
}
```

Die Klasse `Person` ist einfach zu entwerfen:

Listing 3.36: Klasse `Person` *(cppbuch/k3/personalverwaltung/Person.h)*

```
#ifndef PERSON_H
#define PERSON_H
```

```
#include <string>
class Person {
public:
  Person(const std::string& name_, const std::string& personalnummer_)
      : name{name_}, personalnummer{personalnummer_}
  {
  }

  const auto& getName() const { return name; }

  const auto& getPersonalnummer() const { return personalnummer; }

private:
  std::string name;
  std::string personalnummer;
};
#endif
```

Auch die Klasse `Personalverwaltung` ist nach den obigen Ausführungen nicht schwierig, wenn man sich zunächst auf die Prototypen der Methoden beschränkt:

Listing 3.37: Klasse `Personalverwaltung` *(cppbuch/k3/personalverwaltung/Personalverwaltung.h)*

```
#ifndef PERSONALVERWALTUNG_H
#define PERSONALVERWALTUNG_H
#include "Person.h"
#include <vector>

class Personalverwaltung {
public:
  explicit Personalverwaltung(const std::string& dateiname);

  void ausgeben() const;

  void dialog() const;

private:
  std::vector<Person> personal;
};
#endif
```

Für die Implementierung der Methoden der Klasse `Personalverwaltung` muss man sich mehr Gedanken machen. Das überlasse ich Ihnen (siehe die nächste Aufgabe)! Die Lösung dürfte aber nicht schwer sein, wenn Sie die Aufgabe 1.25 von Seite 111 gelöst oder deren Lösung nachgesehen haben.

Übungen

3.5 Implementieren Sie die oben deklarierten Methoden der Klasse `Personalverwaltung` in einer Datei *Personalverwaltung.cpp*.

3.6 Wie können Sie mit C++ erreichen, dass ein Attribut *direkt*, also ohne Einsatz einer Methode, zwar gelesen, aber nicht verändert werden kann? Beispiel:

22

Von der UML nach C++

Dieses Kapitel behandelt die folgenden Themen:

- Vererbung
- Interfaces
- Assoziationen
- Multiplizität
- Aggregation
- Komposition

Die Unified Modeling Language (UML) ist eine weit verbreitete grafische Beschreibungssprache für Klassen, Objekte, Zustände, Abläufe und noch mehr. Sie wird vornehmlich in der Phase der Analyse und des Softwareentwurfs eingesetzt. Auf die UML-Grundlagen wird hier nicht eingegangen; dafür gibt es gute Bücher wie [Oe]. Hier geht es darum, die wichtigsten UML-Elemente aus Klassendiagrammen in C++-Konstruktionen, die der Bedeutung des Diagramms möglichst gut entsprechen, umzusetzen. Die vorgestellten C++-Konstruktionen sind Muster, die als Vorlage dienen können. Diese Muster sind nicht einzigartig, sondern nur Empfehlungen, die Umsetzung zu gestalten. Im Einzelfall kann eine Variation sinnvoll sein.

22.1 Vererbung

Über Vererbung als »ist ein«-Beziehung wurde in diesem Buch schon einiges gesagt, was hier nicht wiederholt werden muss. Sie finden alles dazu in Kapitel 6. Die Abbildung 22.1 zeigt das zugehörige UML-Diagramm.

Abbildung 22.1: Vererbung (»ist ein«-Beziehung)

In vielen Darstellungen wird die Oberklasse oberhalb der abgeleiteten Unterklasse dargestellt; in der UML ist aber nur der Pfeil mit dem Dreieck entscheidend, nicht die relative Lage. In C++ wird Vererbung syntaktisch durch »`: public`« ausgedrückt:

Listing 22.1: Syntaktische Repräsentation der Vererbung

```
class Unterklasse : public Oberklasse {
  // ... Rest weggelassen
};
```

22.2 Interface anbieten und nutzen

Interface anbieten

Abbildung 22.2 zeigt das zugehörige UML-Diagramm. Die Klasse Anbieter implementiert das Interface Schnittstelle-X. Bei der Vererbung stellt die abgeleitete Klasse die Schnittstelle der Oberklasse zur Verfügung. Insofern gibt es eine Ähnlichkeit, auch gekennzeichnet durch die gestrichelte Linie im Vergleich zum vorherigen Diagramm.

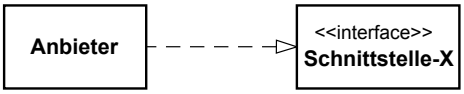

Abbildung 22.2: Interface-Anbieter

Die Ähnlichkeit wird in der Umsetzung nach C++ abgebildet: `Anbieter` wird von dem Interface `SchnittstelleX`[1] abgeleitet. Um klarzustellen, dass es um ein Interface geht, soll `SchnittstelleX` abstrakt sein. Das Datenobjekt `d` wird nicht als `const`-Referenz übergeben, weil `service()` damit auch die Ergebnisse an den Aufrufer übermittelt. Ein einfaches Programmbeispiel finden Sie im Verzeichnis *cppbuch/k22/interface*.

[1] Die UML erlaubt Bindestriche in Namen, C++ nicht.

Listing 22.2: Schnittstellenklasse

```
class SchnittstelleX {
public:
  virtual void service(Daten& d) = 0;       // abstrakte Klasse
  virtual ~SchnittstelleX() = default;      // virtueller Destruktor
  SchnittstelleX() = default;
  SchnittstelleX(const SchnittstelleX&) = delete;
  SchnittstelleX& operator=(const SchnittstelleX&) = delete;
};

class Anbieter : public SchnittstelleX {
public:
  void service(Daten& d)
  {
    // ... Implementation der Schnittstelle
  }
};
```

Interface nutzen

Bei der Nutzung des Interfaces bedient sich der Nutzer einer entsprechenden Methode des Anbieters. Die Abbildung 22.3 zeigt das zugehörige UML-Diagramm.

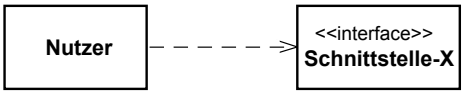

Abbildung 22.3: Interface-Nutzer

Ein `Nutzer` muss ein `Anbieter`-Objekt kennen, damit der Service genutzt werden kann. Aus diesem Grund wird in der folgenden Klasse bereits dem Konstruktor von `Nutzer` ein `Anbieter`-Objekt übergeben, und zwar per Referenz, nicht per Zeiger. Der Grund: Zeiger können `nullptr` sein, aber undefinierte Referenzen gibt es nicht.

Listing 22.3: Nutzer der Schnittstelle

```
class Nutzer {
public:
  Nutzer(SchnittstelleX& a)
  : anbieter(a)
  {
    daten = ...
  }

  void nutzen()
  {
    anbieter.service(daten);
  }
private:
  Daten daten;
  SchnittstelleX& anbieter;
};
```

Warum wird die Referenz oben nicht als `const` übergeben? Das kann je nach Anwendungsfall sinnvoll sein oder auch nicht. Es hängt davon ab, ob sich der Zustand des `Anbieter`-Objekts durch den Aufruf der Funktion `service(daten)` ändert. Wenn ja, zum Beispiel durch interne Protokollierung der Aufrufe, entfällt `const`.

22.3 Assoziation

Eine Assoziation sagt zunächt einmal nur aus, dass zwei Klassen in einer Beziehung (mit Ausnahme der Vererbung) stehen. Die Art der Beziehung und zu wie vielen Objekten sie aufgebaut wird, kann variieren. In der Regel gelten Assoziationen während der Lebensdauer der beteiligten Objekte. Nur kurzzeitige Verbindungen werden meistens nicht notiert. Ein Beispiel für eine kurzzeitige Verbindung ist der Aufruf `anbieter.service(daten);`. `anbieter` kennt durch die Parameterübergabe das Objekt `daten`, wird aber vermutlich die Verbindung nach Ablauf der Funktion lösen.

Einfache gerichtete Assoziation

Die Abbildung 22.4 zeigt das UML-Diagramm einer einfachen gerichteten Assoziation.

Abbildung 22.4: Gerichtete Assoziation

Mit »gerichtet« ist gemeint, dass die Umkehrung nicht gilt, wie zum Beispiel die Beziehung »ist Vater von«. Falls zwar Klasse1 die Klasse2 kennt, aber nicht umgekehrt, wird dies durch ein kleines Kreuz bei Klasse1 vermerkt. Es kann natürlich sein, dass eine Beziehung zwischen zwei Objekten *derselben* Klasse besteht. Im UML-Diagramm führt dann der von einer Klasse ausgehende Pfeil auf dieselbe Klasse zurück. In C++ wird eine einfache gerichtete Assoziation durch ein Attribut `zeigerAufKlasse2` realisiert:

Listing 22.4: Gerichtete Assoziation: `Klasse1` kennt `Klasse2`

```
class Klasse1 {
public:
  Klasse1()
  : zeigerAufKlasse2(nullptr)
  { }

  void setKlasse2(Klasse2* ptr2)
  {
    zeigerAufKlasse2 = ptr2;
  }
private:
  Klasse2* zeigerAufKlasse2;
};
```

Ein Zeiger ist hier besser als eine Referenz geeignet, weil es sein kann, dass das Kennenlernen erst nach dem Konstruktoraufruf geschieht.

Gerichtete Assoziation mit Multiplizität

Die Multiplizität, auch Kardinalität genannt, gibt an, zu wie vielen Objekten eine Verbindung aufgebaut werden kann. In Abbildung 22.5 bedeutet die 1, dass jedes Objekt der Klasse2 zu genau einem Objekt der Klasse1 gehört. Das Sternchen * bei Klasse2 besagt, dass einem Objekt der Klasse1 beliebig viele Objekte der Klasse2 zugeordnet sind, also möglicherweise auch keins.

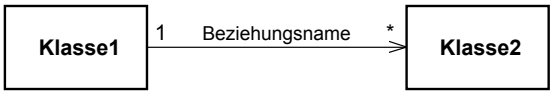

Abbildung 22.5: Gerichtete Assoziation mit Multiplizitäten

Im folgenden C++-Beispiel entspricht `Fan` der Klasse1 und `Popstar` der Klasse2. Ein Fan kennt N Popstars. Die Beziehung ist also »kennt«. Der Popstar hingegen kennt seine Fans im Allgemeinen nicht. Um die Multiplizität auszudrücken, bietet sich ein `vector` an, der Verweise auf `Popstar`-Objekte speichert. Wenn die Verweise eindeutig sein sollen, ist ein `set` die bessere Wahl.

Listing 22.5: Gerichtete Assoziation mit Multiplizität: Ein `Fan` kennt `Popstar`s, aber nicht umgekehrt.

```
class Fan {
public:
  void werdeFanVon(Popstar* star)
  {
    meineStars.insert(star);                // einfügen
  }

  void denKannsteVergessen(Popstar* star)
  {
    meineStars.erase(star);                 // entfernen. Rückgabewert ignoriert
  }
  // Rest weggelassen

private:
  std::set<Popstar*> meineStars;
};
```

Die Objekte als Kopie abzulegen, also `Popstar` als Typ für den Set statt `Popstar*` zu nehmen, hat Nachteile. Erstens ist es wenig sinnvoll, die Kopie zu erzeugen, wenn es doch das Original gibt, und zweitens kostet es Speicherplatz und Laufzeit. Es gibt nur einen Vorteil: Es könnte ja sein, dass es das originale `Popstar`-Objekt nicht mehr gibt, zum Beispiel durch ein `delete` irgendwo. Ein noch existierender Zeiger wäre danach auf eine undefinierte Speicherstelle gerichtet. Eine noch existierende Kopie könnte als Wiedergänger auftreten.

Einfache ungerichtete Assoziation

Eine ungerichtete Assoziation wirkt in beiden Richtungen und heißt deswegen auch bidirektionale Assoziation. Die Abbildung 22.6 zeigt das UML-Diagramm.

Abbildung 22.6: Ungerichtete Assoziation

Wenn zwei sich kennenlernen, kann das mit einer ungerichteten Assoziation modelliert werden. Zur Abwechslung sei die Umsetzung in C++ nicht mit zwei, sondern nur mit einer Klasse (namens `Person`) gezeigt. Das heißt, die Klasse hat eine Beziehung zu sich selbst, siehe Abbildung 22.7. Solche Assoziationen werden auch rekursiv genannt und dienen zur Darstellung der Beziehung verschiedener Objekte derselben Klasse.

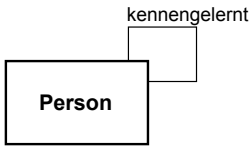

Abbildung 22.7: Rekursive Assoziation

Die Umsetzung in C++ wird am Beispiel von Personen gezeigt, die sich gegenseitig kennenlernen. Ein Aufruf `A.lerntkennen(B);` impliziert, dass `B` auch `A` kennenlernt. Natürlich kann es vorkommen, dass es zwei Personen mit demselben Namen gibt, hier Frau Holle.

Listing 22.6: Assoziation: Personen lernen sich kennen *(cppbuch/k22/bidirektAssoziation/main.cpp)*

```
#include "Person.h"

int main()
{
  Person mabuse("Dr. Mabuse");
  Person klicko("Witwe Klicko");
  Person holle1("Frau Holle");
  Person holle2("Frau Holle");                  // eine Namensvetterin!
  mabuse.lerntkennen(klicko);
  holle1.lerntkennen(klicko);
  holle1.lerntkennen(holle2);
  mabuse.bekannteZeigen();
  klicko.bekannteZeigen();
  holle1.bekannteZeigen();
}
```

Die entscheidende Methode der Klasse `Person` ist `lerntkennen(Person& p)` (siehe unten). Beim Eintrag in die Menge der Bekannten wird festgestellt, ob der Eintrag vorher schon vorhanden war. Wenn nicht, wird er auch auf der Gegenseite vorgenommen. Wenn in der Menge der Bekannten nur die Namen als String gespeichert würden, könnten sich zwei Personen mit demselben Namen nicht kennenlernen. Deswegen werden die Adressen der `Person`-Objekte gespeichert (siehe Attribut `set<Person*>` in der Klasse unten).

Listing 22.7: Klasse `Person` *(cppbuch/k22/bidirektAssoziation/Person.h)*

```
#ifndef PERSON_H
#define PERSON_H
#include <iostream>
#include <set>
#include <string>

class Person {
public:
  Person(const std::string& name_)
  : name(name_)
  {}

  const std::string& getName() const
  {
    return name;
  }

  void lerntkennen(Person& p)
  {
    bool nichtvorhanden = bekannte.insert(&p).second;
    if (nichtvorhanden) {          // falls unbekannt, auch bei p eintragen
      p.lerntkennen(*this);
    }
  }

  void bekannteZeigen() const
  {
    std::cout << "Die Bekannten von " << getName() << " sind:\n";
    for (auto bekannt : bekannte) {
      std::cout << bekannt->getName() << '\n';
    }
  }

private:
  std::string name;
  std::set<Person*> bekannte;
};
#endif
```

22.3.1 Aggregation

Die »Teil-Ganzes«-Beziehung (englisch *part of*) wird auch *Aggregation* genannt. Sie besagt, dass ein Objekt aus mehreren Teilen besteht (die wiederum aus Teilen bestehen können). Die Abbildung 22.8 zeigt das UML-Diagramm. Die Struktur entspricht der gerichteten Assoziation, sodass deren Umsetzung in C++ hier Anwendung finden kann. Ein Teil kann für sich allein bestehen, also auch vom Ganzen gelöst werden. Letzteres geschieht in C++ durch Nullsetzen des entsprechenden Zeigers.

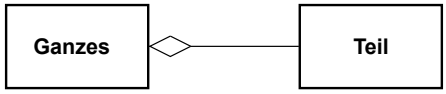

Abbildung 22.8: Aggregation

22.3.2 Komposition

Die Komposition ist eine spezielle Art der Aggregation, bei der die Existenz der Teile vom Ganzen abhängt. Damit ist gemeint, dass die Teile zusammen mit dem Ganzen erzeugt und auch wieder vernichtet werden. Ein Teil ist somit stets genau einem Ganzen zugeordnet; die Multiplizität kann also nur 1 sein. Formal ist auch 0 erlaubt. Für ein isoliertes Objekt ist jedoch der Begriff »Teil« nicht sinnvoll. Die Abbildung 22.9 zeigt das UML-Diagramm.

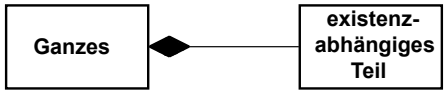

Abbildung 22.9: Komposition

Es empfiehlt sich, bei der Umsetzung in C++ Werte statt Zeiger zu nehmen. Dann ist gewährleistet, dass die Lebensdauer der Teile an das Ganze gebunden ist:

Listing 22.8: Umsetzung der Komposition

```
class Ganzes {
public:
  Ganzes(int datenFuerTeil1, int datenFuerTeil2)
  : ersterTeil(datenFuerTeil1),
    zweiterTeil(datenFuerTeil2)
  {
    // ...
  }
  // ...

private:
  Teil ersterTeil;
  Teil zweiterTeil;
};
```

Register

Symbole

* 46, 214, 453
*= 46, 372
+, +=, -, -= 46
++ 46, 374, 449, 453
, 60, 82, 503
-> 217
->* 263
-- 46, 378
.* 263
... *siehe* Ellipse
/, /= 46
/* ... */ 31
// 31
:: 62, 174, 311
::* 263
; 33, 71
<, <=, >, >= 46, 57
<=> 46, 388
<< 46, 106, 256, 361, 417
<<=, >>= 46
=, == 46, 57, 58
>> 46, 104, 256, 415, 440
?: 71
[ ] 218, 222, 366
[ ][ ]
  Matrixklasse 404
  Zeigerdarstellung 246
139
%, %= 46
& Adress-Operator 121
&, &= Bit-Operatoren 46
&& logisches UND 58
&& Shell 640
&& R-Wert 470
\ 31, 55, 137
\0 223, 227–229
\", \a, \b 55
\f, \n, \r, \t, \v 55, 104
\x 55
\\ 55
~ 46
^ 46
|= 46
|| 58
! 46, 58
!= 57, 58, 453
$<, $^, $@ 637
@D, @F 647
” 33, 223, 224

A

abgeleitete Klasse 298, 306, 310
  und virtueller Destruktor 321
Abhängigkeit (make) 634
  automatische Ermittlung 639

abort() 947
abs() 52, 776, 777, 945, 946
abstrakter Datentyp 169, 969
abstrakte Klasse 315, 969
accumulate() 724
acos() 777, 945
acosh() 777
Adapter, Iterator- 889
Additionsoperator 359
adjacent_difference() 728
adjacent_find() 759
adjustfield 426
Adresse 214
  symbolische 34
Adressoperator 214
advance() 887
Aggregat 89, 220, 852
Aggregation 699, 969
Aktualparameter 116
<algorithm> 701, 790, 895
Algorithmus 28, 449
  accumulate() 724
  adjacent_difference() 728
  adjacent_find() 759
  all_of, any_of 738
  binary_search() 761
  clamp() 798
  copy(), copy_backward() 787
  copy_if(), copy_n() 789
  count(), count_if() 737
  equal() 775
  equal_range() 763
  exclusive_scan() 728
  fill(), fill_n() 722
  find(), find_if(), find_if_not() 754
  find_end() 758
  find_first_of() 755
  for_each(), for_each_n() 786
  generate(), _n() 723
  includes() 764
  inclusive_scan() 727
  inner_product() 726
  inplace_merge() 753
  iota() 723
  is_heap(), is_heap_until() 772
  is_partitioned() 745
  is_permutation() 740
  is_sorted(), is_sorted_until() 747
  iter_swap() 790
  lexicographical_compare() 742
  lexicographical_compare_three_way() 743
  lower_bound() 762
  make_heap() 771
  make_pair() 824
  make_tupel() 825
  max(...) 796
  max_element() 730
  merge() 752
  mergesort() 753
  min(...) 796
  min_element() 730
  minmax() 797
  minmax_element() 730
  mismatch() 773
  move(), move_backward() 822
  next_permutation() 739
  none_of 738
  nth_element() 750
  partial_sort(), -_copy 749
  partial_sum() 727
  partition() 744
  partition_copy(), partition_point() 744
  pop_heap() 770
  prev_permutation() 739
  push_heap() 771
  remove(), -_if(), -_copy(), -_copy_if() 794
  replace(), -_if(), -_copy(), -_copy_if() 792
  reverse(), reverse_copy() 735
  rotate(), rotate_copy() 731
  sample() 736
  search() 756, 758
  search_n() 760
  set_difference() 767
  set_intersection() 766
  set_symmetric_difference() 768
  set_union() 765
  shuffle() 732, 733
  sort() 747

sort_heap() 772
split() 702
stable_partition() 744
stable_sort() 747
swap(), swap_ranges() 790
transform() 791
transform_exclusive_scan() 728
transform_inclusive_scan() 728
unique(), unique_copy() 733
upper_bound() 762
Alias-Name 121, 215, 217
*this 240
alignment_of, alignof() 511
all_of 738
allgemeiner Konstruktor 178
allocator 818
alternative Funktions-Syntax 281
Anführungszeichen 33, 223, 224
Anker (bei regulärem Ausdruck) 522
anonymer Namespace 150
Anweisung 64
any() (Bitset) 830
any_of 738
app 437
append()
Pfad 806
String 926
apply() 827
Äquivalenz 388, 762
-klasse 656
arg() 776, 777
argc 239
Argument *siehe* Parameter
Argument Dependent Lookup (ADL) 970
argv[] 239
Arität (Template) 497
Arithmetik
mit Iteratoren 452
mit Zeigern 220
arithmetische Operatoren 46
Array
char 225
von C-Strings 225
dynamisches 230, 246, 247, 364
Freigabe 234
als Funktionsparameter 236, 243
mehrdimensionales 242, 246, 247
vs. Zeiger 219
zweidimensionales
Matrixklasse 404, 406
Array2d 249
<array> 842, 851
ASCII 905
Dateien 256
Tabelle 56, **951**
asctime() 950
asin() 777, 945
asinh() 777
assert() 139
assign() 848, 927
assignable_from 518
Assoziation (UML) 696
Assoziativität von Operatoren 59, 956
async() 559
at() 91, 94, 849, 852, 857, 867, 876, 925
atan() 777, 945
atan2() 945
atanh() 777
ate 437
atexit() 947
atof(), atoi(), atol() 947
Atom-Uhr 601
atomic 556
atomic_ref 558
Attribut
Compilersteuerung 957
einer Klasse 171, 970
Aufforderung **970**
Aufzählungstyp 85
Ausdruck
Auswertung 59, 60
Definition 40
mathematischer 52, 128
Ausgabe **104**, 106, **417**
benutzerdefinierter Typen 418
Datei- 107, 413
Formatierung
mit Flags 424
mit std::format() 419
mit Manipulatoren 427
Weite der 421, 424
Ausgabeoperator 361, 417

Ausnahme 345, 970
Auswertungsreihenfolge 52, 60, 956
auto 99, 458, 680
als Rückgabetyp 155, 176, 313
auto[] 101, 825
automatische Variable 146
make 637
Autotools (GNU) 653

B

back() 849, 852, 854, 857, 859, 925
back_inserter(), -_insert_iterator 891
Backslash
Zeichenkonstante \ 56
Zeilenfortsetzung 137
Backspace 56
bad() 436
bad_alloc 351
badbit 435
bad_cast 336, 351
bad_function_call 351
bad_typeid 339, 351
bad_weak_ptr 351
base() 889
basefield 426
basic_string 923
basic_-Streamklassen 414
Basisklasse **298**
und virtueller Destruktor 321
Konstruktor 330
Subobjekt 333
virtuelle 332
Bedingungsausdruck 67, 68
Bedingungsoperator ?: 71
beg 438
begin()
Container 455, 844
Namespace std 222, 845, 888
string 273, 924
vector 237
Belegungsgrad 873
benutzerdefinierte
Datentypen 84, 87
Klassen 361
Literale 401
Typen (Ausgabe) 418
Typen (Eingabe) 440
Benutzungszählung 938
Bereichsnotation 846, 972
Bereichsoperator :: 62, 174, 311
namespace 148
Bibliothek
C 162, 943
C++ 816
Bibliotheksmodul 142
dynamisch 650
statisch 648
Bidirectional-Iterator 883
Big Three (Regel) 254
Big Three/Five/Zero (Regel) 483
Binärdatei 258
binäre Ein-/Ausgabe 255
binärer Operator 359
optimiert 480
binäres Prädikat 734, 774, 896
binäre Zahlendarstellung 47
Binärzahl 43
binary 109, 437
binary_search() 761
bind 833
Binden 142
dynamisches *siehe* dynamisches B.
statisches *siehe* statisches Binden
Bit
-feld 101
Operatoren 46, 47
Verschiebung 47
pro Zahl 41
bitset 827
bitweises ODER, UND, XOR 46
Block 31, 33, 62, **66**, 203
und dyn. Objekte 233
bool 57
boolalpha 58, 425, 428
break 72, 82
bsearch() 947
-Bstatic (Makefile) 649
Bucket 872
bucket() 875
Byte 54
std::byte 87, 946
Reihenfolge 602

C

C++-Schlüsselwörter 953
C-Arrays 218
C-Funktionen einbinden 163
C-Header 943
C-String 223
call wrapper 541
Callback-Funktion 260, 835
`canonical()` (Filesystem) 806
`capacity()` 850, 926
capturing group 520
`case` 73
  lokale Variablen 74
`<cassert>` 139, 943
cast *siehe* Typumwandlung
`catch` 346
`cbegin()`
  Container 844
  Namespace `std` 845, 888
  `string` 925
`<cctype>` **944**
`<cctype>` 791
cdecl 267
`ceil()` 945
`cend()`
  Container 844
  Namespace `std` 845, 888
  `string` 925
`cerr` 104, 414
`char` 54, 221
`char*` 223
`CHAR_BIT` (Bits pro Byte) 603
`char* const` vs. `const* char` 217
`char_traits` 224, 279, 513
`char8_t`, `char16_t`, `char32_t` 400, 906
`<chrono>` 532
`cin` 33, 104, 414, 436
`clamp()` 798
`class` 172, 304
`class` (bei Template-Parametern) 156
`clear()` 436, 848, 863, 928
Client-Server
  Beziehung 173
  und callback 260
`clock()`, `clock_t` 950
`clog` 104, 414
`close()` 108
CMake 653
`<cmath>` **945**
`<cmath>` 52, 817, 943
`co_await` 567
code bloat 295
Code-Formatierung 112
`collate` 910
`combine()` 904
`compare()` 910, 930
`<compare>` 388
Compilationsmodell 295
Compiler 29, 31, **34**, 142, 171, 173
  -befehle 954
  -direktiven 134
  und Templates 159
  Typumwandlung 196
`<complex>` 776
Computerarithmetik 52
Concepts 514
`conditional` 513
`configure` 653
`conj()` 777
`connect()` 572
`const` 53
  correctness 175
  Elementfunktionen 172, **174**
  globale Konstante 147
`const&` *siehe* Referenz auf `const`
`const_cast<>()` 337
`const char*` vs. `char* const` 217
`consteval` 154
`constexpr`
  Funktion 152
  Konstante 53
  Konstruktor / Methode 190
`constinit` 290
`const_iterator` 843
`const_local_iterator` 875
`const_pointer` 843
`const_reference` 843
`const_reverse_iterator` 845, 890
constraint, Vererben von 326
Container 448
  implizite Datentypen 843
  Methoden 844

Container-Adapter 858
container_type 858
contains() 864
Contiguous-Iterator 883
continue 82
convertible_to 518
copy semantics 472
copy() (Filesystem) 808
copy() 787, 925
copy_backward() 787
copy_if() 789
copy_n() 789
copy_constructible 518
copy_options 809
copy_result (Ranges) 721
copysign() 945
co_return 565
Coroutinen 564
cos(), cosh() 776, 945
count() 830, 864
  Algorithmus 737
count_if() 737
cout 33, 104, 414
co_yield 564
__cplusplus 164
crbegin()
  Container 845
  Namespace std 845, 888
  string 925
create_directory() 809
crend()
  Container 845
  Namespace std 845, 888
  string 925
critical section 544
CRLF 605
<cstddef> **946**
<cstddef> 48, 216
<cstdlib> 127, 259
c_str() 925
<cstring> **947**
<cstring> 223, 261, 271, 274
<ctime> **949**
<ctime> 376
ctype 708, 710, 910, 911
cur 438
curr_symbol 915
current_path 806
<cwctype> 944
CXXFLAGS 636

D

dangling (Ranges) 722
dangling pointer 233
data race 544, 546
data() 849, 852, 925
Datagramm 600
date_order() 917
Datei
  ASCII 256
  binär 258
  Ein-/Ausgabe 107, 413
  kopieren 109, 808
  löschen 806
  Öffnungsarten 437
  öffnen 109
  Positionierung 438
  schließen 108
  umbenennen 810
  Zugriffsrechte 803
Daten
  als Attributwerte 971
  static-Element- 284
Datenbankanbindung 621
Datenkapselung 971
Datentypen 34, 40, **169**
  abstrakte *siehe* abstrakter Datentyp
  benutzerdefinierte 84
  int und unsigned 61, 71
  logische 57
  parametrisierte 156, 291
  polymorphe 319
  strukturierte 87
  zusammengesetzte 84
Datum
  Klasse / Gültigkeit 374
  regulärer Ausdruck 778
daytime (Port 13) 595
Deadlock 563
dec 425, 428
decay, decay_t 512

decimal_point() 913, 914
decltype 281
decltype(auto) 283, 509, 822
declval 284
default (in switch-Anweisung) 73
= default 186
default constructor 176
default_delete<X[]> 686
defaultfloat 425
Default-Parameter *siehe* vorgegebene P.
#define 134, 137
Definition **144**, 971
  von static-Elementdaten 284
  von Objekten 178
Deklaration 33, 34, **144**, 971
  einer Funktion 115
  Funktionszeiger 258
  Lesen einer D. 264
  in for-Schleifen 80
Deklarationsanweisung 65
Dekrementierung 46
Dekrementoperator 378
Delegation 342
delegierender Konstruktor 189
delete 230, 232, 287, 320, 321
  überladen 393
delete [ ] 234
= delete 186, 682
Deleter 686, 687
deque 856
<deque> 842, 856
Dereferenzierung 214, 260
derived_from 518
Destruktor **203**, 331, 483
  implizite Deklaration 203
  und exit() 205
  virtueller 320
detach() 537
Dezimalpunkt 49, 913
Dialog 584
diamond problem 333
difference_type 843
Differenz (Menge) 767
difftime() 950
digits, digits10 796
distance() 887
Distribution 639, 971
div(), div_t 946
divides 832
Division durch 0 352
DNS 593
do while **77**
domain_error 351
double 49, 220
  nicht als Laufvariable 81
  korrekter Vergleich 678
downcast 336
Drei (die großen Drei) 254, 483
Drei-Wege-Vergleich *siehe* Spaceship-Operator
drop() 460
Dubletten entfernen 733
Durchschnitt (Menge) 766
DYLD_LIBRARY_PATH 651
dynamic_cast<>() 336
dynamic_pointer_cast 940
dynamischer Typ 339
dynamisches
  Array 364
  Binden 258, 311, 972
  Datenobjekt 229

E

e, E 49
Editor 29
effizienter binärer Operator 480
egrep 519
Ein- und Ausgabe 104
Einbinden von C-Funktionen 163
Eingabe 415
  Datei- 107, 413
  von Strings 105
  benutzerdefinierter Typen 440
Einschränkung *siehe* constraint
Elementdaten, Zeiger auf 264
Elementfunktion 170
  als Funktionsobjekt 836
  Zeiger auf 263
Ellipse
  in catch-Klausel 347
  template parameter pack 498
else 66
#else, #elif 134

emplace() 847, 858, 860, 861
emplace_back() 850, 854, 857
emplace_front() 854, 857
empty() 844, 858, 859, 861, 925
  Namespace std 845
enable_if 512
end 438
end()
  Container 455, 844
  Namespace std 222, 845, 888
  string 273, 924
  vector 237
#endif 136
endl 55, 430
  oder '\n'? 107
ends 430
ends_with() 930
ENTER 31, 105
»enthält«-Beziehung 332
enum 85
Environment, env[] 239
EOF 416
eof() 347, 416, 417, 436, 450
eofbit 435
epsilon() 796
equal() 775
equal_range() 763, 864
equalsIgnoreCase() 710
equal_to 833
erase() 848, 863, 928
ereignisgesteuerte Programmierung 570
errno 344
error_code 804
Exception 970
  arithmetische Fehler 352
  und Destruktor 345
  Handling 345
  Hierarchie 349
  Speicherleck durch Exc. 684
exception 349
<exception> 351
Exception-Sicherheit 355
exclusive_scan() 728
ExecutionPolicy 800
exists() (Filesystem) 807
exit() 127, 947
  und Destruktor 205
Exklusiv-Oder (Menge) 768
exp() 777, 945
explicit 186
explizite Instanziierung von Templates 296
Exponent 49–51
export 165
extension() (Filesystem) 806
extent 511
extern 145–147
extern "C" 164
extern template 295
external linkage 150
extract() 864

F

f, F 49
Fünf (die großen Fünf) 483
fabs() 945
Facette 910
fail() 436, 437
failbit 435, 442
fakultaet() 114
[[fallthrough]] 74
Fallunterscheidung 72
false 58
false_type 506
falsename() 913
Fehlerbehandlung 343
  Ein- und Ausgabe 434
Fibonacci 496, 729
__FILE__ 141
filename() 806
Filesystem
  canonical() 806
  copy() 808
  create_directory() 809
  current_path() 806
  exists() 807
  extension() 806
  filename() 806
  is_directory() 807
  recursive_directory_iterator 812
  remove() 806
  remove_all() 807

stem() 806
fill() 424, 722
fill_n() 722
Filter 460
filter() 460
final 322, 682
find()
Algorithmus 754
assoziative Container 863
string 929
find_end() 758
find_first_of() 755
find_...-Methoden (string) 929
fixed 425, 426, 428
Fixture 665
flache Kopie 252
flags() 425
flip() 851
Bitset 829
float 49
floatfield 426
floating_point 518
floor() 945
flush() 107
flush (Manipulator) 430
fmod() 945
fmtflags 425
Fold-Expression 501
for 79
Kurzform 98
foreach(), for_each_n() 786
Formalparameter 116
Formatierung 419
forward() 822
Forward-Iterator 882
<forward_list> 842
forwarding reference 479
frac_digits() 915
Fragmentierung (Speicher) 234
Framework 571
free store 230
free() 397
frexp() 945
friend 277
from_chars() 705
front() 849, 852, 854, 857, 859, 925
front_inserter(), -_insert_iterator 891
fstream 414, 438
<fstream> 107
Füllzeichen 420, 424
__func__ 141
function 835
<functional> 351, 832
Funktion 114
frei oder global 149
mit Gedächtnis (static) 117
mit initializer_list 846
klassenspezifische 284
mathematische 52, **945**
Parameterübergabe
per Referenz 121
per Wert 119
per Zeiger 235
vorgegebene Parameterwerte 124
rein virtuelle 315
mit Definition 316
static 284
alternative Syntax 281
Überschreiben 310
virtuelle *siehe* virtuelle Funktionen
Funktionsobjekte 386, 431
function 835
mem_fn 836
Funktions-Template 156
Funktor *siehe* Funktionsobjekte
future 562
<future> 351

G

Ganzzahlen 41
garbage collection 234
gcd() 799
gegenseitige Abhängigkeit von Klassen 210
Genauigkeit 50
Generalisierung 298
generate(), generate_n() 723
generische Programmierung 448
GET (http) 605
get() 104, 256, 415, 416
getenv() 947

getline() für Strings 106, 931
getline(char*,...) 416
getloc() 915
get_money() (Manipulator) 428
get_monthname() 917
get_time() (Manipulator) 428
get_time(), get_weekday(), get_year() (locale) 917
ggT 76
  std::gcd() 799
  schnell 198
Gleichheitsoperator bei Vererbung 410
Gleichverteilung 782
Gleitkommazahl 53
  Syntax 49
global 62
  Funktion 149
  Namensraum 162
  Variable 145, 146
glvalue 469
gmtime() 950
GNU Autotools 653
good() 436
goodbit 435
goto 972
Grafische Benutzungsschnittstelle 569
greater, greater_equal 833
greedy (regex-Auswertung) 522
Grenzwerte von Zahltypen 795
Groß- und Kleinschreibung 31, 41
größter gemeinsamer Teiler *siehe* ggT
grouping() 913, 914
guard (Threads) 545
Gültigkeitsbereich 127
  Block 62
  Datei 150
  Funktion 116
  Klassen 172
  und new 233

H

Hängender Zeiger 233
hardware_concurrency() 533
has_facet() 904
hash() 911
hasher 874
Hash-Funktion 872, 874
has_infinity 796
__has_include 136
has_sort_function 507
has_value() (optional) 839
»hat«-Beziehung 699
Header 33, 162
  Datei 142
    Inhalt 145
  Http 605
  der Standardbibliothek 816
Heap 768
hex 425, 428
Hexadezimalzahl 43
hexfloat 425
Host Byte Order 602

I

-I Compileroption 134
iconv 909
IDE 30, 36
Identität von Objekten 169, **972**
identity 721
IEC 60559, IEEE 754 50
if 66
if constexpr 71, 500
#if, #ifdef, #ifndef 134
ifstream 107, 414
ignore() 416
imag() 776, 777
imbue() 709, 902
Implementation 142
  -svererbung 340
  -sdatei, Inhalt 145
implizite Deklaration
  Destruktor 203
  Konstruktor 176
  Zuweisungsoperator 369, 409
import 165
in 437
#include 32, **134**
Include-Guard 135
includes() 764
inclusive_scan() 727
Indexoperator 90, **218**, 222, 246, **366**
index_sequence 830

infinity() 796
Initialisierung
  array 852
  direkte I. der Attribute 177
  und virtuelle Basisklassen 333
  C-Array 220, 243
  einfacher Datentypen 44
  mit Element-Initialisierungsliste 179
  Konstante in Objekten 180, 285
  globaler Konstanten 145, 147
    Reihenfolge 290
  mit {}-Liste 187, 846, 888
  mit konstruktor-interner Liste 285
  von Objekten 176
  von Referenzen 961
  Reihenfolge der Initialisierung
    von Attributen 179
    von Funktionsargumenten 133
  in for-Schleife 79
  von static-Elementdaten 284
  struct 88
  und Vererbung 302
  und Zuweisung 45, 182
initializer_list 188, 366, 797, 846
  für zweidimensionales Array 846
Inklusionsmodell 295
Inkrementierung 46
Inkrementoperator 374
inline
  Elementfunktion 175
  Funktion 151
  Konstante 147, 288
  Variable 151, 288
inner_product() 726
innere Klasse 456, 623
inplace_merge() 753
Input-Iterator 882
insert() 848, 863, 871, 927
inserter() 892
insert_iterator 892
insert_or_assign 867, 876
Instanz 168, 972
Instanziierung von Templates 293
  explizite 296
  ökonomische (bei vielen Dateien) 295
int 33, 41
int-Parameter in Templates 293
intX_t, int_fastX_t, int_leastX_t (X = 8, 16, 32, 64), intmax_t 48
integer_sequence 830
integral 518
integral_constant 506
integral promotion 61
Interface (UML) 694
internal 425, 428
internal linkage 150
Internet-Anbindung 591
Intervall (und Notation) 972
invalid_argument 351
<iomanip> 428, 430
ios 414, 434, 436
  failure 436
  Flags zur Dateipositionierung 438
  Methoden 425, 436
<ios> 428
ios_base 414
  binary 109
  Fehlerstatusbits 435
  Flags 425, 426
  Manipulatoren 428
iostate 434
<iostream> 33, 414, 415, 418, 428
iota, iota_view 723
IPv4, IPv6 593
IPv4-Adresse (regulärer Ausdruck) 780
is() 911
is_abstract 511
isalnum(), isalpha() 910, 944
is_arithmetic 509, 510, 515
is_array 510
is_base_of 511
isblank() 944
is_bounded 796
is_class 506, 510
iscntrl() 910, 944
is_const 511
is_convertible 511
isdigit() 132, 184, 910, 944
is_directory() (Filesystem) 807
is_enum 510

is_exact 796
is_final 511
is_floating_point 510
is_function 510
is_fundamental 511
isgraph() 910, 944
is_heap(), is_heap_until() 772
is_iec559, is_integer 796
is_integral 510
islower() 910, 944
is_lvalue_reference 510
is_modulo 796
is_null_pointer 510
ISO 10646 906
ISO 8859-1, ISO 8859-15 905
is_partitioned() 745
is_permutation() 740
is_pointer 510
is_polymorphic 511
isprint() 910, 944
ispunct() 910
is_reference 510
is_rvalue_reference 510
is_same 511
is_signed 511, 796
is_sorted(), is_sorted_until() 747
isspace() 910, 944
ist-ein-Beziehung 298, 307, 325
istream 414, 415
Istream-Iterator 892
istream::seekg(), tellg() 438
istream::ws 430
istringstream 414, 439
is_unsigned 511
isupper() 910, 944
is_void 510
isxdigit() 910, 944
iter_swap() 790
Iterator 273, 449, 453, 881
  Adapter 889
  Bidirectional 883
  Contiguous 883
  Forward 882
  Input 882
  Insert 890
  Output 882
  Random Access 883
  Reverse 889
  Stream 892
  Tag 883
  Zustand 454
iterator 843
iterator_category 882
<iterator> 881

J

Jahr 949
join() 536, 537
jthread (Klasse) 538

K

Kardinalität 697
Kategorie (locale) 910
key_equal 874
key_type 863, 870
key_comp(), key_compare 864
Klammerregeln 59
Klasse 168, 171, 973
  abgeleitete *siehe* abgeleitete Klasse
  abstrakte 315
  Basis- *siehe* Basisklasse
  Deklaration 173
  innere 456, 623
  konkrete 315
  Ober- *siehe* Oberklasse
  für einen Ort 172
  Unter- *siehe* Unterklasse
  für rationale Zahlen 194
Klassenname (typeid) 339
klassenspezifische
  Daten und Funktionen 284
  Konstante 288
Klassen-Template 291
Klassifikation 298, 973
Kleinschreibung 31
kleinstes gemeinsames Vielfaches *siehe* lcm()
Kollisionsbehandlung 872
Kommandointerpreter 635
Kommandozeilenparameter 239
Kommaoperator 60, 82, 503

Kommentar 31
komplexe Zahlen 776
Komplexität 978
Komposition 700
konkrete Klasse 315
Konsole 30
  auf UTF-8 einstellen 905
Konstante 53
  globale 145, 147
  klassenspezifische 288
konstante Objekte 174
Konstruktor 173, **176**
  allgemeiner 178
  implizite Deklaration 176
  delegierender 189
  erben 308
  Kopier- *siehe* Kopierkonstruktor
  vorgegebene Parameterwerte 178
  Typumwandlungs- *siehe* Typumwandlungskonstruktor
Kontrollstrukturen 64
Konvertieren von Datentypen *siehe* Typumwandlung
Kopie, flache/tiefe 252
Kopieren
  von Dateien 109
  von Objekten 252
  von Zeichenketten 228
Kopierkonstruktor **181**, 367, 483
  Auslassung durch Compiler 183
Kreuzreferenzliste 718
kritischer Bereich 544
Kurzform-Operatoren 45

L

l, L 49
Länge eines Vektors 726
Lambda-Funktionen 485
`LANG` 902
late binding 258
Laufvariable 79, 80
Laufzeit 214
  und Funktionszeiger 258
  und `new` 230
  und Polymorphie 311
  Typinformation 338
`lcm()` 799
`LD_LIBRARY_PATH` 651
ldd 651
`ldexp()` 945
`ldiv()`, `ldiv_t` 946
`left` 425, 428
`length()` (C-String, `constexpr`) 224, 279, 513
`length()` (`string`) 925
`length_error` 351
`less`, `less_equal` 833
`lexicographical_compare()` 742
`lexicographical_compare_three_way()` 743
lexikografischer Vergleich 390, **973**
`<limits>` 42, 49, 795
`__LINE__` 141
Linken 146, 162
  dynamisches 650, 973
  internes, externes 150
  statisches 648, 973
Linker 35
linksassoziativ 59, 242
`list` 854
`<list>` 819, 842
Liste
  Initialisierungs- 179, 285
  Initialisierungs- (bei C-Arrays) 243
`Liste` (Klasse) 455
Literal 224, 974
  benutzerdefiniert 401
  String 400
  Zahl- 53
  Zeichen- 54, 906
`load_factor()` 875
`local_iterator` 874, 875
`<locale>` 901
`localtime()` 376, 950
`log()`, `log10()` 776, 945
`logic_error` 350, 351
`logical_and`, `-_not`, `-_or` 833
logischer Datentyp 57
logische Negation 58
lokal (Block) 62
lokale Objekte 217
`long` 41

long double 49
lower_bound() 762, 865
lvalue 469
L-Wert 470

M

magic number 974
main() 31, 32, **127**
MAKE 646
make 633
  automatische Ermittlung von Abhängigkeiten 639
  parallelisieren 647
  rekursiv 646
  Variable 636
Makefile 142, **634**
make_from_tuple() 826
make_heap() 771
make_index_sequence 830
make_integer_sequence 830
make_pair() 824
make_shared 685, 939
make_tupel() 825
make_unique 268, 685, 937
Makro 137
malloc() 397
Manipulatoren 427
Mantisse 50
<map> 842, 865
mapped_type 867
match_results 526
mathematischer Ausdruck 52
mathematische Funktionen 945
Matrix
  C-Array 242
  C-Array, dynamisch 246
  Klasse 249
max() 796
max(initializer_list<T>) 797
max_bucket_count() 875
max_element() 730
max_exponent, max_exponent10 796
max_load_factor() 875
max_size() 844
[[maybe_unused]] 957
mehrdimensionales Array 242
Mehrfachvererbung 300, 328, 331
MeinString (Klasse) 271
mem_fn() 836
member function *siehe* Elementfunktion
memcpy() 366
memory leak 233, 684
<memory> 351, 819
Mengenoperationen auf sortierten Strukturen 764
merge() 752, 855, 864
mergesort() 753
messages 910, 919
Metaprogrammierung 493
Methode 168, 170, 974
  Regeln zur Konstruktion von Prototypen 675
midpoint() 799
MIME 974
min() 796
min(initializer_list<T>) 797
min_element() 730
min_exponent, min_exponent10 796
MinGW-Konsole 35
minmax(...) 797
minmax_element() 730
minus 832
Minute 949
mischen 752
mismatch() 773
mkdir 641
mktime() 950
modf() 945
Modulare Programmgestaltung 141
Module 164
Modulo 46
modulus 832
Monat 949
monetary 910, 914
money_get 915
moneypunct 914
money_put 916
Monitor-Konzept 554
move semantics 472
move() 476

move(), move_backward()
Container-Bereich 822
move_constructible 518
moving constructor 475
mt19937 733, 781
multimap 869
multiplies 832
Multiplikationsoperator 372
Multiplizität (UML) 697
multiset 871
mutable
Attribut 174
Lambda-Funktion 490
mutex 544

N

'\n' 33
oder endl? 107
und regex_replace 528
Nachbedingung 133, 974
Nachkommastellen 49
precision 427
Name 41
einer Klasse (typeid) 339
name() 339, 904
Namenskonflikte bei Mehrfachvererbung 331
Namenskonventionen 41
Namespace 63, 147
anonym 150
in Header-Dateien 150
Verzeichnisstruktur 643
namespace 32, 147
namespace std 63
NaN (not a number) 389
narrow() 912
nationale Sprachumgebung 902
NDEBUG 140
negate 832
Negation
bitweise 46, 47
logische 58
negative_sign() 915
neg_format() 915
Network Byte Order 602
Netzwerkprogrammierung 591
neue Zeile 56, 66
new **229**, 232, 287
Fehlerbehandlung 354
Placement-Form 941
überladen 393
<new> **941**
new_handler 354
<new> 351
next() 887
next_permutation() 739
noboolalpha 428
Nodehandle 864
[[nodiscard]] 957
noexcept 348
none() (Bitset) 830
none_of 738
norm() 776, 777
Normalverteilung 784
noshowbase, -point, -pos 428
noskipws 428
not_equal_to 833
nothrow 355
notify_one(), -_all() 550, 691
nounitbuf, nouppercase 428
npos **924**
nth_element() 750
NTP – Network Time Protocol 602
NULL 216, 946
nullopt 839
nullptr 216
Null-Zeiger und new 355
<numbers> 191
numeric 910, 912
numeric_limits 49, 795
numerische Auslöschung 51
numerische Umwandlung 703, 706, 931
num_get, num_put 912
NummeriertesObjekt (Klasse) 284
numpunct 913

O

O-Notation 978
Oberklasse **298**, 310, 975
Subobjekt einer 302
Subtyp einer 306
Zugriffsrechte vererben 304

Oberklassenkonstruktor 299, 302
object slicing 307, 315, 409
Objekt 28, **169**, 173, 975
-code 34
dynamisches 229
als Funktions- 386
-hierarchie 332
Identität *siehe* Identität von Objekten
Initialisierung 176
konstantes 174
-orientierung 167
Übergabe per Wert 182
verkettete Objekte 232
verwitwetes 233
vollständiges 333, 334, 978
Objektcode 35
`oct` 425, 428
ODER
bitweises 46
logisches 58
Öffnungsarten für Streams 437
`offsetof` 946
`ofstream` 107, 414
Oktalzahl 43
`omanip` 430
one definition rule 144
`open()` 107
Open Source 975
Operator
arithmetischer 46
binärer 359
Bit- 46
für `char` 57
als Funktion 358
Kurzform 47
für Literale 399
für logische Datentypen 58
Präzedenz 59, **955**
relationale 46, 58
Syntax 358
Typumwandlungs- 378
unärer 359
für ganze Zahlen 45
`operator delete()` 393
`operator new()` 393
`operator string()` 379
`operator()()` 386
`operator*()` 373, 380, 453
`operator*=()` 372
`operator++()` 374, 377, 453
`operator++(int)` 680
`operator+=()` 361
`operator->()` 380
`operator<=>()` 388
`operator<<()` 361, 417
`operator=()` 370
`operator==()` 453, 830
bei Vererbung 410
`operator>>()` 415
`operator[]()` 406, 849, 852, 857, 883
`operator!=()` 453
Optimierung durch Vermeiden temporärer Objekte 183
`optional` 839
`Ort` (Klasse) 172
`ostream` 361, 414, 417
`ostream::endl`, `ends`, `flush()` 430
Ostream-Iterator 892
`ostream::seekp()`, `tellp()` 438
`ostringstream` 414, 439
osyncstream 552
`out` 437
`out_of_range` 351
Output-Iterator 882
overflow 44, 51
`overflow_error` 351
`override` 313, 681

P

`packaged_task` 562
`pair` 823
Parameter
-expansion 499
einer Funktion 115
Pack 498
-übergabe
per Referenz 121
per Wert 119
per Zeiger 235
parametrisierte Datentypen 156, 291
»part-of«-Beziehung 699

partial_ordering 389
partial_sort(), partial_sort_copy 749
partial_sum() 727
partielle Spezialisierung von Templates 882
partition() 744
partition_copy(), partition_point() 744
path 803
patsubst 639
peek() 417
perfect forwarding 822
Performance 465
Permutationen 739
Pfeiloperator 217
PHONY 636
π, pi, pi_v 191
Pipe (ranges) 460
Placement new/delete 941
plus 832
pointer 843
Pointer, smarte 380
polar() 777
polymorpher Typ 319, 339
Polymorphismus 311, 975
pop() 858, 860, 861
pop_back() 850, 854, 857
pop_front() 854, 857
pop_heap() 770
portabel (Zeichensatz) 908
pos_format() 915
Positionierung innerhalb einer Datei 438
positive_sign() 915
POSIX 902
POST (http) 610
postcondition *siehe* Nachbedingung
Postfix-Operator 374
pos_type 438
pow() 777
Prädikat
  Algorithmus mit P. 896
  binäres 734, 774, 896
  unäres 737
Präfix-Operator 374
Präprozessor 133, 639
Präzedenz von Operatoren 59, **955**
precision() 426
precondition *siehe* Vorbedingung
prev() 887
prev_permutation() 739
PRINT (Makro) 139
printf() 497
Priority-Queue 860
private 172, 303
private Vererbung 340
Programm
  ausführbares 35
  Strukturierung 113
Programmierrichtlinien 112, 684
Projection (Ranges) 721
proj() 777
Projekt 37, 142
promise 562
protected 303
protected-Vererbung 342
Prototyp
  Funktions- 114
  einer Methode 172
  Regeln zur Konstruktion 675
prvalue 470
ptrdiff_t 845, 946
public 172, **303**
Pufferung (Ein-/Ausgabe) 104, 418
push() 858, 859, 861
push_back() 850, 854, 857
  vector 94
push_front() 854, 857
push_heap() 771
put() 107, 256, 418
putback() 416
put_money(), -_time() (Manipulator) 428

Q

qsort() 259, 947
Qt 571
QThread 587
Quantifizierer 522
Quellcode 34
Queue 859
<queue> 819, 842, 859
quoted() 428

R

race condition 544, 563
radix 796
RAII 545, 665, 685, **976**
<random> 733
random access 454
Random-Access-Iterator 883
random_access_iterator 518
random_device 781
range based for 98
range_error 351
Ranges (Bereiche), <ranges> 459
rank 511
<ratio> 837
Rationale Zahl
  Klasse 194
  Template std::ratio 837
rbegin()
  Container 845, 889
  Namespace std 845, 888
  string 925
rdstate() 436
read() 255
real() 776, 777
Rechengenauigkeit 50, 81
rechtsassoziativ 59, 242
recursive_directory_iterator 812
reelle Zahlen 49
ref(), reference_wrapper 541, 838
reference
  bitset 828
  Container 843
  vector<bool> 850
reference collapsing rules 478
reference counting 938
Referenz 121, 131
  auf Basisklasse 314
  auf const 122, 181
  auf istream 415, 440
  auf Oberklasse 306, 314
  auf ostream 362, 417
  Parameterübergabe per 121
  Rückgabe per 367
  auf R-Wert 472
  -semantik 253, 465
  weiterleitende 479
Regel der 0/3/5 *siehe* Rule of 0/3/5
regex_iterator 526
regex_match() 527
regex_replace() 528, 713
regex_search() 528, 712
<regex> 351, 527
regex_iterator 703
reguläre Ausdrücke 519
Reihenfolge
  Auswertungs- 51, 60
  der Initialisierung
    von Attributen 179
    von Funktionsargumenten 133
  umdrehen 735
rein virtuelle Funktion 315
  mit Definition 316
reinterpret_cast<>() 221, 255, 338
Rekursion 120
  Template-Metaprogrammierung 494, 499
rekursiver Abstieg 128
rekursiver Make-Aufruf 646
relationale Operatoren 46, 58
remove()
  Algorithmus 794
  Datei/Verzeichnis 806
  Liste 855
remove_all() Dateien/Verzeichnisse 807
remove_const 511
remove_if() 794, 855
remove_reference 511, 822
rename() Datei/Verzeichnis 810
rend()
  Container 845, 889
  Namespace std 845, 888
  string 925
replace() 792, 928
replace_copy(),-_if(), -_copy_if() 792
requires 515
reserve() 850, 926
reset() (Bitset) 829
resetiosflags() 428
resize() 850, 855, 857, 926
resume() 564
return 33, 183
Return Value Optimization (RVO) 183

reverse() (list) 855
reverse(), reverse_copy() 735
reverse (Ranges) 460
reverse_iterator 845
Reverse-Iterator 889
Reversible Container 845
right 425, 428
»rohes« Stringliteral 400
rotate(), rotate_copy() 731
round_error(), round_style() 796
RTTI 338
Rückgabetyp auto 155, 176, 313
Rule of zero/three/five 483
runtime_error 351
rvalue 470
R-Wert, Referenz auf R-Wert 470

S

safe_iterator_t 722
safe_subrange_t 722
same_as 517
sample() 736
scan_is(), scan_not() 911
Schleifen 75
  und Container 97
  do while 77
  for 79
  und Strings 226
  Tabellensuche 92
  -terminierung 77
  while 75
Schlüsselwörter 953
Schnittmenge 766
Schnittstelle 142, 976
  einer Funktion 118
  Regeln zur Konstruktion 675
scientific 425, 426, 428
scope 62
scoped locking 545
scoped_lock 545
search() 756, 758
search_n() 760
seekg(), seekp() 438
Seiteneffekt 68, 115, 133, 227, 229
  im Makro 140
Seitenvorschub 56
Sekunde 949
Selektion 66
sentinel 92, 221
Sequenz
  -konstruktor 187, 188, 366
  -methoden (Container) 848
Server-Client
  Beziehung 173
  und callback 260
set() (Bitset) 829
<set> 842, 869
setbase() 428
set_difference() 767
setf() 425, 426
setfill() 428
set_intersection() 766
setiosflags() 428
setprecision() 428
setstate() 436
set_symmetric_difference() 768
set_terminate() 352
set_union() 765
setw() 428
SFINAE 507, 976
shared_ptr 385, 684, **938**
  für Arrays 685
short 41
showbase, showpoint, showpos 425, 428
showContainer() 508
shrink_to_fit() 850, 857, 926
shuffle() 732, 733
Sichtbarkeit 62
  -sbereich (namespace) 147
sign() 945
Signal 572, 575
Signalton 56
Signatur 125, 300, 311, 314, **976**
signed char 54
signed_integral 518
sin(), sinh() 776, 945
single entry/ single exit 83
size()
  Container 844
  Namespace std 90, 219, 845
  string 94
  vector 90

size_t 48, 946
size_type 843
sizeof 220
  nicht bei dynamischen Arrays 247
sizeof... (variadische Templates) 499
Skalarprodukt 726
skipws 425, 428
sleep_for(), sleep_until() 532, 537
Slot 572, 575
Smart Pointer 380
  und Exceptions 684
Socket 595
Sommerzeit 949
Sonderzeichen 56
sort_heap() 772
sortable 517
Sortieren
  mit qsort() 260
  mit sort() 747
  stabiles 747
  durch Verschmelzen 753
source_location 140
Spaceship-Operator 388, 844
span 237, 879
 842
Speicher
  -klasse 146
  -leck 233, 684
  -platzfreigabe 217
Spezialisierung
  von Klassen 298
  von Templates 158
splice() 855
split() 702
Sprachumgebung 902
SQL 621
sqrt() 52, 777, 945
sregex_iterator 703
ssize() 98, 845
<sstream> 440
stable_partition() 744
stable_sort() 747
Stack 62
  Klasse 291
<stack> 819, 842, 858
stack unwinding 345
Standard
  -bibliothek
    C 162, 943
    C++ 816
  -header 163, 819
  -klassen 819
  Typumwandlung 60, 262
    Zeiger 262
Standard Ein-/Ausgabe 104
starts_with() 930
static 146
  Attribute und Methoden 284
  in Funktion 117
  Initialisierungsreihenfolge 290
static_assert 140
static_cast<>() 56, 335
-static und -Bstatic (Makefile) 649
statisches Binden 311, **976**
Statusabfrage einer Datei 436
std 32, 63, 149
<stdexcept> 351
stdio 425
Stelligkeit (Template) 497
stem() (Filesystem) 806
Stichprobe 736
STL 447
stod() 704
stoi() 703
  und verwandte numerische Konversionsfunktionen 932
stop_requested(),stop_token() 538
strcat(), strchr(), strcmp() 948
strncat(), strncmp() 949
strpbrk(), strrchr(), strstr() 948
strcpy() 229, 948
strcspn() 948
Stream
  Öffnungsarten 437
  Iterator 892
stream 413
streamsize 415
strerror() 948
Streuspeicherung 872
strftime() 950
String 94
  in Zahl umwandeln 703

Klasse MeinString 271
Länge 226, 227
Literal 400, 906
string 94, 923
append() 926
assign() 927
at() 94, 925
back() 925
begin() 924
capacity() 926
cbegin(), cend(), crbegin(), crend() 925
clear() 928
compare() 930
copy() 925
c_str() 925
data() 925
empty() 925
end() 924
erase() 928
find() 929
find_...-Methoden 929
front() 925
insert() 927
length() 94, 925
max_size() 925
operator+=() 926
rbegin(), rend() 925
replace() 928
reserve() 926
resize() 926
shrink_to_fit() 926
size() 94, 925
substr() 930
swap() 925
<string> 819
stringstream 705
string_view 280, 933
<string_view> 819
strlen() 224, 227, 948
strncpy() 229, 949
strong_ordering 389
strtod(), strtol(), strtoul() 947
strtok() 948
struct 87, 304
Strukturierte Bindung 101
Stunde 949
Subobjekt **299**, 306, 330
in virtuellen Basisklassen 332
verschiedene 331
subrange 462
Substitutionsprinzip 325
substr() 930
Subtyp 300, 306, 325, 977
Suffix 49
suspend_always 566
swap() 370, 844, 925
Algorithmus 790
swap-Trick 254
swap_ranges() 790
switch 72
symmetrische Differenz (Menge) 768
Synchronisation 543
Syntax 54
Syntaxdiagramm
?: Bedingungsoperator 71
do while-Schleife 77
enum-Deklaration 85
for-Schleife 79
Funktions-
aufruf 116
definition 115
prototyp 115
Template 156
if-Anweisung 67
mathematischer Ausdruck 129
operator-Deklaration 358
struct-Definition 87
switch-Anweisung 72
Typumwandlungsoperator 379
while-Schleife 75
system() 947
system_clock 532
system_error 351, 436
Szenario 206

T

Tabellensuche 92
Tabulator 56
Tag (des Monats) 949
Tag Dispatching (und Alternative) 884
tan(), tanh() 776, 945

target (make) 634
Taschenrechnersimulation 128
Tastaturabfrage 105
TCP 592
»Teil-Ganzes«-Beziehung 699
tellg(), tellp() 438
Template
  Alias 266
  für Funktionen 156
  Instanziierung von T. 293
    explizite 296
    ökonomische (bei vielen Dateien) 295
  int-Parameter 293
  für Klassen 291
  -Konstante 191
  -Metaprogrammierung 494
  Method (Design-Muster) 324, 410
  Spezialisierung 158
    partielle 882
  variable Parameterzahl 497
temporäres Objekt (Vermeidung) 183
Terminal 30
terminate() 351
terminate_handler 352
test() (Bitset) 830
Test Driven Development 657
Test-Suite 658
Textersetzung 137
this 240
this->, *this bei Zugriff auf Oberklassenelement 373
this_thread 536
thousands_sep() 913, 914
Thread 531
thread (Klasse) 534
ThreadGroup 542
Thread-Sicherheit 563
throw 346
tiefe Kopie 252
time 910, 917
time() 376, 950
time_get 917
time_put 918
time_t 376, 949
tm 376, 949
to_array() 853
tolower() 708, 910, 912, 944
top() 858, 861
to_string()
  bitset 830
  Zahlkonvertierung 932
to_ulong() (Bitset) 829
toupper() 708, 907, 910, 912, 944
trailing return type 281
traits 881
traits::eof() 416
transform() 791, 911
transform() (Ranges) 460
transform_exclusive_scan() 728
transform_inclusive_scan() 728
tree (Programm) 642, 812
Trennung von Schnittstellen und Implementation 142
true 58
truename() 913
true_type 506
trunc 437
try 346
try_emplace() 876
Tupel, <tuple> 825
Typ 977
  polymorpher bzw. dynamischer Typ 339
type cast *siehe* Typumwandlung
typedef 265
typeid() 338, 412
type_info 338
<typeinfo> 351
typename (bei Template-Parametern) 156
Type Traits 505
Typinformation 320
  zur Laufzeit 338
Typumwandlung
  cast 56, 216, 260, 335
  durch Compiler 196
  const_cast<>() 337
  dynamic_cast<>() 336
  mit explicit 186
  implizite 61, 186
  mit Informationsverlust 126
  -skonstruktor 184, 196

-soperator 378
ios 436
reinterpret_cast<>() 338
Standard- 60
Zeiger 262
static_cast<>() 335

U

u8 906
u8string, u16string, u32string 400
UCS 906
UDP 592, 600
Überladen
von Funktionen 125
von Operatoren *siehe* operator
Überlauf 44, 51
Überschreiben
von Funktionen 310
Übersetzung 142
-seinheit 144
uintX_t, uint_fastX_t, uint_leastX_t (X = 8, 16, 32, 64), uintmax_t 48
Umgebungsvariable 239
UML 693
Umleitung der Ausgabe auf Strings 439
unärer Operator 359
unäres Prädikat 737
UND
bitweises 46, 47
logisches 58
#undef 136
undefined behaviour 234
underflow 51
underflow_error 351
unsigned_integral 518
Unicode 413, 905
uniform_int_distribution 782
uniform_real_distribution 783
union 102
unique() Sequenzen 855
unique(), -_copy() Algorithmen 733
unique_lock 545
unique_ptr 267, 385, 684, 935
für Arrays 687
Unit-Test 655
unitbuf 418, 425, 428
unordered_map 875
unordered_multimap 878
unordered_multiset 878
unordered_set 878
unsigned 42
unsigned char 54
Unterklasse 298, 977
upper_bound() 762, 865
uppercase 425, 428
URI, URL 592, 713
URL-Codierung 605
use case 205
use_facet() 708, 710, 904
using
enum 87
Deklaration 305, 341
Namespace
Deklaration 148
Direktive 148
statt typedef 265
UTC 950
UTF 400, 977
UTF-8 905
<utility> 284, 476, 790, 822, 825

V

valarray 249
valgrind 399
value(), value_or() (optional) 839
value_comp(), value_compare 865
value_type 867
value_type 843, 863
Variable 34
automatische 146
globale 145, 146
make 636
Name 41
variadic templates 497
variant 831
vector 849
at() 91
push_back() 94
size() 90
vector<bool> 850
<vector> 819, 842, 849
Vektor 89

Klasse 363
Länge (geom.) 726
Verbundanweisung 66
verdecken (Methode) 300, 305
Vereinigung (Menge) 765
Vererbung **297**, 977
der abstrakt-Eigenschaft 316
von constraints 326
der Implementierung 341
Mehrfach- 328
private 340
protected 342
von Zugriffsrechten 304
und Zuweisungsoperator 409
Vergleich
von `double`-Werten 678
bei Vererbung 410
Verschiebung (*move*) 472
verschmelzen (*merge*) 752
Vertrag 326, 978
verwitwetes Objekt 233
Verzeichnis
anlegen 809
anzeigen 811
kopieren 808
löschen 806
umbenennen 810
Verzeichnisbaum
anzeigen 812
make 643
Verzweigung 66
View 460
`views` 460
`virtual` 311, 313, 321
virtuelle Basisklasse 332
virtueller Destruktor 320
virtuelle Funktionen **311**, 314
private 324
rein- 315
`void` 216
als Funktionstyp 115
`void*` 260
Typumwandlung nach 216
`volatile` 978
vollständiges Objekt 333, 334, 978
Vorbedingung 133, 978
vorgegebene Parameterwerte
in Funktionen 124
in Konstruktoren 178
Vorkommastellen 49
Vorrangregeln 59, **955**
Vorwärtsdeklaration 210

W

Wächter (Tabellenende) 92, 221
Wahrheitswert 58
zufällig erzeugen 785
`wait()` 549
mit Lambda-Funktion 554
Warteschlange 859
Wartung 111
`wchar_t` 55, 906, 946
`weak_ordering` 390
`weak_ptr` 940
Webserver 611
Weite der Ausgabe 421, 424
weiterleitende Referenz 479
Wert
eines Attributs 970
Parameterübergabe per 119
Wertebereich ganzer Zahlen 43
Wertsemantik 253, 449, 465
Performanceproblem 467
`what()` 350
`while` **75**, 226, 228
white space *siehe* Zwischenraumzeichen
wide character 55
`widen()` 912
Widget 575
`width()` 424
Wiederverwendung
durch Delegation 342
`wildcard` 639
Winterzeit 949
Wochentag 949
`wofstream` 909
Wrapperklasse für Iterator 889
`write()` 255, 418
`ws` 430
`wstring` 400, 909, 923

X

XOR, bitweises 46
xvalue 469

Y

`yield()` 536

Z

Zahl in String umwandeln 706
Zahlenbereich 42, 50
Zeichen 54
Zeichenkette 33, *siehe auch* String
    C-String 223
    Kopieren einer 227
Zeichenklasse (Regex) 522
Zeichenliteral 906
Zeichensatz 904
Zeiger **213**, 230
    Arithmetik 220
    vs. Array 219
    auf Basisklasse 314, 320
    Darstellung
        von `[ ]` 222
        von `[ ][ ]` 246
    auf Elementdaten 264
    auf Elementfunktionen 263
    auf Funktionen 258
    hängender 233
    intelligente *siehe* Smart Pointer
    Null-Zeiger 216
    auf Oberklasse 306, 314
    auf Objekt (Mehrfachvererbung) 331
    auf lokale Objekte 217
    Parameterübergabe per Z. 235
Zeile
    einlesen *siehe* `getline()`
    neue 56
Zeit-Server 603
Zeitkomplexität 978
Ziel (make) 634
Ziffernzeichen 54
Zufallszahlen 781
Zugriffsspezifizierer und -rechte 303
zusammengesetzte Datentypen 84
Zusicherung 139
Zustand 978
    eines Iterators 454
Zuweisung 33, 65, 70, 978
    und Initialisierung 182
    und Vererbung 306, 409
Zuweisungsoperator 182, **369**, 483
    implizite Deklaration 369, 409
    und Vererbung 409
zweidimensionale Matrix 403
Zweierkomplement 42
Zwischenraumzeichen **104**, 224, 415