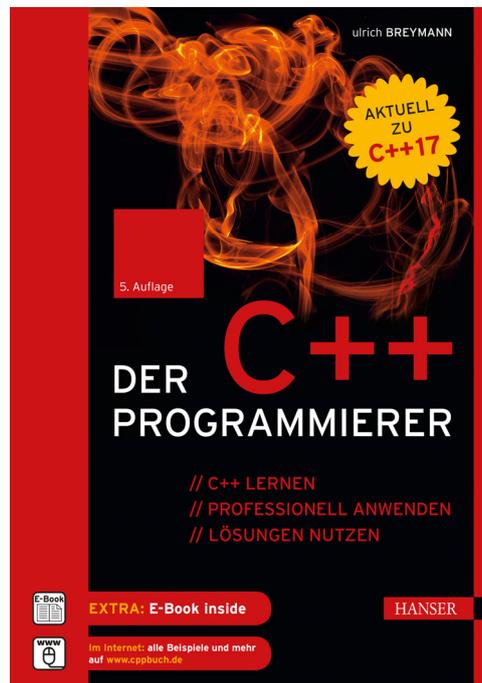


HANSER



Leseprobe

zu

„Der C++Programmierer“ (5. Auflage)

von Ulrich Breymann

ISBN (Buch): 978-3-446-44884-1

ISBN (E-Book): 978-3-446-45386-9

Weitere Informationen und Bestellungen unter
<http://www.hanser-fachbuch.de/9783446448841>

sowie im Buchhandel

© Carl Hanser Verlag München

Inhalt

Vorwort	23
Teil I: Einführung in C++	27
1 Es geht los!	29
1.1 Historisches	29
1.2 Objektorientierte Programmierung	30
1.3 Werkzeuge zum Programmieren	32
1.4 Das erste Programm	33
1.4.1 Namenskonventionen	39
1.5 Integrierte Entwicklungsumgebung	39
1.6 Einfache Datentypen und Operatoren	42
1.6.1 Ausdruck	42
1.6.2 Ganze Zahlen	42
1.6.3 Reelle Zahlen	49
1.6.4 Konstanten	53
1.6.5 Zeichen	54
1.6.6 Logischer Datentyp bool	58
1.6.7 Referenzen	59
1.6.8 Regeln zum Bilden von Ausdrücken	60
1.6.9 Standard-Typumwandlungen	61
1.7 Gültigkeitsbereich und Sichtbarkeit	62
1.7.1 Namespace std.....	64

1.8	Kontrollstrukturen	65
1.8.1	Anweisungen	65
1.8.2	Sequenz (Reihung)	67
1.8.3	Auswahl (Selektion, Verzweigung)	67
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	84
1.9.2	Strukturen.....	87
1.9.3	Der C++-Standardtyp vector	88
1.9.4	Zeichenketten: Der C++-Standardtyp string	93
1.9.5	Container und Schleifen	95
1.9.6	Typermittlung mit auto	97
1.9.7	Deklaration einer strukturierten Bindung mit auto.....	98
1.9.8	Unions und Bitfelder	99
1.10	Einfache Ein- und Ausgabe	101
1.10.1	Standardein- und -ausgabe.....	101
1.10.2	Ein- und Ausgabe mit Dateien	104
2	Programmstrukturierung	109
2.1	Funktionen.....	110
2.1.1	Aufbau und Prototypen	110
2.1.2	Gültigkeitsbereiche und Sichtbarkeit in Funktionen	112
2.1.3	Lokale static-Variable: Funktion mit Gedächtnis	113
2.2	Schnittstellen zum Datentransfer	114
2.2.1	Übergabe per Wert	115
2.2.2	Übergabe per Referenz	119
2.2.3	Gefahren bei der Rückgabe von Referenzen.....	120
2.2.4	Vorgegebene Parameterwerte und unterschiedliche Parameterzahl	121
2.2.5	Überladen von Funktionen	122
2.2.6	Funktion main()	123
2.2.7	Beispiel Taschenrechnersimulation	124
2.2.8	Spezifikation von Funktionen.....	129
2.2.9	Alternative Funktions-Syntax.....	129
2.3	Modulare Programmgestaltung	129
2.3.1	Steuerung der Übersetzung nur mit #include	130

2.3.2	Einbinden vorübersetzter Programmteile	131
2.3.3	Übersetzungseinheit, Deklaration, Definition	132
2.3.4	Dateiübergreifende Gültigkeit und Sichtbarkeit.....	134
2.3.5	Präprozessordirektiven und Makros	136
2.4	Namensräume	144
2.5	inline-Funktionen und -Variable	146
2.5.1	inline-Variablen	147
2.6	constexpr-Funktionen.....	147
2.7	Rückgabetypp auto.....	150
2.8	Funktions-Templates	151
2.8.1	Spezialisierung von Templates	153
2.8.2	Einbinden von Templates	154
2.9	C++-Header.....	156
2.9.1	Einbinden von C-Funktionen	158
3	Objektorientierung 1	159
3.1	Abstrakter Datentyp	160
3.2	Klassen und Objekte	161
3.2.1	const-Objekte und Methoden.....	164
3.2.2	inline-Elementfunktionen.....	165
3.3	Initialisierung und Konstruktoren	166
3.3.1	Standardkonstruktor.....	167
3.3.2	Direkte Initialisierung der Attribute	168
3.3.3	Allgemeine Konstruktoren	168
3.3.4	Kopierkonstruktor.....	171
3.3.5	Typumwandlungskonstruktor	174
3.3.6	Konstruktor und mehr vorgeben oder verbieten.....	176
3.3.7	Einheitliche Initialisierung und Sequenzkonstruktor.....	176
3.3.8	Delegierender Konstruktor	179
3.3.9	constexpr-Konstruktor und -Methoden	180
3.4	Beispiel: Rationale Zahlen	183
3.4.1	Aufgabenstellung	183
3.4.2	Entwurf.....	184
3.4.3	Implementation.....	187
3.5	Destruktoren	192
3.6	Wie kommt man zu Klassen und Objekten? Ein Beispiel.....	194
3.7	Gegenseitige Abhängigkeit von Klassen	199

4	Intermezzo: Zeiger	201
4.1	Zeiger und Adressen	202
4.2	C-Arrays.....	205
4.2.1	C-Array, std::size() und sizeof.....	207
4.2.2	Initialisierung von C-Arrays.....	208
4.2.3	Zeigerarithmetik.....	208
4.2.4	Indexoperator bei C-Arrays.....	209
4.2.5	C-Array mit begin() und end() durchlaufen	209
4.3	C-Zeichenketten.....	210
4.3.1	Schleifen und C-Strings.....	213
4.4	Dynamische Datenobjekte	217
4.4.1	Freigeben dynamischer Objekte.....	220
4.5	Zeiger und Funktionen.....	222
4.5.1	Parameterübergabe mit Zeigern.....	222
4.5.2	Array als Funktionsparameter.....	224
4.5.3	const und Zeiger-Parameter	225
4.5.4	Parameter des main-Programms	225
4.5.5	Gefahren bei der Rückgabe von Zeigern.....	226
4.6	this-Zeiger	227
4.7	Mehrdimensionale C-Arrays.....	229
4.7.1	Statische mehrdimensionale C-Arrays	229
4.7.2	Mehrdimensionales Array als Funktionsparameter.....	230
4.7.3	Dynamisch erzeugte mehrdimensionale Arrays.....	233
4.7.4	Klasse für dynamisches zweidimensionales Array	235
4.8	Binäre Ein-/Ausgabe	241
4.9	Zeiger auf Funktionen.....	244
4.10	Typumwandlungen für Zeiger.....	248
4.11	Zeiger auf Elementfunktionen und -daten	249
4.11.1	Zeiger auf Elementfunktionen.....	249
4.11.2	Zeiger auf Elementdaten	250
4.12	Komplexe Deklarationen lesen	250
4.12.1	Lesbarkeit mit typedef und using verbessern	251
4.13	Alternative zu rohen Zeigern, new und delete.....	253
5	Objektorientierung 2	255
5.1	Eine String-Klasse	255
5.1.1	friend-Funktionen	261

5.2	String-Ansicht.....	262
5.3	Klassenspezifische Daten und Funktionen	265
5.3.1	Klassenspezifische Konstante.....	269
5.4	Klassen-Templates	271
5.4.1	Ein Stack-Template	271
5.4.2	Stack mit statisch festgelegter Größe.....	274
5.5	Typbestimmung mit decltype und declval.....	275
6	Vererbung	279
6.1	Vererbung und Initialisierung.....	285
6.2	Zugriffsschutz	286
6.3	Typbeziehung zwischen Ober- und Unterklasse	288
6.4	Code-Wiederverwendung.....	289
6.4.1	Konstruktor erben.....	290
6.5	Überschreiben von Funktionen in abgeleiteten Klassen	292
6.5.1	Virtuelle Funktionen.....	294
6.5.2	Abstrakte Klassen	299
6.5.3	Virtueller Destruktor.....	304
6.5.4	Private virtuelle Funktionen.....	307
6.6	Probleme der Modellierung mit Vererbung.....	309
6.7	Mehrfachvererbung.....	312
6.7.1	Namenskonflikte	314
6.7.2	Virtuelle Basisklassen	315
6.8	Standard-Typumwandlungsoperatoren	319
6.9	Typinformationen zur Laufzeit.....	322
6.10	Using-Deklaration für protected-Funktionen	323
6.11	Private- und Protected-Vererbung.....	324
7	Fehlerbehandlung	329
7.1	Ausnahmebehandlung	331
7.1.1	Exception-Spezifikation in Deklarationen.....	334
7.1.2	Exception-Hierarchie in C++	335
7.1.3	Besondere Fehlerbehandlungsfunktionen.....	337
7.1.4	Erkennen logischer Fehler	338
7.1.5	Arithmetische Fehler / Division durch 0	340
7.2	Speicherbeschaffung mit new	341
7.3	Exception-Sicherheit	342

8	Überladen von Operatoren	345
8.1	Rationale Zahlen – noch einmal	347
8.1.1	Arithmetische Operatoren.....	347
8.1.2	Ausgabeoperator <<	349
8.2	Eine Klasse für Vektoren	351
8.2.1	Index-Operator [].....	354
8.2.2	Zuweisungsoperator =.....	356
8.2.3	Mathematische Vektoren	359
8.2.4	Multiplikationsoperator	360
8.3	Inkrement-Operator ++	362
8.4	Typumwandlungsoperator	366
8.5	Smart Pointer: Operatoren -> und *	367
8.5.1	Smart Pointer und die C++-Standardbibliothek	372
8.6	Objekt als Funktion	373
8.7	new und delete überladen	375
8.7.1	Unterscheidung zwischen Heap- und Stack-Objekten	378
8.7.2	Fehlende delete-Anweisung entdecken	380
8.7.3	Eigene Speicherverwaltung für einen bestimmten Typ	381
8.7.4	Empfehlungen im Umgang mit new und delete	386
8.8	Operatoren für Literale	386
8.8.1	Stringliterale	387
8.8.2	Benutzerdefinierte Literale	388
8.9	Mehrdimensionale Matrizen	390
8.9.1	Zweidimensionale Matrix als Vektor von Vektoren	391
8.9.2	Dreidimensionale Matrix.....	394
8.10	Zuweisung und Vergleich bei Vererbung	396
9	Dateien und Ströme.....	405
9.1	Ausgabe	407
9.1.1	Formatierung der Ausgabe	407
9.2	Eingabe.....	410
9.3	Manipulatoren.....	413
9.3.1	Eigene Manipulatoren	418
9.4	Fehlerbehandlung.....	420
9.5	Typumwandlung von Dateiobjekten nach bool.....	421
9.6	Arbeit mit Dateien	422
9.6.1	Positionierung in Dateien	423

9.6.2	Lesen und Schreiben in derselben Datei	424
9.7	Umleitung auf Strings	425
9.8	Tabelle formatiert ausgeben	427
9.9	Formatierte Daten lesen	428
9.9.1	Eingabe benutzerdefinierter Typen	428
9.10	Blockweise lesen und schreiben	430
9.10.1	vector-Objekt binär lesen und schreiben	430
9.10.2	array-Objekt binär lesen und schreiben	431
9.10.3	Matrix binär lesen und schreiben	432
9.11	Ergänzungen	434
10	Die Standard Template Library (STL)	435
10.1	Container, Iteratoren, Algorithmen	436
10.2	Iteratoren im Detail	441
10.3	Beispiel verkettete Liste	442
Teil II: Fortgeschrittene Themen		447
11	Performance, Wert- und Referenzsemantik	449
11.1	Performanceproblem Wertsemantik	451
11.1.1	Auslassen der Kopie	451
11.1.2	Temporäre Objekte bei der Zuweisung	452
11.2	Referenzsemantik für R-Werte	453
11.3	Optimierung durch Referenzsemantik für R-Werte	455
11.3.1	Bewegender Konstruktor	458
11.3.2	Bewegender Zuweisungsoperator	458
11.4	Die move()-Funktion	459
11.4.1	Regel zur Template-Auswertung von &t&t-Parametern	461
11.5	Ein effizienter binärer Plusoperator	462
11.5.1	Return Value Optimization (RVO)	463
11.5.2	Kopien temporärer Objekte eliminieren	463
11.5.3	Verbesserung durch verzögerte Auswertung	464
11.5.4	Weitere Optimierungsmöglichkeiten	466
11.6	Rule of three/five/zero	467
11.6.1	Rule of three	467
11.6.2	Rule of five	467
11.6.3	Rule of zero	468

12	Lambda-Funktionen	469
12.1	Eigenschaften	470
12.1.1	Äquivalenz zum Funktionszeiger	471
12.1.2	Lambda-Funktion und Klasse	472
12.2	Generische Lambda-Funktionen	473
12.3	Parametererfassung mit []	475
13	Template-Metaprogrammierung	477
13.1	Grundlagen	478
13.2	Variadic Templates: Templates mit variabler Parameterzahl	480
13.2.1	Ablauf der Auswertung durch den Compiler	481
13.2.2	Anzahl der Parameter	482
13.2.3	Parameterexpansion	482
13.3	Fold-Expressions	484
13.3.1	Weitere Varianten	485
13.3.2	Fold-Expression mit Kommaoperator	486
13.4	Klassen-Template mit variabler Stelligkeit	488
14	Reguläre Ausdrücke	489
14.1	Elemente regulärer Ausdrücke	490
14.1.1	Greedy oder lazy?	492
14.2	Interaktive Auswertung	494
14.3	Auszug des regex-API	497
14.4	Verarbeitung von \n	498
14.5	Anwendungen	500
15	Threads	501
15.1	Zeit und Dauer	502
15.2	Threads	503
15.3	Die Klasse thread	507
15.3.1	Thread-Group	509
15.4	Synchronisation kritischer Abschnitte	510
15.5	Thread-Steuerung: Pausieren, Fortsetzen, Beenden	513
15.5.1	Data Race	518
15.6	Warten auf Ereignisse	518
15.7	Reader/Writer-Problem	524
15.7.1	Wenn Threads verhungern	528
15.7.2	Reader/Writer-Varianten	529

15.8	Atomare Veränderung von Variablen	529
15.9	Asynchrone verteilte Bearbeitung einer Aufgabe	532
15.10	Thread-Sicherheit	534
16	Grafische Benutzungsschnittstellen	535
16.1	Ereignisgesteuerte Programmierung	536
16.2	GUI-Programmierung mit Qt	537
16.2.1	Installation und Einsatz	537
16.2.2	Meta-Objektsystem	538
16.2.3	Der Programmablauf	539
16.2.4	Ereignis abfragen	540
16.3	Signale, Slots und Widgets	541
16.4	Dialog	550
16.5	Qt oder Standard-C++?	553
16.5.1	Threads	554
16.5.2	Verzeichnisbaum durchwandern	555
17	Internet-Anbindung	557
17.1	Protokolle	558
17.2	Adressen	558
17.3	Socket	562
17.3.1	Bidirektionale Kommunikation	565
17.3.2	UDP-Sockets	567
17.3.3	Atomuhr mit UDP abfragen	569
17.4	HTTP	571
17.4.1	Verbindung mit GET	573
17.4.2	Verbindung mit POST	577
17.5	Mini-Webserver	578
18	Datenbankanbindung	587
18.1	C++-Interface	588
18.2	Anwendungsbeispiel	592
Teil III: Ausgewählte Methoden und Werkzeuge der Softwareentwicklung		599
19	Effiziente Programmerzeugung mit make	601
19.1	Wirkungsweise	603

19.2	Variablen und Muster	605
19.3	Universelles Makefile für einfache Projekte	606
19.4	Automatische Ermittlung von Abhängigkeiten	608
19.4.1	Getrennte Verzeichnisse: src, obj, bin	609
19.5	Makefile für Verzeichnisbäume	611
19.5.1	Rekursive Make-Aufrufe	612
19.5.2	Ein Makefile für alles	614
19.6	Automatische Erzeugung von Makefiles.....	615
19.6.1	Makefile für rekursive Aufrufe erzeugen	616
19.7	Erzeugen von Bibliotheken	617
19.7.1	Statische Bibliotheksmodule	618
19.7.2	Dynamische Bibliotheksmodule.....	619
19.8	Code Bloat bei der Instanziierung von Templates vermeiden.....	622
19.8.1	extern-Template.....	623
19.9	CMake	625
19.10	GNU Autotools	626
20	Unit-Test.....	627
20.1	Werkzeuge	628
20.2	Test Driven Development	629
20.3	Boost Unit Test Framework.....	630
20.3.1	Beispiel: Testgetriebene Entwicklung einer Operatorfunktion	632
20.3.2	Fixture.....	636
20.3.3	Testprotokoll und Log-Level.....	637
20.3.4	Prüf-Makros	638
20.3.5	Kommandozeilen-Optionen.....	642
Teil IV: Das C++-Rezeptbuch:		
Tipps und Lösungen für typische Aufgaben		643
21	Sichere Programmentwicklung.....	645
21.1	Regeln zum Design von Methoden	645
21.2	Defensive Programmierung.....	647
21.2.1	double- und float-Werte richtig vergleichen	648
21.2.2	const und constexpr verwenden	649
21.2.3	Anweisungen nach for/if/while einklammern	649
21.2.4	int und unsigned/size_t nicht mischen	649

21.2.5	size_t oder auto statt unsigned int verwenden.....	650
21.2.6	Postfix++ mit Präfix++ implementieren	650
21.2.7	Ein Destruktor darf keine Exception werfen	651
21.2.8	explicit-Typumwandlungsoperator bevorzugen	651
21.2.9	explicit-Konstruktor für eine Typumwandlung bevorzugen	651
21.2.10	Leere Standardkonstruktoren vermeiden	651
21.2.11	Mit override Schreibfehler reduzieren.....	651
21.2.12	Kopieren und Zuweisung verbieten	651
21.2.13	Vererbung verbieten	652
21.2.14	Überschreiben einer virtuellen Methode verhindern	653
21.2.15	»Rule of zero« beachten	653
21.2.16	One Definition Rule.....	653
21.2.17	Defensiv Objekte löschen	653
21.2.18	Speicherbeschaffung und -freigabe kapseln.....	654
21.2.19	Programmierrichtlinien einhalten	654
21.3	Exception-sichere Beschaffung von Ressourcen.....	654
21.3.1	Sichere Verwendung von unique_ptr und shared_ptr.....	654
21.3.2	So vermeiden Sie new und delete!.....	655
21.3.3	shared_ptr für Arrays korrekt verwenden	656
21.3.4	unique_ptr für Arrays korrekt verwenden	657
21.3.5	Exception-sichere Funktion	658
21.3.6	Exception-sicherer Konstruktor	658
21.3.7	Exception-sichere Zuweisung	659
21.4	Empfehlungen zur Thread-Programmierung.....	660
21.4.1	Warten auf die Freigabe von Ressourcen	660
21.4.2	Deadlock-Vermeidung.....	661
21.4.3	notify_all oder notify_one?.....	662
21.4.4	Performance mit Threads verbessern?.....	662
22	Von der UML nach C++	663
22.1	Vererbung	664
22.2	Interface anbieten und nutzen	664
22.3	Assoziation	666
22.3.1	Aggregation.....	669
22.3.2	Komposition	670
23	Algorithmen für verschiedene Aufgaben.....	671
23.1	Algorithmen mit Strings	672

23.1.1	String splitten	672
23.1.2	String in Zahl umwandeln	673
23.1.3	Zahl in String umwandeln	675
23.1.4	Strings sprachlich richtig sortieren	676
23.1.5	Umwandlung in Klein- bzw. Großschreibung	678
23.1.6	Strings sprachlich richtig vergleichen	680
23.1.7	Von der Groß-/Kleinschreibung unabhängiger Zeichenvergleich	681
23.1.8	Von der Groß-/Kleinschreibung unabhängige Suche	682
23.2	Textverarbeitung	683
23.2.1	Datei durchsuchen	683
23.2.2	Ersetzungen in einer Datei	685
23.2.3	Lines of Code (LOC) ermitteln	687
23.2.4	Zeilen, Wörter und Zeichen einer Datei zählen	688
23.2.5	CSV-Datei lesen	688
23.2.6	Kreuzreferenzliste	690
23.3	Operationen auf Folgen	692
23.3.1	Container anzeigen	693
23.3.2	Folge mit gleichen Werten initialisieren	693
23.3.3	Folge mit Werten eines Generators initialisieren	694
23.3.4	Folge mit fortlaufenden Werten initialisieren	694
23.3.5	Summe und Produkt	695
23.3.6	Mittelwert und Standardabweichung	696
23.3.7	Skalarprodukt	696
23.3.8	Folge der Teilsummen oder -produkte	697
23.3.9	Folge der Differenzen	698
23.3.10	Kleinstes und größtes Element finden	699
23.3.11	Elemente rotieren	701
23.3.12	Elemente verwürfeln	702
23.3.13	Dubletten entfernen	702
23.3.14	Reihenfolge umdrehen	704
23.3.15	Stichprobe	705
23.3.16	Anzahl der Elemente, die einer Bedingung genügen	706
23.3.17	Gilt ein Prädikat für alle, keins oder wenigstens ein Element einer Folge?	707
23.3.18	Permutationen	708
23.3.19	Lexikografischer Vergleich	711
23.4	Sortieren und Verwandtes	712
23.4.1	Partitionieren	712

23.4.2	Sortieren.....	714
23.4.3	Stabiles Sortieren	715
23.4.4	Partielles Sortieren	716
23.4.5	Das n.-größte oder n.-kleinste Element finden	717
23.4.6	Verschmelzen (merge)	718
23.5	Suchen und Finden	721
23.5.1	Element finden	721
23.5.2	Element einer Menge in der Folge finden	722
23.5.3	Teilfolge finden.....	723
23.5.4	Teilfolge mit speziellem Algorithmus finden	725
23.5.5	Bestimmte benachbarte Elemente finden	726
23.5.6	Bestimmte aufeinanderfolgende Werte finden	727
23.5.7	Binäre Suche.....	728
23.6	Mengenoperationen auf sortierten Strukturen	731
23.6.1	Teilmengenrelation	731
23.6.2	Vereinigung.....	732
23.6.3	Schnittmenge	733
23.6.4	Differenz	733
23.6.5	Symmetrische Differenz.....	734
23.7	Heap-Algorithmen	735
23.7.1	pop_heap	736
23.7.2	push_heap.....	737
23.7.3	make_heap	737
23.7.4	sort_heap	738
23.7.5	is_heap	738
23.8	Vergleich von Containern auch ungleichen Typs.....	739
23.8.1	Unterschiedliche Elemente finden	739
23.8.2	Prüfung auf gleiche Inhalte	741
23.9	Rechnen mit komplexen Zahlen: Der C++-Standardtyp complex.....	742
23.10	Schnelle zweidimensionale Matrix	744
23.10.1	Optimierung mathematischer Array-Operationen	747
23.11	Vermischtes	750
23.11.1	Erkennung eines Datums.....	750
23.11.2	Erkennung einer IPv4-Adresse	752
23.11.3	Erzeugen von Zufallszahlen	753
23.11.4	for_each – Auf jedem Element eine Funktion ausführen	758
23.11.5	Verschiedene Möglichkeiten, Container-Bereiche zu kopieren.....	758

23.11.6	Vertauschen von Elementen, Bereichen und Containern	761
23.11.7	Elemente transformieren	761
23.11.8	Ersetzen und Varianten	763
23.11.9	Elemente herausfiltern	764
23.11.10	Grenzwerte von Zahltypen	766
23.11.11	Minimum und Maximum	767
23.11.12	Wert begrenzen.....	768
23.11.13	ggT und kgV.....	769
23.12	Parallelisierbare Algorithmen	770
24	Datei- und Verzeichnisoperationen.....	771
24.1	Übersicht	772
24.2	Pfadoperationen.....	773
24.3	Datei oder Verzeichnis löschen.....	774
24.3.1	Möglicherweise gefülltes Verzeichnis löschen	775
24.4	Datei oder Verzeichnis kopieren	776
24.5	Datei oder Verzeichnis umbenennen	777
24.6	Verzeichnis anlegen	777
24.7	Verzeichnis anzeigen	778
24.8	Verzeichnisbaum anzeigen	779
Teil V:	Die C++-Standardbibliothek	781
25	Aufbau und Übersicht	783
25.1	Auslassungen	785
25.2	Beispiele des Buchs und die C++-Standardbibliothek	787
26	Hilfsfunktionen und -klassen	789
26.1	Relationale Operatoren	789
26.2	Unterstützung der Referenzsemantik für R-Werte.....	790
26.2.1	move()	790
26.2.2	forward()	791
26.3	Paare.....	792
26.4	Tupel.....	794
26.5	bitset	796
26.6	Indexfolgen	799
26.7	variant statt union	800
26.8	Funktionsobjekte.....	801

26.8.1	Arithmetische, vergleichende und logische Operationen	801
26.8.2	Binden von Argumentwerten.....	802
26.8.3	Funktionen in Objekte umwandeln.....	804
26.9	Templates für rationale Zahlen.....	806
26.10	Hüllklasse für Referenzen.....	808
26.11	Optionale Objekte	808
26.12	Type Traits	810
26.12.1	Wie funktionieren Type Traits? – ein Beispiel.....	811
26.12.2	Abfrage von Eigenschaften	813
26.12.3	Abfrage numerischer Eigenschaften	815
26.12.4	Typbeziehungen.....	815
26.12.5	Typumwandlungen	816
26.13	Auswahl weiterer Traits	816
26.13.1	decay.....	816
26.13.2	enable_if	816
26.13.3	conditional	817
26.13.4	default_order	817
27	Container	819
27.1	Gemeinsame Eigenschaften.....	821
27.1.1	Initialisierungslisten	823
27.1.2	Konstruktion an Ort und Stelle.....	823
27.1.3	Reversible Container.....	824
27.2	Sequenzen	825
27.2.1	vector	826
27.2.2	vector<bool>	827
27.2.3	list.....	829
27.2.4	deque	831
27.2.5	stack	833
27.2.6	queue	834
27.2.7	priority_queue.....	836
27.2.8	array	838
27.3	Assoziative Container	840
27.4	Sortierte assoziative Container.....	842
27.4.1	map und multimap	843
27.4.2	set und multiset	847
27.5	Hash-Container.....	849

27.5.1	unordered_map und unordered_multimap	853
27.5.2	unordered_set und unordered_multiset	855
28	Iteratoren	857
28.1	Iterator-Kategorien	858
28.1.1	Anwendung von Traits	859
28.2	Abstand und Bewegen	862
28.3	Zugriff auf Anfang und Ende	863
28.3.1	Reverse-Iteratoren	864
28.4	Insert-Iteratoren	865
28.5	Stream-Iteratoren	866
29	Algorithmen	869
29.1	Algorithmen mit Prädikat	870
29.2	Übersicht	871
30	Nationale Besonderheiten	875
30.1	Sprachumgebung festlegen und ändern	876
30.1.1	Die locale-Funktionen	877
30.2	Zeichensätze und -codierung	879
30.3	Zeichenklassifizierung und -umwandlung	883
30.4	Kategorien	884
30.4.1	collate	884
30.4.2	ctype	885
30.4.3	numeric	887
30.4.4	monetary	888
30.4.5	time	891
30.4.6	messages	893
30.5	Konstruktion eigener Facetten	895
31	String	897
31.1	string_view für String-Literale	907
32	Speichermanagement	909
32.1	unique_ptr	909
32.1.1	make_unique	911
32.2	shared_ptr	912
32.2.1	make_shared	913
32.2.2	Typumwandlung in einen Oberklassentyp	913

32.3	weak_ptr	914
32.4	new mit Speicherortangabe	915
33	Numerische Arrays (valarray)	917
33.1	Konstruktoren	918
33.2	Elementfunktionen	918
33.3	Binäre Valarray-Operatoren	921
33.4	Mathematische Funktionen.....	923
33.5	slice und slice_array	924
33.6	gslice und gslice_array	927
33.7	mask_array	930
33.8	indirect_array	931
34	Ausgewählte C-Header	933
34.1	<cassert>	933
34.2	<cctype>	934
34.3	<cmath>.....	934
34.4	<cstddef>.....	936
34.5	<cstdlib>	936
34.6	<ctime>	937
34.7	<cstring>	939
A	Anhang.....	941
A.1	ASCII-Tabelle	941
A.2	C++-Schlüsselwörter	943
A.3	Compilerbefehle	944
A.4	Rangfolge der Operatoren	945
A.5	C++-Attribute für den Compiler	947
A.6	Lösungen zu den Übungsaufgaben	947
A.7	Installation der Software für Windows	957
A.7.1	Installation des Compilers und der Entwicklungsumgebung.....	957
A.7.2	Integrierte Entwicklungsumgebung einrichten	958
A.7.3	De-Installation.....	958
A.8	Installation der Software für Linux	959
A.8.1	Installation des Compilers	959
A.8.2	Installation von Boost	960
A.8.3	Installation und Einrichtung von Code::Blocks	960
A.8.4	Beispieldateien entpacken	961

A.9	Installationshinweise für OS X.....	961
A.9.1	Installation von Boost.....	962
A.9.2	Beispieldateien entpacken	962
Glossar	963
Literaturverzeichnis	973
Register	977

Vorwort

Die 5. Auflage unterscheidet sich von der 4. Auflage durch die Umstellung auf den 2017 von der zuständigen ISO/IEC-Arbeitsgruppe JTC1/SC22/WG21 verabschiedeten C++-Standard. Die Veränderungen gegenüber dem C++-Standard von 2014 sind durch Markierungen am Seitenrand (wie hier) gekennzeichnet. Das Buch ist konform zum neuen C++-Standard, ohne den Anspruch auf Vollständigkeit zu erheben – das Standard-Dokument [ISO C++] umfasst allein mehr als 1600 Seiten. Sie finden in diesem Buch eine verständliche und mit vielen Beispielen angereicherte Einführung in die Sprache. Im Teil »Das C++-Rezeptbuch« gibt es zahlreiche Tipps und Lösungen für typische Aufgaben, die in der täglichen Praxis anfallen. Es gibt konkrete, sofort umsetzbare Lösungsvorschläge. Zahlreiche Algorithmen für praxisnahe Problemstellungen helfen bei der täglichen Arbeit. Auf größtmögliche Portabilität wird geachtet: Die Beispiele funktionieren unter Linux genauso wie unter Windows und OS X. Die problembezogene Orientierung lässt die in die Sprache einführenden Teile kürzer werden. Damit wird das Lernen erleichtert und die Qualität des Buchs, auch als Nachschlagewerk dienen zu können, bleibt erhalten.

C++17

Für wen ist dieses Buch geschrieben?

Dieses Buch ist für alle geschrieben, die einen kompakten und gleichzeitig gründlichen Einstieg in die Konzepte und Programmierung mit C++ suchen. Es ist für Anfänger¹ gedacht, die noch keine Programmiererfahrung haben, aber auch für andere, die diese Programmiersprache kennenlernen möchten. Beiden Gruppen und auch C++-Erfahrenen dient das Buch als ausführliches Nachschlagewerk.

Die ersten zehn Kapitel führen in die Sprache ein. Es wird sehr schnell ein Verständnis des objektorientierten Ansatzes entwickelt. Die sofortige praktische Umsetzung des Gelernten steht im Vordergrund. C++ wird als Programmiersprache unabhängig von speziellen Produkten beschrieben. C-Kenntnisse werden nicht vorausgesetzt. Das Buch eignet sich zum Selbststudium und als Begleitbuch zu einer Vorlesung oder zu Kursen. Die vielen Beispiele sind leicht nachzuvollziehen und praxisnah umsetzbar. Klassen und Objekte,

¹ Geschlechtsbezogene Formen meinen hier und im Folgenden stets Männer, Frauen und andere.

Templates und Exceptions sind Ihnen bald keine Fremdworte mehr. Es gibt mehr als 90 Übungsaufgaben – mit Musterlösungen im Anhang. Durch das Studium dieser Kapitel werden aus Anfängern bald Fortgeschrittene.

Diesen und anderen Fortgeschrittenen und Profis bietet das Buch kurze Einführungen in die Themen Thread-Programmierung, Netzwerk-Programmierung mit Sockets einschließlich eines kleinen Webservers, Datenbank-Anbindung, grafische Benutzeroberflächen und mehr. Dabei wird durch Einsatz der Boost-Library und des Qt-Frameworks größtmögliche Portabilität erreicht.

Softwareentwicklung ist nicht nur Programmierung: Einführend werden anhand von Beispielen unter anderem die Automatisierung der Programmierzeugung mit Make, die Dokumentationserstellung mit Doxygen und die Versionskontrolle behandelt. Das Programmdesign wird durch konkrete Umsetzungen von UML²-Mustern nach C++ unterstützt. Das integrierte »C++-Rezeptbuch« mit mehr als 150 praktischen Lösungen, das sehr 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

Beim Programmieren geht es nicht nur darum, eine Programmiersprache zu lernen. Es geht auch darum, Programme zu schreiben, die hohen Qualitätsansprüchen gerecht werden. Dazu gehört das Know-how, die Mittel der Sprache 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. In diesem Sinn liegt ein Schwerpunkt des Buchs auf der Methodik des Programmierens, unterstützt durch Darstellung der informatischen Grundlagen, viele Hinweise im Text und demonstriert an vielen Beispielen.

■ Übersicht

Schwerpunkt von Teil I ist die Einführung in die Programmiersprache. Die anschließenden Teile gehen darüber hinaus und konzentrieren sich auf die verschiedenen Probleme der täglichen Praxis. Die in dieser Übersicht verwendeten Begriffe mögen Ihnen zum Teil noch unbekannt sein – in den betreffenden Kapiteln werden sie ausführlich erläutert.

Teil I – Einführung in C++

Kapitel 1 vermittelt zunächst die Grundlagen, wie ein Programm geschrieben und zum Laufen gebracht wird. Es folgen einfache Datentypen und Anweisungen zur Kontrolle des Programmablaufs. Die Einführung der C++-Datentypen `vector` und `string` und einfache Ein- und Ausgaben beenden das Kapitel.

Kapitel 2 zeigt Ihnen, wie Sie Funktionen schreiben. Makros, Templates für Funktionen und die modulare Gestaltung von Programmen folgen.

Objektorientierung ist der Schwerpunkt von *Kapitel 3*. Dabei geht es nicht nur um die Konstruktion von Objekten, sondern auch um den Weg von der Problemstellung zu Klassen und Objekten. Zeiger sowie die Erzeugung von Objekten zur Laufzeit sind Inhalt von

² UML ist eine grafische Beschreibungssprache für die objektorientierte Programmierung.

Kapitel 4. Kapitel 5 führt das Thema Objektorientierung fort. Dabei erfahren Sie, wie eine String-Klasse funktioniert, und wie Sie Klassen-Templates konstruieren. *Kapitel 6* zeigt Ihnen das Mittel objektorientierter Sprachen, um Generalisierungs- und Spezialisierungsbeziehungen auszudrücken: die Vererbung mit ihren Möglichkeiten. Strategien zur Fehlerbehandlung mit Exceptions finden Sie in *Kapitel 7. Kapitel 8* zeigt, wie Sie Operatorsymbolen wie + und - eigene Bedeutungen zuweisen können und in welchem Zusammenhang das sinnvoll ist. Sie lernen, »intelligente« Zeiger (Smart Pointer) zu konstruieren und Objekte als Funktionen einzusetzen. *Kapitel 9* beschreibt ausführlich die Ein- und Ausgabemöglichkeiten, die vorher nur einführend gestreift wurden, einschließlich der Fehlerbehandlung und der Formatierung der Ausgabe. Eine Einführung in die Standard Template Library (STL) bietet *Kapitel 10*. Die STL und ihre Wirkungsweise bilden die Grundlage eines sehr großen Teils der C++-Standardbibliothek.

Teil II – Fortgeschrittene Themen

In diesem Teil folgen fortgeschrittene Themen wie Wert- und Referenzsemantik (*Kapitel 11*), Lambda-Funktionen (*Kapitel 12*), Template-Metaprogrammierung (*Kapitel 13*), reguläre Ausdrücke (*Kapitel 14*) und die Programmierung paralleler Abläufe mit Threads (*Kapitel 15*). *Kapitel 16* zeigt, wie grafische Benutzungsschnittstellen konstruiert werden. Wie ein Programm die Verbindung mit dem Internet aufnehmen kann, dokumentiert *Kapitel 17*. Und wohin mit den ganzen Daten, die bei Programmende nicht verloren gehen sollen? In *Kapitel 18* lernen Sie, wie ein Programm an eine Datenbank angebunden wird. Manche der genannten Themen sind so umfangreich, dass sie selbst Bücher füllen. Für diese Themen wird daher nur ein Einstieg geboten.

Teil III – Ausgewählte Methoden und Werkzeuge der Softwareentwicklung

Die Entwicklung von Programmen besteht nicht nur aus dem Schreiben von Code. Die Compilation eines Projekts mit vielen Programmdateien und Abhängigkeiten kann schnell ein komplexer Vorgang werden. Die Automatisierung dieses Prozesses mit dem Tool *make* ist Thema von *Kapitel 19*. Programme sind nicht auf Anhieb fehlerfrei. Sie müssen getestet werden: *Kapitel 20* stellt ein Werkzeug für den Unit-Test vor und zeigt den praktischen Einsatz.

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

Sichere Programmentwicklung ist die Überschrift von *Kapitel 21*. Sie finden dort Regeln zum Design von Methoden und Tipps zur defensiven Programmierung, die die Wahrscheinlichkeit von Fehlern vermindern. *Kapitel 22* zeigt Rezepte, wie Sie bestimmte UML-Muster in C++-Konstruktionen umwandeln können. Algorithmen für viele verschiedene Aufgaben finden Sie in *Kapitel 23*. Wegen der Vielzahl empfiehlt sich ein Blick in das Inhaltsverzeichnis, um einen Überblick zu gewinnen. *Kapitel 24* enthält fertige Rezepte zum Anlegen, Löschen und Lesen von Verzeichnissen und mehr.

Teil V – Die C++-Standardbibliothek

Hier wird die C++-Standardbibliothek in Kürze beschrieben. Die Inhalte dieses Teils sind: Hilfsfunktionen und -klassen, Container, Iteratoren, Algorithmen, Einstellung nationaler Besonderheiten, String, Speicherverwaltung, Funktionen der Programmiersprache C.

Anhang

Der Anhang enthält unter anderem verschiedene hilfreiche Tabellen und die Lösungen der Übungsaufgaben.

■ 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* ab Seite 963 aufgeführt.

Es gibt ein recht umfangreiches *Stichwortverzeichnis* ab Seite 977 und ein sehr detailliertes *Inhaltsverzeichnis*.



Software zum Buch

Auf der Webseite <http://www.cppbuch.de/> finden Sie die Software zu diesem Buch. Sie enthält unter anderem einen Compiler, eine Integrierte Entwicklungsumgebung sowie alle Programmbeispiele und die Lösungen zu den Aufgaben. Sie finden dort auch weitere Hinweise, Errata und nützliche Links.

■ Zu guter Letzt

Allen Menschen, die dieses Buch durch Hinweise und Anregungen verbessern halfen, sei an dieser Stelle herzlich gedankt. Frau Brigitte Bauer-Schiewek und Frau Irene Weilhart vom Hanser Verlag danke ich für die gute Zusammenarbeit.

Bremen, im Oktober 2017

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

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, sodass in Abständen neue Versionen

des Standards herausgegeben wurden. Die Kurznamen sind entsprechend den Jahreszahlen C++03, C++11, C++14 und C++17. C++17 wurde im Juli 2017 von der zuständigen ISO/IEC-Arbeitsgruppe JTC1/SC22/WG21 verabschiedet und bei der ISO zur Veröffentlichung eingereicht.

1.2 Objektorientierte Programmierung

Nach üblicher Auffassung heißt Programmieren, 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, geschrieben in einer besonderen Sprache, die der Rechner »versteht«. Der Schwerpunkt dieser Betrachtungsweise liegt auf den einzelnen Schritten oder Anweisungen an den Rechner, die zur Lösung einer Aufgabe abzuarbeiten sind.

Was fehlt hier beziehungsweise wird bei dieser Sicht 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. Die *objektorientierte Programmierung* berücksichtigt besonders die Kapselung von Daten und den darauf ausführbaren Funktionen sowie die Wiederverwendbarkeit von Software und die Übertragung von Eigenschaften von Klassen auf andere Klassen, Vererbung genannt. Auf die einzelnen Begriffe wird noch eingegangen. Das Motiv hinter der objektorientierten Programmierung ist die rationelle und vor allem ingenieurmäßige Softwareentwicklung.

Wiederverwendung heißt, Zeit und Geld zu sparen, indem bekannte Klassen wiederverwendet werden. Das Leitprinzip ist hier, das Rad nicht mehrfach neu zu erfinden! Unter anderem durch den Vererbungsmechanismus kann man Eigenschaften von bekannten Objekten ausnutzen. Zum Beispiel sei *Konto* eine bekannte Objektbeschreibung mit den »Eigenschaften« Inhaber, Kontonummer, Betrag, Dispo-Zinssatz und so weiter. In einem Programm für eine Bank kann nun eine Klasse *Waehrungskonto* entworfen werden, für die alle Eigenschaften von *Konto* übernommen (= geerbt) werden könnten. Zusätzlich wäre nur noch die Eigenschaft »Währung« hinzuzufügen.

Wie Computer können Objekte Anweisungen ausführen. Wir müssen ihnen nur »erzählen«, was sie tun sollen, indem wir ihnen eine *Aufforderung* oder *Anweisung* senden, die in einem Programm formuliert wird. Anstelle der Begriffe »Aufforderung« oder »Anweisung« wird manchmal *Botschaft* (englisch *message*) verwendet, was jedoch den Aufforderungscharakter nicht zur Geltung bringt. Eine gängige Notation (= Schreibweise) für solche Aufforderungen ist *Objektname.Anweisung*(gegebenenfalls *Daten*). Beispiele:

```
dieAmpel.blinken(gelb);
dieAmpel.ausschalten(); // keine Daten notwendig!
dieAmpel.einschalten(gruen);
dasRechteck.zeichnen(position, hoehe, breite); // Daten in cm
dasRechteck.verschieben(5.0); // Daten in cm
```

Die Beispiele geben schon einen Hinweis, dass die Objektorientierung uns ein Hilfsmittel zur Modellierung der realen Welt in die Hand gibt.

Klassen

Es wird zwischen der *Beschreibung* von Objekten und den *Objekten selbst* unterschieden. Die Beschreibung besteht aus Attributen und Operationen. Attribute bestehen aus einem Namen und Angaben zum Datenformat der Attributwerte. Eine Kontobeschreibung könnte so aussehen:

Attribute:

Inhaber: Folge von Buchstaben
 Kontonummer: Zahl
 Betrag: Zahl
 Dispo-Zinssatz in %: Zahl

Operationen:

überweisen(Ziel-Kontonummer, Betrag)
 abheben(Betrag)
 einzahlen(Betrag)

Eine Aufforderung ist nichts anderes als der Aufruf einer Operation, die auch Methode genannt wird. Ein *tatsächliches* Konto k1 enthält *konkrete* Daten, also Attributwerte, deren Format mit dem der Beschreibung übereinstimmen muss. Die Tabelle zeigt k1 und ein weiteres Konto k2. Beide Konten haben *dieselben* Attribute, aber *verschiedene* Werte für die Attribute.

Tabelle 1.1: Attribute und Werte zweier Konten

Attribut	Wert für Konto k1	Wert für Konto k2
Inhaber	Roberts, Julia	Depp, Johnny
Kontonummer	12573001	54688490
Betrag	-200,30 €	1222,88 €
Dispo-Zinssatz	13,75 %	13,75 %

Julia will Johnny 1000 € überweisen. Bei einer Überweisung wird die IBAN (internationale Bankkontonummer) angegeben. Dem Objekt k1 wird also der Auftrag mit den benötigten Daten mitgeteilt: *k1.überweisen("DE25123456000054688490", 1000.00)*.

»DE25123456000054688490« ist die (hier fiktive) IBAN. Die letzten 10 Stellen der IBAN enthalten die Kontonummer, die von der empfangenden Bank extrahiert wird, um den Empfänger zu ermitteln. Anderes Beispiel: Johnny will 22 € abheben. Die Aufforderung wird an k2 gesendet: *k2.abheben(22)*.

Es scheint natürlich etwas merkwürdig, wenn einem Konto ein Auftrag gegeben wird. In der objektorientierten Programmierung werden Objekte als Handelnde aufgefasst, die auf Anforderung selbstständig einen Auftrag ausführen, entfernt vergleichbar einem Sachbearbeiter in einer Firma, der seine eigenen Daten verwaltet und mit anderen Sachbearbeitern kommuniziert, um eine Aufgabe zu lösen.

- Die *Beschreibung* eines tatsächlichen Objekts gibt seine innere *Datenstruktur* und die möglichen *Operationen* oder *Methoden* an, die auf die inneren Daten anwendbar sind.
- Zu *einer* Beschreibung kann es kein, ein oder beliebig viele Objekte geben.

Die Beschreibung eines Objekts in der objektorientierten Programmierung heißt *Klasse*. Die tatsächlichen Objekte heißen auch *Instanzen* einer Klasse.

Auf die inneren Daten eines Objekts nur mithilfe der vordefinierten Methoden zuzugreifen, dient der Sicherheit der Daten und ist ein allgemein anerkanntes Prinzip. Das Prinzip wird *Datenabstraktion* oder Geheimnisprinzip genannt. Der Softwareentwickler, der die Methoden konstruiert hat, weiß ja, wie die Daten konsistent (das heißt widerspruchsfrei) bleiben und welche Aktivitäten mit Datenänderungen verbunden sein müssen. Zum Beispiel muss eine Erhöhung des Kontostands mit einer Gutschrift oder einer Einzahlung verbunden sein. Außerdem wird jeder Buchungsvorgang protokolliert. Es darf nicht möglich sein, dass jemand anderes die Methoden umgeht und direkt und ohne Protokoll seinen eigenen Kontostand erhöht. Wenn Sie die Unterlagen eines Kollegen haben möchten, greifen Sie auch nicht einfach in seinen Schreibtisch, sondern Sie bitten ihn darum, dass er sie Ihnen gibt.

Die hier verwendete Definition einer Klasse als Beschreibung der Eigenschaften einer Menge von Objekten wird im Folgenden beibehalten. Gelegentlich findet man in der Literatur andere Definitionen, auf die hier nicht weiter eingegangen wird. Weitere Informationen zur Objektorientierung sind in Kapitel 3 und in der einschlägigen Literatur zu finden.

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 er keine versteckten Sonderzeichen enthalten, weswegen LibreOffice oder Word nicht geeignet sind. Ein für Windows sehr gut geeigneter Texteditor ist *Notepad++*¹. *Kwrite* läuft unter Linux

¹ <https://notepad-plus-plus.org/>

und Mac OS stellt *TextEdit* zur Verfügung, besser geeignet ist jedoch der Editor von *Xcode*, das Sie dazu jedoch installieren müssen. Um für den Anfang die Einarbeitung in eine Integrierte Entwicklungsumgebung zu sparen, können Sie stattdessen einen einfachen Texteditor benutzen und das Programm in der Konsole (=MS-DOS-Eingabeaufforderung oder PowerShell-Fenster, Linux/MacOS-Terminal)² mit den weiter unten gezeigten Kommandos compilieren.

Der Compiler

Compiler sind die Programme, die Ihren Programmtext in eine für den Computer verarbeitbare Form übersetzen. Von Menschen geschriebener und für Menschen lesbarer Programmtext kann nicht vom Computer »verstanden« werden. 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. Falls Sie nicht schon einen C++-Compiler oder ein C++-Entwicklungssystem haben, um die Beispiele korrekt zu übersetzen, bietet sich als Alternative zu einem Editor die Benutzung der in Abschnitt 1.5 beschriebenen Entwicklungsumgebungen an. Ein viel verwendeter Compiler ist der GNU C++-Compiler [GCC]. Entwicklungsumgebung und Compiler sind kostenlos erhältlich. Ein Installationsprogramm für einen unter Windows lauffähigen Compiler finden Sie auf der Website zum Buch [CPP]. Hinweise zur Installation finden Sie in Abschnitt A.7 (Seite 957). Bei den meisten Linux-Systemen ist der GNU C++-Compiler enthalten oder kann nachträglich installiert werden. Auf einem Mac mit OS X sollten Sie die Entwicklungsumgebung Xcode mit dem Clang-Compiler³ installieren (siehe Hinweise dazu ab Seite 961).

1.4 Das erste Programm

Sie lernen hier die Entwicklung eines ganz einfachen Programms kennen. Dabei wird Ihnen zunächst das Programm vorgestellt und wenige Seiten weiter 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 1.1: Das erste Programm!

```
int main() { // Noch tut dieses Programm nichts!  
    // Lies zwei Zahlen ein
```

² Statt des Wortungetüms Eingabeaufforderungsfenster oder des Begriffs Terminal werde ich von nun an in der Regel das Wort *Konsole* verwenden.

³ http://clang.llvm.org/get_started.html

```

    /* Berechne die Summe beider
       Zahlen */
    // Zeige das Ergebnis auf dem Bildschirm an
}

```

Hier sehen Sie schon ein einfaches C++-Programm. Es bedeuten:

```

int           ganze Zahl zur Rückgabe
main         Schlüsselwort für Hauptprogramm
( )          Innerhalb dieser Klammern können dem Hauptprogramm
             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 obigen Programm sind lediglich *Kommentare* enthalten und noch keine Anweisungen an den Computer, so dass unser Programm (noch) 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 den menschlichen Leser eines Programms, um ihm die Anweisungen zu erläutern, zum Beispiel für den Programmierer, der Ihr Nachfolger wird, 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.

Listing 1.2: Summe zweier Zahlen berechnen

```

// cppbuch/k1/summe.cpp
// Hinweis: Alle Programmbeispiele sind von der Internet-Seite zum Buch herunterladbar
// (http://www.cppbuch.de/).

```

```
// Die erste Zeile in den Programmbeispielen gibt den zugehörigen Dateinamen an.
#include<iostream>
using namespace std;

int main() {
    int summe;
    int summand1;
    int summand2;
    // Lies zwei Zahlen ein
    cout << "_Zwei_ganze_Zahlen_eingeben:";
    cin >> summand1 >> summand2;
    /* Berechne die Summe beider Zahlen */
    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, erfahren Sie nur wenige Seiten danach.

`#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 2.3.5.

`using namespace std;` Der Namensraum `std` wird benutzt. Schreiben Sie es einfach in jedes Programm an diese Stelle und haben Sie Geduld: Eine genauere Erklärung folgt später (Seiten 64 und 144).

`int main()` `main()` ist das Hauptprogramm (es gibt auch Unterprogramme). Der zu `main()` gehörende Programmcode wird durch die geschweiften Klammern `{` und `}` eingeschlossen. Ein mit `{` und `}` begrenzter Bereich heißt *Block*. Mit `int` ist gemeint, dass das Programm `main()` nach Beendigung eine Zahl vom Typ `int` (= ganze Zahl) an das Betriebssystem zurückgibt. Dazu dient die unten beschriebene `return`-Anweisung. Normalerweise – das heißt bei ordnungsgemäßigem Programmablauf – wird die Zahl 0 zurückgegeben. Andere Zahlen können über das Betriebssystem einem folgenden Programm einen Fehler signalisieren.

`int summe;` `int summand1;` `int summand2;` *Deklaration* von Objekten: Mitteilung an den Compiler, der entsprechend Speicherplatz bereitstellt und ab jetzt die Namen `summe`, `summand1` und `summand2` innerhalb des Blocks `{ }` kennt. Es gibt verschiedene Zahlentypen in C++. Mit `int` sind ganze Zahlen gemeint: `summe`, `summand1`, `summand2` sind ganze Zahlen.

;	Ein Semikolon beendet jede Deklaration und jede Anweisung (aber keine Verbundanweisung, siehe weiter unten).
cout	Ausgabe: cout (Abkürzung für <i>character out</i> oder <i>console out</i>) ist die Standardausgabe. Der Doppelpfeil deutet an, dass alles, was rechts davon steht, zur Ausgabe cout gesendet wird, zum Beispiel <code>cout << summand1;</code> . 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: <code>cout << "\"C++\"</code> ist der Nachfolger von <code>\"C\"!</code> ; erzeugt die Bildschirmausgabe <code>"C++" ist der Nachfolger von "C"!</code> .
'\n'	die Ausgabe des Zeichens \n bewirkt eine neue Zeile.
return 0;	Unser Programm läuft einwandfrei, es gibt daher 0 zurück. Diese Anweisung darf im <code>main()</code> -Programm 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:

- Objekte 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, z. B. `sume = summand1 + summand2;` im Programm auf Seite 35, kennt der Compiler den falschen Namen `sume` nicht und gibt eine Fehlermeldung aus. Damit 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).

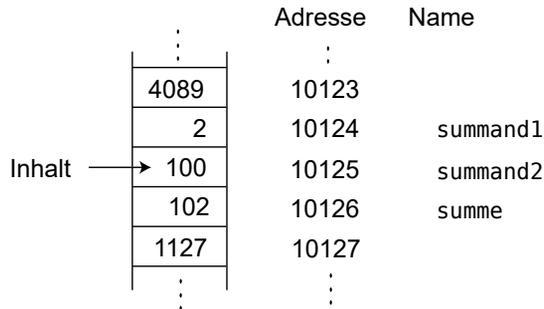


Abbildung 1.1: Speicherbereiche mit Adressen

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. 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« (englisch *source code*) genannt. Der Compiler erzeugt aus dem Quelltext 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 zusammengebunden 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 Rechnerpeicher lädt und startet. Diese Schritte werden stets ausgeführt, auch wenn sie in den Programmumgebungen verborgen ablaufen. Bibliotheksmodule können auch während der Programmausführung geladen werden (nicht im Bild dargestellt). Weitere Details werden in Abschnitt 2.3 erläutert.

Wie bekomme ich ein Programm zum Laufen?

Nachdem Sie den Programmtext mit einem Editor geschrieben haben, kann er übersetzt (compiliert) und ausgeführt werden. Integrierte Entwicklungsumgebungen (englisch *Integrated Development Environment, IDE*) haben einen speziell auf Programmierzwecke zugeschnittenen Editor, der darüber hinaus auf Tastendruck oder Mausklick die Überset-

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 1.7 zeigt die Syntax von while-Schleifen. *AnweisungOderBlock* ist wie auf Seite 67 definiert. Die Bedeutung einer while-Schleife ist: Solange die Bedingung wahr ist, die Auswertung also ein Ergebnis ungleich 0 oder `true` 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 1.8). Die Anweisung oder der Block innerhalb der Schleife heißt *Schleifenkörper*. Schleifen können wie `if`-Anweisungen beliebig geschachtelt werden.

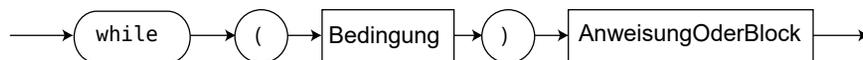


Abbildung 1.7: Syntaxdiagramm einer while-Schleife

```

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

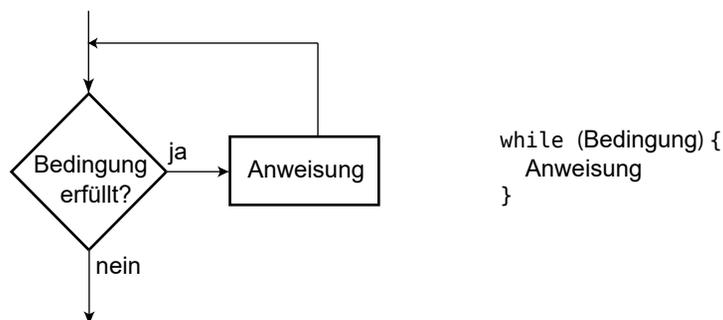


Abbildung 1.8: Flussdiagramm für eine while-Anweisung

Beispiele

■ Unendliche Schleife:

```

while(true)
  Anweisung
  
```

- Anweisung wird nie ausgeführt (unerreichbarer Programmcode):

```
while(false)
    Anweisung
```

- Summation der Zahlen 1 bis 99:

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

- Berechnung des größten gemeinsamen Teilers $\text{ggT}(x, y)$ für zwei natürliche Zahlen x und y nach Euklid. Es gilt:
 - $\text{ggT}(x, x)$, also $x = y$: Das Resultat ist x .
 - $\text{ggT}(x, y)$ bleibt unverändert, falls die größere der beiden Zahlen durch die Differenz ersetzt wird, also $\text{ggT}(x, y) == \text{ggT}(x, y-x)$, falls $x < y$. Das Ersetzen der Differenz geschieht im folgenden Beispiel iterativ, also durch eine Schleife.

Listing 1.14: Beispiel für `while`-Schleife

```
// cppbuch/k1/ggt.cpp    Berechnung des größten gemeinsamen Teilers
#include <iostream>
using namespace std;

int main() {
    unsigned int x;
    unsigned int y;
    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 1.9 zeigt die Syntax 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 1.10). 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.

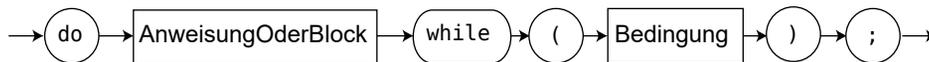
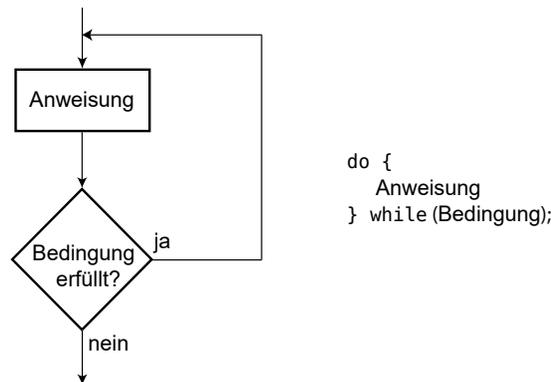


Abbildung 1.9: Syntaxdiagramm einer do while-Schleife



```

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

Abbildung 1.10: Flussdiagramm für eine do while-Anweisung

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.

```

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 1.15: Beispiel für `do while`-Schleifen

```
// cppbuch/k1/primzahl.cpp: Berechnen einer Primzahl, die auf eine gegebene Zahl folgt
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    // Mehrere, durch " getrennte Texte ergeben eine lange Zeile in der Ausgabe.
    cout << "Berechnung_der_ersten_Primzahl,_die_>="
         << "_der_eingegebenen_Zahl_ist\n";
    long z;
    // do while-Schleife zur Eingabe und Plausibilitätskontrolle
    do {
        cout << "Zahl_>_3_eingeben_:";           // Abfrage, solange z ≤ 3 ist
        cin >> z;
    } while(z <= 3);
    if(z % 2 == 0) {
        // Falls z gerade ist: nächste ungerade Zahl nehmen
        ++z;
    }
    bool gefunden {false};
    do {
        // limit = Grenze, bis zu der gerechnet werden muss.
        // sqrt() arbeitet mit double, daher wird der Typ explizit umgewandelt.
        long limit {1 + static_cast<long>( sqrt(static_cast<double>(z)))};
        long rest;
        long teiler {1};
        do {
            // Kandidat z durch alle ungeraden Teiler dividieren
            teiler += 2;
            rest = z % teiler;
        } while(rest > 0 && teiler < limit);
        if(rest > 0 && teiler >= limit) {
            gefunden = true;
        }
        else {
            // sonst nächste ungerade Zahl untersuchen:
            z += 2;
        }
    } while(!gefunden);
    cout << "Die_nächste_Primzahl_ist_" << z << '\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 1.11 zeigt die Syntax einer `for`-Schleife.

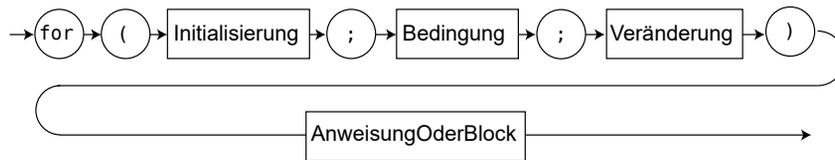


Abbildung 1.11: Syntaxdiagramm einer for-Schleife

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 1.16: Beispiel für for-Schleife

```
// cppbuch/k1/fakultaet.cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Fakultät_berechnen._Zahl_>=_0?_: ";
    unsigned int n;
```

```

cin >> n;
unsigned long fak {1L};
for(unsigned 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:

```

{
    Initialisierung;
    while(Bedingung) {
        Anweisung
        Veraenderung;
    }
}

```

Die äußeren Klammern sorgen dafür, dass in der Initialisierung deklarierte Variablen wie bei der for-Schleife nach dem Ende nicht mehr gültig sind. Anweisung kann wie immer auch eine Verbundanweisung (Block) sein, in der mehrere Anweisungen stehen können, durch geschweifte Klammern begrenzt. Die umformulierte Entsprechung des obigen Beispiels (ASCII-Tabelle von 65 ... 69 ausgeben) lautet:

```

{
    int i {65}; // Initialisierung
    while(i < 70) { // Bedingung
        cout << i << "_ " << static_cast<char>(i) << '\n'; // Anweisung
        ++i; // Veränderung
    }
}

```

*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 statischer Objekte (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.

3.6 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 heraus sucht. Ähnlichkeiten mit der Aufgabe 1.18 von Seite 108 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. Dies ist nicht unbedingt einfach, weil spontan gefundene Beziehungen zwischen Objekten im Programm nicht immer die wesentliche Rolle spielen.



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 1.18 werden alle Aktivitäten in `main()` abgehandelt. Das ist nicht vorteilhaft, 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 Wahl 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 Wahl 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 ausführlich behandelt, zum Beispiel [Oe13]. 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"); // Konstruktor
    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 <vector>
#include "person.h"

class Personalverwaltung {
public:
    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 1.18 von Seite 108 gelöst oder deren Lösung nachgesehen haben.

```
if(pa == nullptr) { // Fehlerhafte Annahme
...
}
```

Der Fehler liegt in dem undefinierten Wert von `pa` nach der Löschoption. Falls ein Zeiger nach dem Löschen noch verwendet werden kann, setzen Sie ihn direkt nach dem `delete` mit `pa = nullptr;` auf Null. Dann kann er geprüft werden und es gibt eine definierte Fehlermeldung. Besser noch ist jedoch die Vermeidung solcher Konstruktionen zugunsten der Kapselung von `new` und `delete` oder der Verwendung von `unique_ptr` bzw. `shared_ptr`, siehe folgenden Abschnitt.

21.2.18 Speicherbeschaffung und -freigabe kapseln

Die Operatoren `new` und `delete` sind stets paarweise zu verwenden. Um Speicherfehler zu vermeiden, empfiehlt sich das »Verpacken« dieser Operationen in Konstruktor und Destruktor wie bei der Vektorklasse des Kapitels 8 oder die Verwendung der »Smart Pointer« (`unique_ptr`, `shared_ptr`), siehe unten. Ein weiterer Vorteil ist die korrekte Speicherfreigabe bei Exceptions (siehe unten).

21.2.19 Programmierrichtlinien einhalten

Das Einhalten von Programmierrichtlinien unterstützt das Schreiben gut lesbarer Programme. Es gibt einige dieser Richtlinien, die sich in großen Teilen ähneln. Oft hat eine softwareentwickelnde Firma eine eigene Richtlinie.

21.3 Exception-sichere Beschaffung von Ressourcen

Wenn eine Ressource beschafft werden soll, kann ein Problem auftreten. Das kann eine Datei sein, die nicht gefunden wird oder ein Fehlschlag beim Beschaffen von Speicher. Weil die Probleme strukturell ähnlich sind, beschränke ich mich hier auf Probleme bei der dynamischen Beschaffung von Speicher. Das kann in einer Methode oder auch schon im Konstruktor auftreten. Ziel ist es, beim Auftreten von Exceptions kein Speicherleck zu erzeugen und die betroffenen Objekte in ihrem Zustand zu belassen.

21.3.1 Sichere Verwendung von `unique_ptr` und `shared_ptr`

Bei der Konstruktion eines `unique_ptr` bzw. `shared_ptr` (Beschreibung in Kapitel 32) soll die Erzeugung des Zeigers mit `new` stets innerhalb der Parameterliste geschehen.

```
Ressource *pr = new Ressource(id);
// weiterer Code
shared_ptr<Ressource> sptr(pr); // 1. falsch!

shared_ptr<Ressource> p(new Ressource(id)); // 2. nicht perfekt, aber richtig!
```

Begründung: Im Fall 1 kann es die folgenden Fehler geben:

- Es wäre möglich, `delete pr` aufzurufen. Bei der Zerstörung von `spr` wird der Destruktor für `*pr` auch aufgerufen, dann also insgesamt *zweimal*.
- Es könnte sein, dass im Bereich »weiterer Code« eine Exception auftritt. Der resultierende Sprung des Programmablaufs aus dem aktuellen Kontext führt dazu, dass `delete` nicht mehr möglich ist. Das erzeugte Objekt bleibt unerreichbar im Speicher.

Im Fall 2 kann dies nicht geschehen: Wenn eine Exception geworfen wird, werden automatisch die Destruktoren aller auf dem Laufzeit-Stack befindlichen Objekte des verlassenen Gültigkeitsbereichs aufgerufen, also auch der Destruktor des `shared_ptr`-Objekts, der wiederum für das Löschen des übergebenen Objekts sorgt – eine Realisierung des Prinzips »Resource Acquisition Is Initialization« (RAII, siehe Glossar). Entsprechendes gilt für `unique_ptr`. Noch besser, weil einfacher, ist die gänzliche Vermeidung von `new`, wie der folgende Abschnitt zeigt.

21.3.2 So vermeiden Sie `new` und `delete`!

Wie gezeigt, muss man sich um `delete` nicht mehr kümmern, wenn `unique_ptr` oder `shared_ptr` eingesetzt werden. Die Hilfsfunktionen `make_unique` (siehe Abschnitt 32.1.1) und `make_shared` (siehe Abschnitt 32.2.1) vereinfachen die Schreibweise weiter, sodass auch `new` entfällt. Dabei werden nur noch die Argumente für den Konstruktor übergeben. Im folgenden Beispiel benötigt der Konstruktor nur ein `int`-Argument:

Listing 21.13: `new` und `delete` vermeiden

```
// vector mit shared_ptr
std::vector<std::shared_ptr<Ressource>> vec1(10);
vec1[0] = std::shared_ptr<Ressource>(new Ressource(1));           // mit new
// einfacher ist:
vec1[0] = std::make_shared<Ressource>(1);                       // ohne new

// vector mit unique_ptr
std::vector<std::unique_ptr<Ressource>> vec2(10);
vec2[0] = std::unique_ptr<Ressource>(new Ressource(2));         // mit new
// einfacher ist:
vec2[0] = std::make_unique<Ressource>(2);                      // ohne new
```

Die Zuweisung im zweiten Beispiel ist nur möglich, weil auf der rechten Seite ein `R`-Wert (temporäres Objekt) steht. Wäre es nicht temporär, gäbe es eine Fehlermeldung des Compilers. Beispiel:

```
auto uniqueptr999 = std::make_unique<Ressource>(999);
vec2[0] = uniqueptr999;                                         // Fehler!
```

Damit wird verhindert, dass es zwei `unique_ptr`-Objekte geben kann, die auf dasselbe Heap-Objekt verweisen. Eine Kopie ist nicht erlaubt.

So vermeiden Sie `new[]` und `delete[]`!

Nach `new[]` nur `delete` statt `delete[]` zu schreiben, wäre ein Fehler. Er ist leicht zu vermeiden, wenn auf `new[]` zugunsten von `vector` verzichtet wird. In den meisten Fällen wird das ohne Probleme möglich sein. Ein Beispiel dafür ist die `String`-Klasse von Seite

257. Manche empfehlen die Verwendung von `unique_ptr<T>` (siehe unten). Innerhalb einer Klasse, deren Objekte kopierbar sein sollen und für die Speicherplatz beschafft werden soll, würde man nur den Destruktor sparen, nicht aber den Kopierkonstruktor und Zuweisungsoperator. Die Verwendung von `vector` spart auch diese ein.

21.3.3 `shared_ptr` für Arrays korrekt verwenden

Der Destruktor eines `shared_ptr`-Objekts wendet `delete` auf den intern gespeicherten Zeiger an, wenn kein anderer `shared_ptr` auf die Ressource verweist. Dies kann zu einem Speicherleck führen, wenn der Zeiger mit `new []` erzeugt wurde, wie auf Seite 220 beschrieben. Zwar kann es sein, dass im Fall der falschen Anweisung das Speicherleck nicht bemerkt wird oder dass der Compiler aus dem Kontext den Fehler erkennt und korrigiert. Nach [ISOC++] ist das Verhalten jedoch undefiniert, das heißt, alle Möglichkeiten vom Weiterlaufen des Programms bis zum Absturz des Programms sind »legal«. Damit ist auch das Verhalten des folgenden Programms undefiniert:

```
void funktion() {
    shared_ptr<int> p(new int[10]);           // falsch
    // ... etliche Zeilen weggelassen
}                                           // Memory-Leak möglich
```

Die Lösung des Problems ist die Übergabe eines `std::default_delete<X[]>`-Objekts an den Konstruktor. Der `shared_ptr`-Destruktor ruft den `operator()()` des übergebenen Objekts auf, wenn kein anderer `shared_ptr` auf die Ressource verweist. Der überladene Funktionsoperator enthält die `delete[]`-Anweisung. Einfacher ist es, beim Typ gleich den Arraytyp `X[]` statt nur `X` zu vermerken. Das Listing 21.14 zeigt beide Fälle. Das Programm dokumentiert die Löschung der Objekte.

Listing 21.14: `shared_ptr` mit Arrays verwenden

```
// cppbuch/k32/sharedptr_arrays.cpp
#include <iostream>
#include <memory>

struct X {
    X(int i = 0)
        : wert(i) {
    }
    ~X() {
        std::cout << "X(" << wert << ")_gelöscht\n";
    }
    int wert;
};

int main() {
    // Zwei Varianten
    std::shared_ptr<X> p1(new X[5], std::default_delete<X[]>());
    std::shared_ptr<X[]> p2(new X[5]);           // <X[]> statt <X>!
    for(int i=0; i < 5; ++i) {
        p1.get()[i].wert = i + 1;               // Zuweisen eines Werts
        p2[i].wert = 10*i + 11;                 // Kurzform geht nur bei shared_ptr<X[]>
    }
}
```

Statt `std::default_delete<X[]>` können Sie eine selbstgeschriebene Klasse nehmen. Sie muss nur die folgende Funktion enthalten:

```
void operator()(T* ptr) { // T = Template-Parameter
    delete[] ptr;
}
```



Tipp

Sie können die beschriebenen möglichen Probleme vermeiden, wenn Sie auf `shared_ptr` für Arrays verzichten und stattdessen einen `shared_ptr` mit einem `vector` verwenden, etwa so: `auto ptr = std::make_shared<std::vector<X>>()`; siehe auch Abschnitt 21.3.2 oben. *Oder noch einfacher:* Genügt vielleicht nur ein `vector<X>` statt eines `shared_ptr` für den Vektor?

21.3.4 `unique_ptr` für Arrays korrekt verwenden

Das Problem des vergessenen Deleters tritt bei `unique_ptr` nicht auf, weil der Typ des für die Löschung zuständigen Objekts zur Schnittstelle gehört.

```
template <class T, class D = default_delete<T>> class unique_ptr;
```

Wenn ein Arraytyp, gekennzeichnet durch `[]`, eingesetzt wird, kann der zweite Typ entfallen. Er wird dann durch den vorgegebenen (default) Typ für den Deleter ersetzt, der `delete []` aufruft. Die Funktion `f()` zeigt, wie es geht.

Listing 21.15: `unique_ptr` und Array

```
// cppbuch/k32/arrayunique.cpp
#include<memory>

void f() {
    std::unique_ptr<int[]> arr = std::make_unique<int[]>(10);
    // entspricht std::unique_ptr<int[]> arr(new int[10]);
    // Benutzung des Arrays weggelassen
} // kein Memory-Leak, Array wird korrekt gelöscht

int main() {
    f();
}
```

Um den voreingestellten (englisch *default*) Template-Parameter sichtbar zu machen, könnte die erste Zeile in `f()` so geschrieben werden:

```
std::unique_ptr<int[], std::default_delete<int[]>>
    arr = std::make_unique<int[]>(10);
```



Tipp

Im Verzeichnis `cppbuch/k11/move/unique_ptr` finden Sie ein Beispiel für einen String-Typ, der einen `unique_ptr` auf ein `char`-Array verwendet. Überlegen Sie sich bei einem ähnlichen Problem aber, ob nicht doch ein `vector` die einfachere Lösung ist.

21.3.5 Exception-sichere Funktion

Listing 21.16: Exception-unsichere Funktion

```
void func() {
    Datum heute;
    Datum *pD = new Datum;
    heute.aktuell();
    pD->aktuell();
    delete pD;
}
// fehlerhaft, siehe Text
// Stack-Objekt
// Heap-Objekt beschaffen
// irgendeine Berechnung
// irgendeine Berechnung
// Heap-Objekt freigeben
```

Wenn die Funktion `aktuell()` eine Ausnahme auswirft, wird der Destruktor von Objekt `heute` gerufen und das Objekt vom Stack geräumt. Das Objekt, auf das `pD` zeigt, wird jedoch niemals freigegeben, weil `delete` nicht mehr erreicht wird und `pD` außerhalb des Blocks unbekannt ist:

Listing 21.17: Anwendung der Funktion von Listing 21.16

```
int main() {
    try {
        func();
    }
    catch(...) {
        //... pD ist hier unbekannt
    }
}
```

Aus diesem Grund sollen ausschließlich Stack-Objekte (automatische Objekte) verwendet werden, wenn Exceptions auftreten. Dies ist immer möglich, wenn Beschaffung und Freigabe eines dynamischen Objekts innerhalb eines Stack-Objekts versteckt werden. Das Hilfsmittel dazu kennen wir bereits, nämlich die »intelligenten« Zeiger aus Abschnitt 8.5:

Listing 21.18: Exception-sichere Funktion

```
void func() {
    // shared_ptr der Standardbibliothek, siehe Abschnitt 8.5.1. Header: <memory>
    auto pDshared = std::make_shared<Datum>();
    pDshared->aktuell();
}
// irgendeine Berechnung
```

Nun ist `pDshared` ein automatisches Objekt. Wenn jetzt eine Exception auftritt, gibt es kein Speicherleck, weil der Destruktor von `pDshared` den beschafften Speicher freigibt.

21.3.6 Exception-sicherer Konstruktor

Das Ziel, den Zustand eines Objekts bei Auftreten einer Exception unverändert zu lassen, ist in diesem Fall nicht erreichbar – das Objekt wird ja erst durch den Konstruktor erzeugt. Es geht also darum, dass

1. Ressourcen, die innerhalb des Konstruktors beschafft werden, freigegeben werden, und dass
2. Exceptions beim Aufrufer aufgefangen werden können.

In diesem Zusammenhang ist es wichtig zu wissen, wie sich C++ verhält, wenn in einem Konstruktor eine Exception auftritt.



Verhalten bei einer Exception im Konstruktor

- Für alle vollständig erzeugten (Sub-)Objekte wird der Destruktor aufgerufen. »Vollständig erzeugt« heißt, dass der Konstruktor bis zum Ende durchlaufen wurde.
- Für *nicht* vollständig erzeugte (Sub-)Objekte wird *kein* Destruktor aufgerufen.

Damit ist klar, dass es nicht mehrere rohe Zeiger, die auf Heap-Objekte verweisen, als Attribute einer Klasse geben sollte. Bei einer Exception bei der Erzeugung des letzten würde der Destruktor des ersten nicht aufgerufen werden. Abhilfe: Heap-Objekte nur mit `unique_ptr` bzw. `shared_ptr` realisieren – oder auf Heap-Objekte verzichten, z.B. weil ein Vektor genommen werden könnte.

21.3.7 Exception-sichere Zuweisung

Wenn bei einer Kopie Speicher beschafft werden muss, ist es zuerst zu erledigen! Der Grund: Falls es dabei eine Exception geben sollte, würden alle folgenden, den Zustand des Objekts verändernden Anweisungen gar nicht erst ausgeführt. Die Problematik findet sich typischerweise beim Kopierkonstruktor und dem Zuweisungsoperator. Dazu gehören auch der Kurzformoperator `+=` und die Bildung temporärer Objekte. Sehen wir uns dazu eine andere (und umständliche) Lösung für den Zuweisungsoperator von Seite 358 an:

Listing 21.19: Exception-sicherer Zuweisungsoperator

```
template<typename T>                                // Exception-sicher
Vektor<T>& Vektor<T>::operator=(const Vektor<T>& v) { // Zuweisung
    T* temp = new T[v.anzahl];                       // zuerst neuen Platz beschaffen
    for(std::size_t i = 0; i < v.anzahl; ++i) {      // kopieren
        temp[i] = v.start[i];
    }
    delete [] start;                                 // alten Platz freigeben
    anzahl = v.anzahl;                              // Verwaltungsinformation aktualisieren
    start = temp;
    return *this;
}
```

Man könnte vordergründig daran denken, erst den alten Platz freizugeben, weil er ohnehin nicht mehr gebraucht wird, und dabei in der Summe sogar Speicher sparen, wenn nämlich bei `new` der alte Speicherplatz wiederverwendet werden sollte. Auch bräuchte man die Variable `temp` nicht:

Listing 21.20: *Nicht* Exception-sicherer Zuweisungsoperator

```
template<typename T>                                // NICHT Exception-sicher!
Vektor<T>& Vektor<T>::operator=(const Vektor<T>& v) { // Zuweisung
    delete [] start;                                 // weg damit, es wird schon gutgehen!
    start = new T[v.anzahl];                         // neuen Platz beschaffen
    for(std::size_t i = 0; i < v.anzahl; ++i) {      // kopieren
        start[i] = v.start[i];
    }
}
```

```

    anzahl = v.anzahl; // Verwaltungsinformation aktualisieren
    return *this;
}

```

Wenn bei der Speicherplatzbeschaffung ein Problem auftreten sollte, wäre der Inhalt des Objekts durch das direkt vorangegangene `delete` zerstört! Von dem Problem, dass `&v == this` sein könnte, will ich gar nicht erst reden.

Der »swap-Trick« liefert eine noch bessere Möglichkeit, die Zuweisung Exception-sicher zu gestalten. Sie sahen ihn bereits auf Seite 358. Dieses Muster lässt sich auf jede Klasse übertragen. Sie muss nur eine passende `swap()`-Methode besitzen:

Listing 21.21: Schema für einen einfachen Exception-sicheren Zuweisungsoperator

```

Klasse& Klasse::operator=(Klasse kopie) { // temporäre Kopie per Wert
    swap(kopie); // wirft keine Exception
    return *this;
}

```

21.4 Empfehlungen zur Thread-Programmierung

21.4.1 Warten auf die Freigabe von Ressourcen

Verwenden Sie die Konstruktionen

```

while(ressourcenNochNichtBereit) {
    cond.wait(lock);
}

```

mit einer Bedingungsvariablen `cond` und einem Lock-Objekt `lock`. Eine Begründung finden Sie auf Seite 517 und davor auch ein Beispiel. Die Alternative

```

while(ressourcenNochNichtBereit) {
    sleep(zeitdauer);
}

```

soll nur genommen werden, wenn sich die Variante mit `wait()` als schwierig erweist; solche Fälle gibt es. Keinesfalls sollten Sie

```

while(ressourcenNochNichtBereit) {
}

```

schreiben – es würde sinnlos CPU-Zeit verbraten. Warum wird oben nicht

```

if(ressourcenNochNichtBereit) { // nicht empfehlenswert
    cond.wait(lock);
}

```

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 [Oe13]. 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 ist in diesem Buch schon einiges gesagt worden, das 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.

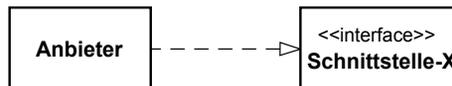


Abbildung 22.2: Interface-Anbieter

Die Ähnlichkeit wird in der Umsetzung nach C++ abgebildet: Anbieter wird von dem Interface SchnittstelleX¹ 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

```

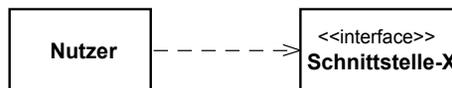
class SchnittstelleX {
public:
    virtual void service(Daten& d) = 0;        // abstrakte Klasse
    virtual ~SchnittstelleX() = default;     // virtueller Destruktor
    SchnittstelleX() = default;
    SchnittstelleX(const SchnittstelleX&) = default;
    SchnittstelleX& operator=(const SchnittstelleX&)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

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 NULL 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 zunächst 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);` oben: `anbieter` kennt durch die Parameterübergabe das Objekt `daten`, wird aber vermutlich die Verbindung nach Ablauf der Funktion lösen.

Einfache gerichtete Assoziation

Das UML-Diagramm einer einfachen gerichteten Assoziation sehen Sie in der Abbildung 22.4.



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 Konstruktoraufwurf 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. * bei Klasse2 besagt, dass einem Objekt der Klasse1 beliebig viele (auch 0) Objekte der Klasse2 zugeordnet sind.

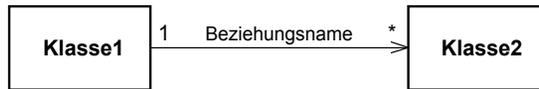


Abbildung 22.5: Gerichtete Assoziation mit Multiplizitäten

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 Popstars, aber nicht umgekehrt.

```

class Fan {
public:
    void werdeFanVon(Popstar* star) {
        meineStars.insert(star);           // zu insert() siehe Seite 841
    }

    void denKannsteVergessen(Popstar* star) {
        meineStars.erase(star);           // 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.

Register

Symbole

* 46, 202, 441
*= 46, 360
+, +=, -, -= 46
++ 46, 362, 437, 441
, 61, 81, 486
->* 249
-- 46, 365
.* 249
... *siehe* Ellipse
/, /= 46
/* ... */ 34
// 34
:: 62, 164, 293
::* 249
; 36, 71
<, <=, >, >= 46, 57
<< 46, 103, 243, 349, 407
<<=, >>= 46
=, == 46, 57, 58
>> 46, 101, 243, 411, 428
?: 71
[] 206, 209, 354
[] []
 Matrixklasse 391
 Zeigerdarstellung 232
141
%, %= 46

& Adress-Operator 59
&, &= Bit-Operatoren 46
&& logisches UND 58
&& Shell 608
&& R-Wert 454
\ 34, 55, 139, 338
\0 210, 214–216
\", \a, \b 55
\f, \n, \r, \t, \v 55, 102
\x 55
\\ 55
~ 46
^ 46
|= 46
|| 58
! 46, 58
!= 57, 58, 441
\$<, \$^, @\$ 605
@D, @F 613
" 36, 210, 211

A

abgeleitete Klasse 279, 288, 292
 und virtueller Destruktor 305
Abhängigkeit (make) 603
 automatische Ermittlung 608
abort() 937
abs() 52, 742, 743, 923, 935, 936

- Abstrakter Datentyp 160, 963
- abstrakte Klasse 299, 963
- accumulate() 695
- acos() 742, 923, 935
- acosh() 742
- Adapter, Iterator- 864
- Additionsoperator 347
- adjacent_difference() 698
- adjacent_find() 726
- adjustfield 409
- Adresse 202
 - symbolische 36
- Adressierung, indirekte (valarray) 931
- Adressoperator 203
- advance() 862
- Aggregat 838
- Aggregation 669, 963
- Aktualparameter 112
- <algorithm> 671, 761, 869
- Algorithmus 30, 437
 - accumulate() 695
 - adjacent_difference() 698
 - adjacent_find() 726
 - all_of 707
 - any_of 707
 - binary_search() 728
 - clamp() 768
 - copy(), copy_backward() 758
 - copy_if() 760
 - copy_n() 760
 - count(), count_if() 706
 - equal() 741
 - equal_range() 729
 - fill(), fill_n() 693
 - find(), find_if(), find_if_not() 721
 - find_end() 725
 - find_first_of() 722
 - for_each(), for_each_n() 758
 - generate(), _n() 694
 - includes() 731
 - inner_product() 696
 - inplace_merge() 720
 - iota() 694
 - is_heap(), is_heap_until() 738
 - is_partitioned() 713
 - is_permutation() 709
 - is_sorted(), is_sorted_until() 714
 - iter_swap() 761
 - lexicographical_compare() 711
 - lower_bound() 729
 - make_heap() 737
 - make_pair() 793
 - make_tuple() 794
 - max(...) 767
 - max_element() 699
 - merge() 718
 - mergesort() 719
 - min(...) 767
 - min_element() 699
 - minmax() 767
 - minmax_element() 700
 - mismatch() 739
 - move(), move_backward() 791
 - next_permutation() 708
 - none_of 707
 - nth_element() 717
 - partial_sort(), -_copy 716
 - partial_sum() 697
 - partition() 712
 - partition_copy(), partition_point() 712
 - pop_heap() 736
 - prev_permutation() 708
 - push_heap() 737
 - remove(), -_if(), -_copy(), -_copy_if() 764
 - replace(), -_if(), -_copy(), -_copy_if() 763
 - reverse(), reverse_copy() 704
 - rotate(), rotate_copy() 701
 - sample() 705
 - search() 723, 725
 - search_n() 727
 - set_difference() 733
 - set_intersection() 733
 - set_symmetric_difference() 734
 - set_union() 732
 - shuffle() 702
 - sort() 714
 - sort_heap() 738
 - split() 672
 - stable_partition() 712

- stable_sort() 715
- swap(), swap_ranges() 761
- transform() 761
- unique(), unique_copy() 702
- upper_bound() 729
- Alias-Name 59, 203, 205
 - *this 227
- alignment_of 815
- alignof() 382, 815
- all_of 707
- allgemeiner Konstruktor 168
- allocator 786
- alternative Funktions-Syntax 129
- Anführungszeichen 36, 210, 211
- Anker (bei regulärem Ausdruck) 492
- anonymer Namespace 136
- ANSI-Sequenzen zur Bildschirm-
ansteuerung 55
- Anweisung 65
- any() (Bitset) 799
- any_of 707
- app 423
- append()
 - Pfad 774
 - String 900
- apply()
 - Tupel 796
 - valarray 920
- Äquivalenzbeziehung 728
- Äquivalenzklasse 628
- arg() 742, 743
- argc 225
- Argument *siehe* Parameter
- Argument Dependent Lookup (ADL) 964
- argv[] 225
- Arität (Template) 480
- Arithmetik
 - mit Iteratoren 440
 - mit Zeigern 208
- arithmetische Operatoren 46
- Array
 - char 212
 - von C-Strings 213
 - dynamisches 218, 233, 234, 351
 - Freigabe 221
 - als Funktionsparameter 224, 230
 - mehrdimensionales 229, 233, 234
 - Matrixklasse 391
 - valarray 917
 - vs. Zeiger 207
- <array> 820, 838
- Array2d 235, 590
- ASCII 879
 - Dateien 242
 - Tabelle 56, 941
- asctime() 938
- asin() 742, 923, 935
- asinh() 742
- assert() 142
 - mit Exception 338
- assign() 825, 901
- Assoziation (UML) 666
- Assoziativität von Operatoren 60, 946
- async() 532
- at() 89, 93, 827, 832, 839, 845, 899
- atan() 742, 923, 935
- atan2() 924, 935
- atanh() 742
- ate 423
- atexit() 937
- atof(), atol() 937
- atoi() 937
- Atom-Uhr 569
- atomic 529
- Attribut
 - Compilersteuerung 947
 - einer Klasse 161, 964
- Aufforderung 31, 964
- Aufzählungstyp 84
- Ausdruck
 - Auswertung 60
 - Definition 42
 - mathematischer 52, 124
- Ausgabe 101, 103, 407
 - benutzerdefinierter Typen 407
 - Datei- 104, 405
 - Formatierung 407
 - Weite der 408
- Ausgabeoperator 349, 407
- Ausnahme 331, 964
 - behandlung 331
- Ausrichtung an Speichergrenzen 381

- Auswertungsreihenfolge 52, 60, 946
- auto 97, 150, 446, 650
- auto[] 98, 794
- automatische Variable 134
 - make 605
- Autotools (GNU) 626
- B**
- back() 826, 829, 832, 835, 838, 899
- back_inserter(), _insert_iterator 865
- Backslash
 - Zeichenkonstante \ 56
 - Zeilenfortsetzung 139, 338
- Backspace 56
- bad() 421
- bad_alloc 337
- badbit 420
- bad_cast 320, 337
- bad_function_call 337
- bad_typeid 323, 337
- bad_weak_ptr 337
- base() 864
- basefield 409
- basic_string 897
- basic_~Streamklassen 406
- Basisklasse 279
 - und virtueller Destruktor 305
 - Konstruktor 313
 - Subobjekt 317
 - virtuelle 315
- Bedingungsausdruck 68
 - make 619
- Bedingungsoperator ?: 71
- beg 424
- begin()
 - Container 442, 822
 - Namespace std 209, 863
 - string 257, 898
 - vector 91
- Belegungsgrad 850
- benutzerdefinierte
 - Datentypen 84, 87
 - Klassen 349
 - Literale 388
 - Typen (Ausgabe) 407
 - Typen (Eingabe) 428
- Benutzungszählung 912
- Bereichsnotation 823, 966
- Bereichsoperator :: 62, 164, 293
 - namespace 145
- Bibliothek
 - C 156, 933
 - C++ 784
- Bibliotheksmodul 132
 - dynamisch 619
 - statisch 618
- Bidirectional-Iterator 858
- Big Three/Five (Regel) 240, 467
- Bildschirmsteuerung mit ANSI-
 - Sequenzen 55
- Binärdatei 244
- binäre Ein-/Ausgabe 241
- binärer Operator 347
 - optimiert 462
- binäres Prädikat 703, 740, 870
- binäre Zahlendarstellung 47
- Binärzahl 44
- binary 107, 423
- binary_search() 728
- bind 802
- Binden 132
 - dynamisches *siehe* dynamisches B.
 - statisches *siehe* statisches Binden
- Bit
 - feld 100
 - Operatoren 46, 47
 - Verschiebung 47
 - pro Zahl 42
- bitset 797
- bitweises ODER, UND, XOR 46
- Block 34, 35, 62, 63, 66, 193
 - und dyn. Objekte 221
- bool 58
- boolalpha 58, 408, 415
- Boost
 - is_any_of() 673
 - lexical_cast() 675
 - split() 673
- Boost.Asio 561
- break 73, 82
- bsearch() 937
- Bstatic (Makefile) 618

- Bucket 850
- bucket() 852
- Byte 54
 - std::byte 86, 936
 - Reihenfolge 569
- C**
- C++-Schlüsselwörter 943
- C-Arrays 205
- C-Funktionen einbinden 158
- C-Header 933
- C-String 210
- call wrapper 514
- Callback-Funktion 246, 804
- canonical() (Filesystem) 774
- capacity() 827, 900
- capturing group 490
- case 73
 - lokale Variablen 74
- <cassert> 142, 933
- cast *siehe* Typumwandlung
- catch 332
- cbegin()
 - Container 822
 - Namespace std 863
 - string 899
- <cctype> 762, 934
- cdecl 252
- ceil() 935
- end()
 - Container 822
 - Namespace std 863
 - string 899
- cerr 101, 407
- char 54, 208
- char* 210
- char16_t, char32_t 880
- char* const vs. const* char 205
- char16_t, char32_t 387
- <chrono> 502
- cin 36, 101, 407, 421
- clamp() 768
- class 162, 287
- class (bei Template-Parametern) 151
- clear() 421, 825, 841, 902
- Client-Server
 - Beziehung 164
 - und callback 246
- clock(), clock_t 938
- clog 101, 407
- close() 105
- CMake 625
- <cmath> 52, 785, 933, 934
- code bloat 622
- collate 680, 884
- combine() 878
- compare() 884, 904
- Compilationsmodell 623
- Compiler 33, 34, 37, 131, 161, 163
 - befehle 944
 - direktiven 130
 - und Templates 154
 - Typumwandlung 185
- <complex> 743
- Computerarithmetik 52
- conditional 817
- configure 626
- conj() 742
- connect() 539
- const 53
 - Elementfunktionen 163, 165
 - globale Konstante 135
- const_pointer 841
- const_cast<>() 321
- const char* vs. char* const 205
- constexpr
 - Funktion 147
 - Konstante 53
 - Konstruktor / Methode 180
- const_iterator 821
- const_local_iterator 852
- const_pointer 826
- const_reference 821
- const_reverse_iterator 824, 865
- const& *siehe* Referenz auf const
- constraint, Vererben von 310
- Container 436
 - implizite Datentypen 821
- Container-Methoden 822
- container_type 833, 834
- continue 82
- copy elision 451

- copy semantics 455
- copy() (Filesystem) 776
- copy() 758, 899
- copy_backward() 758
- copy_if() 760
- copy_n() 760
- copysign() 935
- cos(), cosh() 742, 923, 935
- count() 799, 842
 - Algorithmus 706
- count_if() 706
- cout 36, 101, 407
- __cplusplus 158
- crbegin()
 - Container 824
 - Namespace std 863
 - string 899
- create_directory() 777
- crend()
 - Container 824
 - Namespace std 863
 - string 899
- critical section 511
- CRLF 572
- cshift() 920
- <cstddef> 48, 204, 936
- <cstdlib> 124, 245
- c_str() 899
- <cstring> 211, 247, 255, 258, 939
- <ctime> 364, 937
- ctype 679, 681, 884, 885
- cur 424
- curr_symbol 889
- current_path 774
- <cwctype> 934
- CXXFLAGS 605
- D**
- dangling pointer 220
- data race 511, 518
- data() 826, 838, 899
- Datagramm 567
- __DATE__ 143
- date_order() 891
- Datei
 - ASCII 242
 - binär 244
 - Ein-/Ausgabe 104, 405
 - kopieren 106, 776
 - löschen 774
 - Öffnungsarten 423
 - öffnen 107
 - Positionierung 423
 - schließen 105
 - umbenennen 777
 - Zugriffsrechte 771
- Daten
 - als Attributwerte 965
 - static-Element- 265
- Datenbankanbindung 587
- Datenkapselung 965
- Datentypen 36, 42
 - abstrakte *siehe* Abstrakter Datentyp
 - benutzerdefinierte 84
 - int und unsigned 71
 - logische 58
 - parametrisierte 151, 271
 - polymorphe 303
 - strukturierte 87
 - zusammengesetzte 84
- Datum
 - Klasse / Gültigkeit 362
 - regulärer Ausdruck 750
- dec 408, 415
- decay, decay_t 816
- decimal_point() 887, 888
- decltype 275
- decltype(auto) 278, 693, 790
- declval 278
- default (in switch-Anweisung) 73
- default constructor 167
- Default-Parameter *siehe* vorgegebene P.
- default_order 817
- default_delete<X[]> 656
- = default 176
- #define 137, 139
- Definition 132, 965
 - von static-Elementdaten 265
 - von Objekten 168
- Deklaration 35, 36, 132, 965
 - einer Funktion 111

- Funktionszeiger 244
- Lesen einer D. 250
 - in for-Schleifen 79
- Deklarationsanweisung 65
- Dekrementierung 46, 47
- Dekrementoperator 365
- Delegation 326
- delegierender Konstruktor 179
- delete 217, 220, 268, 304, 306
 - überladen 375
- delete [] 221
- = delete 176, 652
- <deque> 820, 831
- deque 831
- Dereferenzierung 202, 246
- Destruktor 192, 314, 467
 - implizite Deklaration 192
 - und exit() 194
 - virtueller 304
- detach() 508
- Dezimalpunkt 49, 887
- Dialog 550
- diamond problem 317
- difference_type 821
- Differenz (Menge) 733
- difftime() 938
- digits, digits10 766
- distance() 862
- Distribution 607, 965
- div(), div_t 936
- divides 801
- Division durch 0 340
- dll 619
- DNS 559
- do while 77
- domain_error 337
- double 49, 208
 - nicht als Laufvariable 81
 - korrekter Vergleich 648
- downcast 320, 398
- Drei (die großen Drei) 240, 467
- Dubletten entfernen 702
- Durchschnitt (Menge) 733
- DYLD_LIBRARY_PATH 621, 962
- dynamic_cast<>() 320
- dynamic_pointer_cast 913

- dynamisches Array 351
- dynamisches Binden 244, 293, 965
- dynamische Datenobjekte 217
- dynamischer Typ 322

E

- e, E 49
- Editor 32
- effizienter binärer Operator 462
- egrep 489
- Ein- und Ausgabe 101
- Einbinden von C-Funktionen 158
- Eingabe 410
 - aufforderungsfenster 33
 - Datei- 104, 405
 - von Strings 103
 - benutzerdefinierter Typen 428
- Einschränkung *siehe* constraint
- Elementdaten, Zeiger auf 250
- Elementfunktion 160, 282
 - als Funktionsobjekt 805
 - Zeiger auf 249
- Ellipse
 - in catch-Klausel 334
 - template parameter pack 481
- else 67
- #else, #elif 137
- emplace() 823, 833, 835, 837
- emplace_back() 827, 829, 832
- emplace_front() 829, 832
- empty() 822, 833, 835, 836, 899
- enable_if 816
- enable_if_t 817
- end 424
- end()
 - Container 442, 822
 - Namespace std 209, 863
 - string 257, 898
 - vector 91
- #endif 139
- endl 43, 55, 415
 - oder '\n'? 104
- ends 415
- ENTER** 34, 102
- »enthält«-Beziehung 315
- enum 85

- Environment, env[] 225
 - EOF 412
 - eof() 332, 412, 421, 438
 - eofbit 420
 - epsilon() 766
 - equal() 741
 - equal_to 802
 - equal_range() 729, 842
 - equalsIgnoreCase() 681
 - erase() 825, 841, 902
 - ereignisgesteuerte Programmierung 536
 - errno 330
 - error_code 772
 - Exception 964
 - arithmetische Fehler 340
 - und Destruktor 331
 - Handling 331
 - Hierarchie 335
 - Speicherleck durch Exc. 654
 - <exception> 337
 - exception 335
 - Exception-Sicherheit 342
 - ExecutionPolicy 770
 - exists() (Filesystem) 775
 - exit() 124, 937
 - und Destruktor 194
 - Exklusiv-Oder (Menge) 734
 - exp() 742, 923, 935
 - explicit 176
 - explizite Instanziierung von
 - Templates 624
 - Exponent 49–51
 - Expression Templates 748
 - extension() (Filesystem) 774
 - extern 134, 135
 - extern "C" 158
 - extern template 623
 - external linkage 136
- F**
- f, F 49
 - Fünf (die großen Fünf) 467
 - fabs() 742, 935
 - Facette 884
 - fail() 421, 422
 - failbit 420, 429
 - fakultaet() 110
 - [[fallthrough]] 74
 - Fallunterscheidung 72
 - false 58
 - false_type 811
 - falsename() 887
 - Fehlerbehandlung 329
 - Ein- und Ausgabe 420
 - Fibonacci 480, 699
 - __FILE__ 143
 - filename() 774
 - Filesystem
 - canonical() 774
 - copy() 776
 - create_directory() 777
 - current_path() 774
 - exists() 775
 - extension() 774
 - filename() 774
 - is_directory() 775
 - recursive_directory_iterator 779
 - remove() 774
 - remove_all() 775
 - stem() 774
 - fill() 408, 693
 - fill_n() 693
 - final 652
 - find() 841
 - Algorithmus 721
 - string 903
 - find_end() 725
 - find_first_of() 722
 - find_...-Methoden (string) 904
 - findstring 619
 - fixed 408, 410, 415
 - Fixture 636
 - flache Kopie 238
 - flags() 409
 - flip() 798, 828
 - float 49
 - floatfield 409
 - floor() 935
 - flush() 410
 - flush (Manipulator) 415
 - fmod() 935
 - fmtflags 408

- Fold-Expression 484
 - for 78
 - Kurzform 96
 - foreach (make) 614
 - foreach(), for_each_n() 758
 - Formalparameter 112
 - Formatierung der Ausgabe 407
 - forward() 791
 - Forward-Iterator 858
 - <forward_list> 820
 - forward_list 786
 - frac_digits() 889
 - Fragmentierung (Speicher) 222
 - Framework 537
 - free store 217
 - free() 378
 - frexp() 935
 - friend 261
 - front() 826, 829, 832, 835, 838, 899
 - front_inserter(), _insert_iterator 865
 - <fstream> 105
 - fstream 406, 424
 - Füllzeichen 408
 - function 804
 - <functional> 337, 801
 - Funktion 110
 - mit Gedächtnis (static) 113
 - mit initializer_list 823
 - klassenspezifische 265
 - mathematische 52, 934
 - vorgegebene Parameterwerte 121
 - Parameterübergabe
 - per Referenz 119
 - per Wert 115
 - per Zeiger 222
 - rein virtuelle 299
 - mit Definition 300
 - static 265
 - alternative Syntax 129
 - Überschreiben
 - in abgeleiteten Klassen 292
 - virtueller Funktionen 297
 - virtuelle *siehe* virtuelle Funktionen
 - Funktionsobjekte 373, 416
 - function 804
 - mem_fn 805
 - Funktions-Template 151
 - Funktork *siehe* Funktionsobjekte
 - <future> 337
- ## G
- g++ 38
 - Ganzzahlen 42
 - garbage collection 222
 - gcd() 769
 - gegenseitige Abhängigkeit von Klassen 199
 - Genauigkeit 50
 - Generalisierung 280
 - generate(), generate_n() 694
 - generische Programmierung 436
 - GET (http) 573
 - get() 102, 243, 411, 412
 - getaddrinfo() 560
 - getenv() 937
 - getline() für Strings 103, 905
 - getline(char*,...) 412
 - getloc() 434
 - get_money() (Manipulator) 414
 - get_monthname() 891
 - getnameinfo() 560
 - get_time() (Manipulator) 414
 - get_time(), get_weekday(), get_year() (locale) 891
 - ggT 76
 - std::gcd() 769
 - schnell 187
 - Gleichheitsoperator bei Vererbung 399
 - Gleichverteilung 754
 - Gleitkommazahl 53
 - Syntax 49
 - global 62, 63
 - Namensraum 157
 - Variable 134, 135
 - gmtime() 938
 - GNU Autotools 626
 - good() 421
 - goodbit 420
 - Grafische Benutzungsschnittstelle 535
 - greater, greater_equal 802
 - greedy (regex-Auswertung) 492

Grenzwerte von Zahltypen 766
 Groß- und Kleinschreibung 34, 39
 größter gemeinsamer Teiler *siehe* ggT
 grouping() 887, 889
 gslice 927
 gslice_array 930
 guard (Threads) 512
 Gültigkeitsbereich 123
 Block 62
 Datei 136
 Funktion 112
 Klassen 162
 und new 220

H

hängender Zeiger *siehe* Zeiger,
 hängender
 hardware_concurrency() 503
 has_sort_function 812
 has_facet() 878
 hash() 885
 Hash-Funktion 850, 851
 hasher 852
 has_infinity 766
 __has_include 138
 has_value() (optional) 809
 »hat«-Beziehung 669
 Header 36
 C++-Standard 156
 der Standardbibliothek 784
 Header (Http) 572
 Header-Datei 130
 Inhalt 133
 Heap 735
 hex 408, 415
 Hexadezimalzahl 44
 Host Byte Order 569

I

-I Compileroption 137
 iconv 883
 IDE 37
 Identität von Objekten 162, 966
 IEC 60559, IEEE 754 50
 if 67
 if constexpr 484, 812

#if, #ifdef, #ifndef 137
 ifeq 619
 ifstream 105, 406
 ignore() 412
 imag() 742, 743
 imbue() 876
 Implementation 131
 -Vererbung 324
 -sdatei, Inhalt 133
 implizite Deklaration
 Destruktor 192
 Konstruktor 167
 Zuweisungsoperator 356, 396
 in 423
 #include 35, 130, 136
 includes() 731
 Indexoperator 89, 206, 209, 232, 354
 index_sequence 799
 indirekte Adressierung (mit
 indirect_array) 931
 infinity() 766
 Initialisierung
 array 838
 direkte I. der Attribute 168
 und virtuelle Basisklassen 317
 C-Array 208, 229
 einfacher Datentypen 45
 mit Element-Initialisierungsliste 169
 Konstante in Objekten 170, 266
 globaler Konstanten 134, 135
 Reihenfolge 271
 mit {}-Liste 177, 823, 863
 mit konstruktor-interner Liste 266
 von Objekten 166
 mit {}-Liste 168
 von Referenzen 951
 Reihenfolge 170
 in for-Schleife 79
 von static-Elementdaten 265
 struct 87
 und Vererbung 285
 und Zuweisung 45, 172
 initializer_list 178, 823
 Inklusions-Modell 623
 Inkrementierung 46, 47
 Inkrementoperator 362

- inline
 - Elementfunktion 165
 - Funktion 146
 - Konstante 135, 269
 - Variable 147, 269
- inner_product() 696
- inplace_merge() 720
- Input-Iterator 858
- insert() 825, 841, 849, 901
- insert_iterator 866
- inserter() 866
- insert_or_assign 845
- Installation der Software 957
- Instanz 32, 966
- Instanziierung von Templates 273
 - explizite 624
 - ökonomische (bei vielen Dateien) 622
- int 35, 42
- int-Parameter in Templates 274
- intX_t, int_fastX_t, int_leastX_t (X = 8, 16, 32, 64), intmax_t 48
- integer_sequence 799
- integral_constant 811
- integral promotion 62
- Interface (UML) 664
- internal 408, 415
- internal linkage 136
- Internet-Anbindung 557
- Intervall (und Notation) 966
- invalid_argument 337
- <iomanip> 414, 416
- <ios> 414
- ios 406, 420, 422
- ios_base 406
- ios_base::binary 107
- ios_base-Fehlerstatusbits 420
- ios_base-Flags 408, 409
- ios_base-Manipulatoren 415
- ios::failure 421
- ios-Flags zur Dateipositionierung 424
- ios-Methoden 409, 410, 421
- iostate 420
- <iostream> 36, 406, 407, 411, 414
- iota() 694
- IPv4, IPv6 559
- IPv4-Adresse (regulärer Ausdruck) 752
- is() 886
- isalnum(), isalpha() 884, 934
- is_any_of() 673
- is_arithmetic 813
- is_base_of 815
- isblank() 934
- is_bounded 766
- is_class 810
- isctrl() 884, 934
- isdigit() 128, 174, 884, 934
- is_directory() (Filesystem) 775
- is_exact 766
- isgraph() 884, 934
- is_heap(), is_heap_until() 738
- is_iec559, is_integer 766
- islower() 884, 934
- is_modulo 766
- ISO 10646 880
- ISO 8859-1, ISO 8859-15 879
- is_partitioned() 713
- is_permutation() 709
- isprint() 884, 934
- ispunct() 884
- is_same 815
- is_signed 766
- is_sorted(), is_sorted_until() 714
- isspace() 884, 934
- ist-ein-Beziehung 280, 289, 309
- istream 406, 410
- Istream-Iterator 866
- istream::seekg(), tellg() 423
- istream::ws 415
- istringstream 406, 425
- isupper() 884, 934
- isxdigit() 884, 934
- iter_swap() 761
- Iterator 257, 437, 441, 857
 - Adapter 864
 - Bidirectional 858
 - Forward 858
 - Input 858
 - Insert 865
 - Output 858
 - Random Access 858

- Reverse 864
 - Stream 866
 - Tag 859
 - Zustand 441
 - <iterator> 857
 - iterator 821
 - iterator_category 858
- J**
- Jahr 938
 - join() 506, 508
 - Jota 694
- K**
- Kardinalität 667
 - Kategorie (locale) 884
 - key_equal 852
 - key_type 841, 848
 - key_comp(), key_compare 842
 - value_comp(), value_compare 842
 - Klammerregeln 60
 - Klasse 31, 161, 966
 - abgeleitete *siehe* abgeleitete Klasse
 - abstrakte 299
 - Basis- *siehe* Basisklasse
 - Deklaration 163
 - konkrete 299
 - Ober- *siehe* Oberklasse
 - für einen Ort 162
 - Unter- *siehe* Unterklasse
 - für rationale Zahlen 183
 - Klassenname 323
 - klassenspezifische
 - Daten und Funktionen 265
 - Konstante 269
 - Klassen-Template 271
 - Klassifikation 280, 967
 - Kleinschreibung 34
 - kleinstes gemeinsames Vielfaches *siehe* lcm()
 - Kollisionsbehandlung 850
 - Kommandointerpreter 603, 606
 - Kommandozeilenparameter 225
 - Kommaoperator 61, 81, 486
 - Kommentar 34
 - komplexe Zahlen 742
 - Komplexität 972
 - Komposition 670
 - konkrete Klasse 299
 - Konsole 33
 - Konsole auf UTF-8 einstellen 879
 - Konstante 53
 - globale 134, 135
 - klassenspezifische 269
 - konstante Objekte 164
 - Konstruktor 163, 166
 - allgemeiner 168
 - implizite Deklaration 167
 - delegierender 179
 - erben 290
 - Kopier- *siehe* Kopierkonstruktor
 - vorgegebene Parameterwerte 169
 - Typumwandlungs- *siehe* Typumwandlungskonstruktor
 - Kontrollstrukturen 65
 - Konvertieren von Datentypen *siehe* Typumwandlung
 - Kopie, flache/tiefe 238
 - Kopieren
 - von Dateien 106
 - von Objekten 238
 - von Zeichenketten 215
 - Kopierkonstruktor 171, 354, 467
 - Auslassung durch Compiler 173
 - Kreuzreferenzliste 690
 - kritischer Bereich 511
 - Kurzform-Operatoren 46
- L**
- l, L 49
 - L-Wert 66, 206, 212, 354, 967
 - Länge eines Vektors 697
 - Lambda-Funktionen 469
 - LANG 876
 - late binding 244
 - Laufvariable 79, 80
 - Laufzeit 202
 - und Funktionszeiger 244
 - und new 217
 - und Polymorphie 293
 - Typinformation 322

- lcm() 769
 - LD_LIBRARY_PATH 621
 - ldd 620
 - ldexp() 935
 - ldiv(), ldiv_t 936
 - left 408, 415
 - length() 899
 - length_error 337
 - less, less_equal 802
 - lexical_cast() 675
 - lexicographical_compare() 711
 - lexikografischer Vergleich 793, 822, 967
 - <limits> 43, 49, 766
 - __LINE__ 143
 - Linken 135, 156
 - dynamisches 619, 967
 - internes, externes 136
 - statisches 618, 967
 - Linker 37
 - linksassoziativ 60, 228
 - list 829
 - <list> 787, 820
 - Liste
 - Initialisierungs- 169, 266
 - Initialisierungs- (bei C-Arrays) 229
 - Liste (Klasse) 442
 - Literal 211, 967
 - benutzerdefiniert 388
 - String 387
 - Zahl- 53
 - Zeichen- 54, 880
 - load_factor() 852
 - local_iterator 852
 - <locale> 875
 - localtime() 364, 938
 - lock_guard 512
 - log(), log10() 742, 923, 935
 - logic_error 336, 337
 - logical_and, !_not, !_or 802
 - logischer Datentyp 58
 - logische Fehler 338
 - logische Negation 58
 - lokal (Block) 62
 - lokale Objekte 205
 - long 42
 - long double 49
 - lower_bound() 729, 843
 - lvalue *siehe* L-Wert
- ## M
- magic number 968
 - main() 34, 35, 123
 - MAKE 613
 - make 601
 - automatische Ermittlung von Abhängigkeiten 608
 - parallelisieren 617
 - rekursiv 612
 - Variable 605
 - Makefile 132, 603
 - make_from_tuple() 795
 - make_heap() 737
 - make_index_sequence 799
 - make_integer_sequence 799
 - make_pair() 793
 - make_shared 655, 913
 - make_tuple() 794
 - make_unique 253, 466, 655, 911
 - Makro 139
 - malloc() 378
 - Manipulatoren 413
 - Mantisse 50
 - <map> 820, 843
 - mapped_type 845
 - mask_array 930
 - match_results 496
 - mathematischer Ausdruck 52
 - mathematische Funktionen 934
 - Matrix
 - C-Array 229
 - C-Array, dynamisch 233
 - Klasse 235
 - Klasse, mit zusammenhängendem Speicher 744
 - Vektor von Vektoren 390
 - max() 766, 767, 920
 - max(initializer_list<T>) 767
 - max_size() 822
 - max_bucket_count() 852
 - max_element() 699
 - max_exponent, max_exponent10 766

- max_load_factor() 853
 - [[maybe_unused]] 947
 - mehrdimensionales Array 229
 - mehrdimensionale Matrizen 390
 - Mehrfachvererbung 282, 312, 314
 - MeinString (Klasse) 255
 - mem_fn() 805
 - member function *siehe* Elementfunktion
 - memcpy() 353
 - <memory> 337, 787
 - memory leak 221, 654
 - Mengenoperationen auf sortierten Strukturen 731
 - merge() 718, 830, 842
 - mergesort() 719
 - messages 884, 893
 - Metaprogrammierung 478
 - Methode 31, 32, 160, 968
 - Regeln zur Konstruktion von Prototypen 645
 - MIME 968
 - min() 766, 767, 920
 - min(initializer_list<T>) 767
 - min_element() 699
 - min_exponent, min_exponent10 766
 - minmax(...) 767
 - minmax_element() 700
 - minus 801
 - Minute 938
 - mischen 718
 - mismatch() 739
 - mktime() 938
 - modf() 935
 - modulare Gestaltung 129
 - Modulo 46
 - modulus 801
 - Monat 938
 - monetary 884, 888
 - money_get 889
 - money_punct 888
 - money_put 890
 - Monitor-Konzept 522
 - move semantics 455
 - move() R-Wert 459
 - move(), move_backward()
 - Container-Bereich 791
 - moving constructor 458
 - mt19937 702, 753
 - multimap 847
 - multiplies 801
 - Multiplikationsoperator 360
 - Multiplizität (UML) 667
 - multiset 849
 - mutable
 - Attribut 165
 - Lambda-Funktion 474
 - mutex 511
- ## N
- '\n'
 - oder endl? 104
 - und regex_replace 498
 - Nachbedingung 129, 968
 - Nachkommastellen 49
 - precision 427
 - Name 39
 - einer Klasse 323
 - name() 323, 878
 - Namenskonflikte bei Mehrfachvererbung 314
 - Namenskonventionen 39
 - Namespace 64, 144
 - anonym 136
 - in Header-Dateien 142
 - Verzeichnisstruktur 611
 - namespace 35, 144
 - namespace std 64
 - narrow() 886
 - nationale Sprachumgebung 876
 - NDEBUG 142, 338
 - negate 801
 - Negation
 - bitweise 46, 48
 - logische 58
 - Negationsoperator (überladen) 422
 - negative_sign() 889
 - neg_format() 889
 - Network Byte Order 569
 - Netzwerkprogrammierung 557
 - neue Zeile 56, 67

- new 217, 220, 268
 - Fehlerbehandlung 341
 - überladen 375
- <new> 337, 915
- new_handler 341
- new Placement-Form 915
- next() 862
- next_permutation() 708
- noboolalpha 415
- [[nodiscard]] 947
- noexcept 334
- none() 799
- none_of 707
- norm() 742, 743
- Normalverteilung 756
- noshowbase, -point, -pos 415
- noskipws 415
- not_equal_to 802
- nothrow 342
- notify_one(), --all() 517, 662
- nounitbuf, nouppercase 415
- npos 898
- NRVO *siehe* RVO
- nth_element() 717
- NTP – Network Time Protocol 569
- NULL 204, 936
- nullopt 809
- nullptr 204
- Null-Zeiger und new 342
- numeric 884, 887
- numeric_limits 49, 766
- numerische Auslöschung 51
- numerische Umwandlung 673, 675, 906
- num_get, num_put 887
- NumeriertesObjekt (Klasse) 266
- numpunct 887
- O**
- O-Notation 972
- Oberklasse 279, 292, 968
 - erben von 282
 - Subobjekt einer 285
 - Subtyp einer 288
 - Zugriffsrechte vererben 286
- Oberklassenkonstruktor 282, 285
- object slicing 289
- Objekt 30, 161, 163, 969
 - code 37
 - dynamisches 217
 - als Funktions- 373
 - hierarchie 315
 - Identität *siehe* Identität von Objekten
 - Initialisierung 166
 - konstantes 164
 - orientierung 159
 - Übergabe per Wert 172
 - verkettete Objekte 219
 - verwitwetes 221
 - vollständiges 317, 972
- Objektcode 37
- oct 408, 415
- ODER
 - bitweises 46
 - logisches 58
- Öffnungsarten für Streams 423
- offsetof 936
- ofstream 105, 406
- Oktalzahl 44
- omanip 416
- one definition rule 133
- open() 105
- Open Source 969
- Operator
 - arithmetischer 46
 - binärer 347
 - Bit- 46
 - für char 57
 - als Funktion 346
 - Kurzform 47
 - für Literale 386
 - für logische Datentypen 58
 - Präzedenz 60, 945
 - relationale 46, 58, 789
 - Syntax 346
 - Typumwandlungs- 366
 - überladen (<<) 407
 - überladen (>>) 428
 - unärer 347
 - für ganze Zahlen 46
- operator!() 422
- operator delete() 375

- operator new() 375
- operator string() 366
- operator()() 373
- operator*() 361, 367, 441
- operator*=() 360
- operator++() 362, 364, 441
- operator++(int) 650
- operator+=() 349
- operator->() 367
- operator<<() 349, 407, 797, 798
- operator<<=() 798
- operator=() 358
- operator==() 441, 799
 - bei Vererbung 399
- operator>>() 411, 797, 798
- operator>>=() 798
- operator[]() 393, 396, 827, 832, 839, 859
- operator&() 797
- operator&=() 798
- operator^=() 798
- operator^() 797
- operator|() 797
- operator|=() 798
- operator~() 798
- operator!=() 441, 799
- Optimierung durch Vermeiden
 - temporärer Objekte 173
- optional 809
- Optionale Objekte 808
- Ort (Klasse) 162
- ostream 349, 406, 407
- Ostream-Iterator 866
- ostream::endl, ends, flush() 415
- ostream::seekp(), tellp() 423
- ostreamstringstream 406, 425
- out 423
- out_of_range 337
- Output-Iterator 858
- overflow 45, 51
- overflow_error 337
- override 302, 651
- P**
- pair 792
- Parameter
 - expansion 482
 - einer Funktion 111
 - Pack 481
 - übergabe
 - per Referenz 119
 - per Wert 115
 - per Zeiger 222
 - parametrisierte Datentypen 151, 271
 - »part-of«-Beziehung 669
 - partial_sum() 697
 - partial_sort(), partial_sort_copy 716
 - partielle Spezialisierung von
 - Templates 858
 - partition() 712
 - partition_copy(), partition_point() 712
 - path 771
 - patsubst 607
 - peek() 413
 - perfect forwarding 791
 - Performance 449
 - Permutationen 708
 - PHONY 604
 - π 53, 181
 - Placement new/delete 915
 - plus 801
 - POD (plain old data type) 969
 - pointer 826, 841
 - Pointer, smarte 367
 - polar() 742
 - polymorpher Typ 303, 322
 - Polymorphismus 293, 969
 - pop() 833, 835, 837
 - pop_back() 827, 829, 832
 - pop_front() 829, 832
 - pop_heap() 736
 - portabel (Zeichensatz) 882
 - pos_format() 889
 - Positionierung innerhalb einer Datei 424
 - positive_sign() 889
 - POSIX 876
 - POST (http) 577
 - postcondition *siehe* Nachbedingung
 - Postfix-Operator 362
 - pos_type 423
 - pow() 742, 924
 - Prädikat

- Algorithmus mit P. 870
 - binäres 703, 740, 870
 - unäres 706
- Präfix-Operator 362
- Präprozessor 130, 136, 608
- Präzedenz von Operatoren 60, 945
- precision() 410
- precondition *siehe* Vorbedingung
- prev() 862
- prev_permutation() 708
- PRINT (Makro) 141
- printf() 480
- Priority-Queue 836
- private 163, 286
- private Vererbung 324
- Programm
 - ausführbares 37
 - Strukturierung 109
- Programmierrichtlinien 654
- Projekte 132
- protected 286
- protected-Vererbung 327
- Prototyp
 - Funktions- 110
 - einer Methode 162
 - Regeln zur Konstruktion 645
- ptrdiff_t 936
- public 163, 282, 286
- push() 833, 835, 836
- push_heap() 737
- push_back() 827, 829, 832
 - vector 93
- push_front() 829, 832
- put() 105, 243, 407
- putback() 412
- put_money(), _time() (Manipulator) 414

Q

- qsort() 245, 937
- Qt 537
- QThread 554
- Quantifizierer 492
- Queue 834
- <queue> 787, 820, 834
- quoted() 414

R

- R-Wert 66, 206, 967
 - Referenz 455
- race condition 511, 534
- radix 766
- RAII 512, 636, 655, 970
- <random> 702
- Random-Access-Iterator 858
- random_device 753
- range_error 337
- <ratio> 806
- Rationale Zahl
 - Klasse 183
 - Template std::ratio 806
- rbegin()
 - Container 824, 864
 - Namespace std 863
 - string 899
- rdstate() 421
- read() 241
- real() 742, 743
- Rechengenauigkeit 50, 81
- rechtsassoziativ 60, 228
- recursive_directory_iterator 779
- reelle Zahlen 49
- ref(), reference_wrapper 514, 808
- reference
 - bitset 797
 - Container 821
 - vector<bool> 827
- reference collapsing rules 454
- reference counting 912
- Referenz 59, 127
 - auf Basisklasse 297
 - auf const 119, 171
 - auf istream 411, 428
 - auf Oberklasse 289, 296
 - auf ostream 350, 407
 - Parameterübergabe per 119
 - Rückgabe per 355
 - auf R-Wert 455
- Referenzsemantik 239, 449
- <regex> 337, 497
- regex_iterator 496
- regex_match() 497
- regex_replace() 498, 685

- regex_search() 498, 683
 - reguläre Ausdrücke 489
 - Reihenfolge
 - Auswertungs- 52, 60
 - der Initialisierung 170
 - umdrehen 704
 - rein virtuelle Funktion 299
 - mit Definition 300
 - reinterpret_cast<>() 208, 241, 321
 - Rekursion 116
 - Template-Metaprogrammierung 478, 482
 - rekursiver Abstieg 124
 - rekursiver Make-Aufruf 612
 - rel_ops 789
 - relationale Operatoren 46, 58, 789
 - remove()
 - Algorithmus 764
 - Datei/Verzeichnis 774
 - Liste 830
 - remove_if() 764, 830
 - remove_reference 790, 816
 - remove_reference_t 816
 - remove_all() Dateien/Verzeichnisse 775
 - rename() Datei/Verzeichnis 777
 - rend()
 - Container 824, 864
 - Namespace std 863
 - string 899
 - replace() 763, 902
 - replace_copy(), _if(), _copy_if() 763
 - reserve() 827, 900
 - reset() 798
 - resetiosflags() 414
 - resize() 827, 830, 832, 900
 - return 36, 173
 - return value optimization *siehe* RVO
 - reverse() (list) 830
 - reverse(), reverse_copy() 704
 - reverse_iterator 824
 - Reverse-Iterator 864
 - Reversible Container 824
 - right 408, 415
 - »rohes« Stringliteral 387
 - rotate(), rotate_copy() 701
 - round_error(), round_style() 766
 - RTTI 322
 - Rule of zero/three/five 467
 - runtime_error 337
 - rvalue *siehe* R-Wert
 - RVO 173, 451, 463
- ## S
- sample() 705
 - scan_is(), scan_not() 886
 - Schleifen 75
 - und Container 95
 - do while 77
 - for 78
 - und Strings 213
 - Tabellensuche 91
 - terminierung 76
 - while 75
 - Schlüsselwörter 943
 - Schnittmenge 733
 - Schnittstelle 131, 970
 - einer Funktion 114
 - Regeln zur Konstruktion 645
 - scientific 408, 410, 415
 - scope 62
 - scoped locking 512
 - search() 723, 725
 - search_n() 727
 - sed 615, 616
 - seekg(), seekp() 423
 - Seiteneffekt 68, 111, 214, 216
 - im Makro 143
 - Seitenvorschub 56
 - Sekunde 938
 - Selektion 67
 - sentinel 91, 209
 - sentry 434
 - Sequenz
 - konstruktor 176, 354
 - 2-dim. Matrixklasse 747
 - methoden (Container) 825
 - Server-Client
 - Beziehung 164
 - und callback 246
 - <set> 820, 847
 - set() 798
 - setbase() 414

- set_difference() 733
- setf() 409
- setfill() 414
- set_intersection() 733
- setiosflags() 414
- setprecision() 414
- setstate() 421
- set_symmetric_difference() 734
- set_terminate() 338
- set_union() 732
- setw() 414
- SFINAE 812, 970
- shared_lock 524
- shared_mutex 524
- shared_ptr 373, 912
 - für Arrays 656
- SHELL 606
- shift() 920
- short 42
- showbase, showpoint, showpos 408, 415
- showContainer() 693
- shrink_to_fit() 827, 832, 900
- shuffle() 702
- Sichtbarkeit 62
 - sbereich (namespace) 144
 - dateübergreifend 134
- sign() 935
- Signal 538, 541
- Signalton 56
- Signatur 122, 282, 293, 294, 296
- signed char 54
- sin(), sinh() 742, 923, 935
- single entry/ single exit 83
- size() 89, 93, 822
- size() (std::size()) 89, 207
- size_t 48, 936
- size_type 821
- sizeof 207
 - nicht bei dynamischen Arrays 234
- sizeof... (variadische Templates) 482
- Skalarprodukt 696
- skipws 408, 415
- sleep_for(), sleep_until() 502, 508
- slice 924
- slice_array 927
- Slot 539, 541
- small string optimization 466
- Smart Pointer 367
 - und Exceptions 654
- Socket 562
- Sommerzeit 938
- Sonderzeichen 56
- sort_heap() 738
- Sortieren
 - mit qsort() 246
 - mit sort() 714
 - stabiles 714
 - durch Verschmelzen 719
- Speicher
 - klasse 134
 - leck 221, 654
 - platzfreigabe 204
 - verwaltung (eigene) 381
- Spezialisierung
 - von Klassen 280
 - von Templates 153
- splice() 830
- split() 672
- Sprachumgebung 876
- SQL 587
- sqrt() 52, 742, 923, 935
- <sstream> 425
- stable_partition() 712
- stable_sort() 715
- Stack 62
 - Klasse 271
- <stack> 787, 820, 833
- stack unwinding 331
- Standard
 - bibliothek
 - C 156, 933
 - C++ 784
 - header 157, 787
 - klassen 787
 - Typumwandlung 61, 248
 - Zeiger 248
- Standard Ein-/Ausgabe 101
- start() 924, 928
- static 134
 - Attribute und Methoden 265
 - in Funktion 113
 - Initialisierungsreihenfolge 271

- static_assert 143
- static_cast<>() 56, 319
- static und -Bstatic (Makefile) 618
- statisches Binden 293, 971
- Statusabfrage einer Datei 421
- std 35, 64, 145
- <stdexcept> 337
- stdio 408
- Stelligkeit (Template) 480
- stem() (Filesystem) 774
- Stichprobe 705
- STL 435
- stod() 674
- stoi() 673, 674
 - und verwandte numerische
Konversionsfunktionen 906
- strcat(), strchr(), strcmp() 939
- strncat(), strncmp() 940
- strpbrk(), strrchr(), strstr() 940
- strcpy() 217, 939
- strncpy() 939
- stream 405
- Stream-Iterator 866
- Stream-Öffnungsarten 423
- streamsize 412
- strerror() 940
- Streuspeicherung 849
- strftime() 938
- stride() 924, 928
- String 93
 - in Zahl umwandeln 673
 - Klasse MeinString 255
 - Länge 214, 215
 - Literal 387, 880
- <string> 787, 897
- string 93, 897
 - append() 900
 - assign() 901
 - at() 93, 899
 - back() 899
 - begin() 898
 - capacity() 900
 - cbegin(), cend(), crbegin(), crend() 899
 - clear() 902
 - compare() 904
 - copy() 899
 - c_str() 899
 - data() 899
 - empty() 899
 - end() 898
 - erase() 902
 - find() 903
 - find_...-Methoden 904
 - front() 899
 - insert() 901
 - length() 93, 899
 - max_size() 899
 - operator+=() 900
 - rbegin(), rend() 899
 - replace() 902
 - reserve() 900
 - resize() 900
 - shrink_to_fit() 900
 - size() 93, 899
 - substr() 904
 - swap() 899
- stringstream 674
- string_view 264, 907
- <string_view> 787
- strlen() 211, 215, 940
- strncpy() 217, 940
- strtod(), strtol(), strtoul() 937
- strtok() 940
- struct 87, 287
- Strukturierte Bindung 98
- Stunde 938
- Subobjekt 282, 289, 313
 - in virtuellen Basisklassen 315
 - verschiedene 315
- Substitutionsprinzip 309
- substr() 904
- Subtyp 283, 288, 309, 971
- Suffix 49
- sum() 920
- swap() 358, 822, 899
 - Algorithmus 761
- swap-Trick 240
- swap_ranges() 761
- switch 72
- symmetrische Differenz (Menge) 734
- Synchronisation 510

- Syntaxdiagramm
 - ?: Bedingungsoperator 71
 - do while-Schleife 77
 - enum-Deklaration 85
 - for-Schleife 78
 - Funktions-
 - aufruf 112
 - definition 112
 - prototyp 111
 - Template 151
 - if-Anweisung 67
 - mathematischer Ausdruck 124
 - operator-Deklaration 346
 - struct-Definition 87
 - switch-Anweisung 72
 - Typumwandlungsoperator 366
 - while-Schleife 75
- system() 937
- system_clock 502
- system_error 337, 421
- Szenario 195
- T**
- Tabellensuche 91
- Tabulator 56
- Tag (des Monats) 938
- tan(), tanh() 742, 923, 935
- target (make) 603
- Taschenrechnersimulation 124
- Tastaturabfrage 102
- TCP 558
- »Teil-Ganzes«-Beziehung 669
- tellg(), tellp() 423
- Template
 - Alias 252
 - für Funktionen 151
 - Instanziierung von T. 273
 - explizite 624
 - ökonomische (bei vielen Dateien) 622
 - int-Parameter 274
 - für Klassen 271
 - Konstante 181
 - Metaprogrammierung 478
 - Spezialisierung 153
 - partielle 858
 - static Attribut 384
 - variable Parameterzahl 480, 748
- Template Method (Design-Muster) 307
- Template-Alias type 816
- temporäres Objekt (Vermeidung) 173
- Terminal 33
- terminate() 337
- terminate_handler 338
- test() (Bitset) 799
- Test Driven Development 629
- Test-Suite 631
- Textersetzung 139
- this 227
- this_thread 506
- this->, *this bei Zugriff auf Oberklassenelement 361
- thousands_sep() 887, 888
- Thread 501
- thread (API) 507
- Thread-Sicherheit 534
- ThreadGroup 509
- throw 332
- tie() 434
- tiefe Kopie 238
- __TIME__ 143
- time 884, 891
- time() 364, 939
- time_get 891
- time_put 892
- time_t 364, 937
- tm 364, 937
- tolower() 679, 884, 886, 934
- top() 833, 836
- to_string()
 - bitset 799
 - Zahlkonvertierung 906
- to_ulong() 799
- toupper() 679, 881, 884, 886, 934
- trailing return type 129
- traits 857
- traits::eof() 412
- transform() 761, 885
- tree (Programm) 610, 779
- Trennung von Schnittstellen und Implementation 132
- true 58

- true_type 811
 - truename() 887
 - trunc 423
 - try 332
 - Tupel, <tuple> 748, 794
 - tuple_cat() 749, 795
 - Typ 971
 - polymorpher bzw. dynamischer Typ 322
 - type cast *siehe* Typumwandlung
 - Type Traits 810
 - Abfrage von Eigenschaften 813
 - Abfrage numerischer Eigenschaften 815
 - decay 816
 - default_order 817
 - enable_if 816
 - Funktionsweise 811
 - Typbeziehungen 815
 - Typumwandlungen 816
 - typedef 251
 - typeid() 322, 401
 - type_info 322
 - <typeinfo> 337
 - typename (bei Template-Parametern) 151
 - Typinformation 304
 - zur Laufzeit 322
 - Typumwandlung
 - cast 56, 204, 246, 319
 - durch Compiler 185
 - const_cast<>() 321
 - dynamic_cast<>() 320
 - mit explicit 176
 - implizit 71, 176
 - mit Informationsverlust 123
 - skonstruktoren 174, 185
 - soperator 366
 - ios 422
 - reinterpret_cast<>() 321
 - Standard- 61
 - Zeiger 248
 - static_cast<>() 319
- U**
- u16string, u32string 387
 - u8 880
 - UCS 880
 - UDP 558, 567
 - Überladen
 - von Funktionen 122
 - von Operatoren *siehe* operator
 - Überlauf 45, 51
 - Überschreiben
 - von Funktionen in abgeleiteten Klassen 292
 - virtueller Funktionen 297
 - Übersetzung 130, 132
 - seinheit 132
 - uintX_t, uint_fastX_t, uint_leastX_t (X = 8, 16, 32, 64), uintmax_t 48
 - Umgebungsvariable 225
 - UML 279, 663
 - Umleitung der Ausgabe auf Strings 425
 - unärer Operator 347
 - unäres Prädikat 706
 - UND
 - bitweises 46, 47
 - logisches 58
 - #undef 139
 - undefined behaviour 221
 - underflow 52
 - underflow_error 337
 - Unicode 405, 879
 - uniform_int_distribution 754
 - uniform_real_distribution 755
 - union 99
 - unique() Sequenzen 830
 - unique(), -_copy() Algorithmen 702
 - unique_lock 512
 - unique_ptr 253, 373, 466, 909
 - für Arrays 657
 - Unit-Test 627
 - unitbuf 408, 410, 415
 - unordered_map 853
 - unordered_multimap 855
 - unordered_multiset 856
 - unordered_set 855
 - unsigned 43
 - unsigned char 54
 - Unterklasse 279, 971
 - upper_bound() 729, 843
 - uppercase 408, 415
 - URI, URL 558, 685

URL-Codierung 572
 use case 194
 use_facet() 679–681, 878
 using
 -Deklaration 325
 Namespace
 Deklaration 145
 Direktive 145
 protected Funktion 323
 statt typedef 251
 UTC 938
 UTF-8 879
 <utility> 278, 459, 465, 761, 790, 794

V

Valarray
 arithmetische Operatoren 921
 Bit-Operatoren 922
 logische Operatoren 922
 mathematische Funktionen 923
 relationale Operatoren 923
 <valarray> 917
 value(), value_or() (optional) 809
 value_type 821, 841, 845
 Variable 36
 automatische 134
 globale 134, 135
 make 605
 Variablenname 39
 variadic templates 480, 748
 variant 800
 vector 826
 at() 89
 push_back() 93
 size() 89
 <vector> 787, 820, 826
 vector<bool> 827
 Vektor 88
 Klasse 351
 Länge (geom.) 697
 Verbundanweisung 66
 Vereinigung (Menge) 732
 Vererbung 279, 971
 der abstrakt-Eigenschaft 299
 von constraints 310
 der Implementierung 325

 Mehrfach- 312
 private 324
 protected 327
 von Zugriffsrechten 286
 und Zuweisungsoperator 396
 Vergleich
 von double-Werten 648
 bei Vererbung 399
 verschmelzen (*merge*) 718
 Vertrag 310, 971
 verwitwetes Objekt 221
 Verzeichnis
 anlegen 777
 anzeigen 778
 kopieren 776
 löschen 774
 umbenennen 777
 Verzeichnisbaum
 anzeigen 779
 make 611
 Verzweigung 67
 virtual 294, 296, 306
 virtuelle Basisklasse 315
 virtueller Destruktor 304
 virtuelle Funktionen 294, 296
 private 307
 rein- 299
 void 204
 als Funktionstyp 111
 void* 246
 Typumwandlung nach 204
 Typumwandlungsoperator 422
 volatile 972
 vollständiges Objekt 317, 972
 Vorbedingung 129, 972
 vorgegebene Parameterwerte
 in Funktionen 121
 in Konstruktoren 169
 Vorkommastellen 49
 Vorrangregeln 60, 945
 Vorwärtsdeklaration 199

W

Wächter (Tabellenende) 91, 209
 Wahrheitswert 58
 zufällig erzeugen 756

- wait() 516, 523
- Warteschlange 834
- wchar_t 55, 880, 936
- weak_ptr 914
- Webserver 578
- Weite der Ausgabe 408
- Wert
 - eines Attributs 964
 - Parameterübergabe per 115
- Wertebereich ganzer Zahlen 44
- Wertsemantik 239, 437, 449
 - Performanceproblem 451
- what() 336
- while 75, 213, 215
- whitespace *siehe* Zwischenraumzeichen
- wide character 55
- widen() 886
- Widget 541
- width() 408
- Wiederverwendung
 - durch Delegation 326
 - durch Vererbung 289
- wildcard 607
- Winterzeit 938
- Wochentag 938
- wofstream 883
- Wrapperklasse für Iterator 864
- write() 241, 407
- ws 415
- wstring 387, 883, 897

- X**
- XOR, bitweises 46

- Y**
- yield() 507

- Z**
- Zahl in String umwandeln 675
- Zahlenbereich 43, 50
- Zeichen 54
- Zeichenkette 36, *siehe auch* String
 - C-String 210
 - Kopieren einer 215
- Zeichenklasse (Regex) 492
- Zeichenliteral 880
- Zeichensatz 879
- Zeiger 201, 217
 - Arithmetik 208
 - vs. Array 207
 - auf Basisklasse 297, 304
 - Darstellung
 - von [] 209
 - von [][] 232
 - auf Elementdaten 250
 - auf Elementfunktionen 249
 - auf Funktionen 244
 - hängender 220
 - intelligente *siehe* Smart Pointer
 - Null-Zeiger 203
 - auf Oberklasse 289, 296
 - auf Objekt (Mehrfachvererbung) 314
 - auf lokale Objekte 205
 - Parameterübergabe per Z. 222
- Zeile
 - einlesen *siehe* getline()
 - neue 56
- Zeit-Server 570
- Zeitkomplexität 972
- Ziel (make) 603
- Ziffernzeichen 54
- Zufallszahlen 753
- Zugriffsspezifizierer und -rechte 286
- zusammengesetzte Datentypen 84
- Zusicherung 142, 338
- Zustand 972
 - eines Iterators 441
- Zuweisung 36, 66, 70, 972
 - und Initialisierung 172
 - und Vererbung 288, 396
- Zuweisungsoperator 172, 356, 467
 - implizite Deklaration 356, 396
 - und Vererbung 396
- zweidimensionale Matrix 744
- Zweierkomplement 43
- Zwischenraumzeichen 102, 212, 411