

# Die C++-Programmiersprache

Aktuell zu C++11

Bearbeitet von  
Bjarne Stroustrup, Frank Langenau

1. Auflage 2015. Buch. 1467 S.  
ISBN 978 3 446 43961 0  
Format (B x L): 21 x 24,5 cm  
Gewicht: 2477 g

[Weitere Fachgebiete > EDV, Informatik > Programmiersprachen: Methoden > Programmier- und Skriptsprachen](#)

schnell und portofrei erhältlich bei

  
DIE FACHBUCHHANDLUNG

Die Online-Fachbuchhandlung [beck-shop.de](http://beck-shop.de) ist spezialisiert auf Fachbücher, insbesondere Recht, Steuern und Wirtschaft. Im Sortiment finden Sie alle Medien (Bücher, Zeitschriften, CDs, eBooks, etc.) aller Verlage. Ergänzt wird das Programm durch Services wie Neuerscheinungsdienst oder Zusammenstellungen von Büchern zu Sonderpreisen. Der Shop führt mehr als 8 Millionen Produkte.



Leseprobe

Bjarne Stroustrup

Die C++-Programmiersprache

aktuell zum C++11-Standard

Übersetzt aus dem Englischen von Frank Langenau

ISBN (Buch): 978-3-446-43961-0

ISBN (E-Book): 978-3-446-43981-8

Weitere Informationen oder Bestellungen unter

<http://www.hanser-fachbuch.de/978-3-446-43961-0>

sowie im Buchhandel.

# Inhalt

<b>Vorwort</b> .....	<b>XXXI</b>
<b>Teil I – Einführung</b> .....	<b>1</b>
<b>1 Vorbemerkungen</b> .....	<b>3</b>
1.1 Zum Aufbau dieses Buchs .....	3
1.1.1 Einführung .....	4
1.1.2 Grundlegende Sprachmittel .....	4
1.1.3 Abstraktionsmechanismen .....	5
1.1.4 Die Standardbibliothek .....	7
1.1.5 Beispiele und Referenzen .....	8
1.2 Der Entwurf von C++ .....	10
1.2.1 Programmierstile .....	12
1.2.2 Typprüfung .....	15
1.2.3 C-Kompatibilität .....	16
1.2.4 Sprache, Bibliotheken und Systeme .....	17
1.3 C++ lernen .....	19
1.3.1 In C++ programmieren .....	21
1.3.2 Ratschläge für C++-Programmierer .....	22
1.3.3 Ratschläge für C-Programmierer .....	23
1.3.4 Ratschläge für Java-Programmierer .....	24
1.4 Historische Anmerkungen .....	25
1.4.1 Chronik .....	26
1.4.2 Die frühen Jahre .....	27
1.4.2.1 Sprachfeatures und Bibliotheksinstrumente .....	28
1.4.3 Der 1998-Standard .....	30
1.4.3.1 Sprachfeatures .....	30
1.4.3.2 Die Standardbibliothek .....	31
1.4.4 Der 2011-Standard .....	32
1.4.4.1 Sprachfeatures .....	33
1.4.4.2 Standardbibliothek .....	34
1.4.5 Wofür wird C++ verwendet? .....	35

1.5	Ratschläge .....	37
1.6	Literaturhinweise .....	38
<b>2</b>	<b>Ein Rundreise durch C++: Die Grundlagen .....</b>	<b>43</b>
2.1	Einführung .....	43
2.2	Die Grundlagen .....	44
2.2.1	Hello, World! .....	44
2.2.2	Typen, Variablen und Arithmetik .....	46
2.2.3	Konstanten .....	48
2.2.4	Tests und Schleifen .....	49
2.2.5	Zeiger, Arrays und Schleifen .....	51
2.3	Benutzerdefinierte Typen .....	53
2.3.1	Strukturen .....	53
2.3.2	Klassen .....	55
2.3.3	Aufzählungen .....	57
2.4	Modularität .....	58
2.4.1	Separate Kompilierung .....	59
2.4.2	Namespaces .....	60
2.4.3	Fehlerbehandlung .....	61
2.4.3.1	Ausnahmen .....	62
2.4.3.2	Invarianten .....	63
2.4.3.3	Statische Assertionen .....	64
2.5	Nachbemerkung .....	65
2.6	Ratschläge .....	65
<b>3</b>	<b>Eine Rundreise durch C++: Abstraktionsmechanismen .....</b>	<b>67</b>
3.1	Einführung .....	67
3.2	Klassen .....	68
3.2.1	Konkrete Typen .....	68
3.2.1.1	Ein arithmetischer Typ .....	69
3.2.1.2	Ein Container .....	71
3.2.1.3	Container initialisieren .....	72
3.2.2	Abstrakte Typen .....	73
3.2.3	Virtuelle Funktionen .....	76
3.2.4	Klassenhierarchien .....	77
3.3	Kopieren und verschieben .....	81
3.3.1	Container kopieren .....	81
3.3.2	Container verschieben .....	83
3.3.3	Ressourcenverwaltung .....	85
3.3.4	Operationen unterdrücken .....	86
3.4	Templates .....	87
3.4.1	Parametrisierte Typen .....	87
3.4.2	Funktions-Templates .....	88

3.4.3	Funktionsobjekte .....	89
3.4.4	Variadische Templates .....	92
3.4.5	Alias .....	93
3.5	Ratschläge .....	94
<b>4</b>	<b>Eine Rundreise durch C++: Container und Algorithmen .....</b>	<b>95</b>
4.1	Bibliotheken .....	95
4.1.1	Überblick über die Standardbibliothek .....	96
4.1.2	Header und Namespace der Standardbibliothek .....	97
4.2	Strings .....	98
4.3	Stream-Ein-/Ausgabe .....	100
4.3.1	Ausgabe .....	100
4.3.2	Eingabe .....	101
4.3.3	Ein-/Ausgabe von benutzerdefinierten Typen .....	102
4.4	Container .....	104
4.4.1	<b>vector</b> .....	104
4.4.1.1	Elemente .....	106
4.4.1.2	Bereichsüberprüfung .....	106
4.4.2	<b>list</b> .....	107
4.4.3	<b>map</b> .....	109
4.4.4	<b>unordered_map</b> .....	110
4.4.5	Überblick über Container .....	110
4.5	Algorithmen .....	112
4.5.1	Iteratoren verwenden .....	113
4.5.2	Iteratortypen .....	115
4.5.3	Stream-Iteratoren .....	116
4.5.4	Prädikate .....	118
4.5.5	Überblick über Algorithmen .....	118
4.5.6	Containeralgorithmen .....	119
4.6	Ratschläge .....	120
<b>5</b>	<b>Eine Rundreise durch C++: Nebenläufigkeit und Dienstprogramme .....</b>	<b>121</b>
5.1	Einführung .....	121
5.2	Ressourcenverwaltung .....	121
5.2.1	<b>unique_ptr</b> und <b>shared_ptr</b> .....	122
5.3	Nebenläufigkeit .....	124
5.3.1	Tasks und Threads .....	125
5.3.2	Argumente übergeben .....	126
5.3.3	Ergebnisse zurückgeben .....	127
5.3.4	Daten gemeinsam nutzen .....	127
5.3.4.1	Warten auf Ereignisse .....	129
5.3.5	Kommunizierende Tasks .....	130

5.3.5.1	<b>future</b> und <b>promise</b> .....	131
5.3.5.2	<b>packaged_task</b> .....	132
5.3.5.3	<b>async()</b> .....	133
5.4	Kleine Hilfskomponenten .....	134
5.4.1	Zeit .....	134
5.4.2	Typfunktionen .....	135
5.4.2.1	<b>iterator_traits</b> .....	135
5.4.2.2	Typprädikate .....	137
5.4.3	<b>pair</b> und <b>tuple</b> .....	138
5.5	Reguläre Ausdrücke .....	139
5.6	Mathematik .....	140
5.6.1	Mathematische Funktionen und Algorithmen .....	140
5.6.2	Komplexe Zahlen .....	140
5.6.3	Zufallszahlen .....	141
5.6.4	Vektorarithmetik .....	143
5.6.5	Numerische Grenzen .....	144
5.7	Ratschläge .....	144
<b>Teil II – Grundlegende Sprachelemente .....</b>		<b>145</b>
<b>6</b>	<b>Typen und Deklarationen .....</b>	<b>147</b>
6.1	Der ISO-C++-Standard .....	147
6.1.1	Implementierungen .....	149
6.1.2	Der grundlegende Quellzeichensatz .....	149
6.2	Typen .....	150
6.2.1	Fundamentale Typen .....	150
6.2.2	Boolesche Typen .....	151
6.2.3	Zeichentypen .....	153
6.2.3.1	Vorzeichenbehaftete und vorzeichenlose Zeichen .....	155
6.2.3.2	Zeichenlitterale .....	156
6.2.4	Ganzzahltypen .....	158
6.2.4.1	Ganzzahlilitterale .....	158
6.2.4.2	Typen von Ganzzahliliteralen .....	159
6.2.5	Gleitkommatypen .....	160
6.2.5.1	Gleitkommalitterale .....	160
6.2.6	Präfixe und Suffixe .....	161
6.2.7	<b>void</b> .....	162
6.2.8	Größen .....	162
6.2.9	Ausrichtung .....	165
6.3	Deklarationen .....	166
6.3.1	Die Struktur von Deklarationen .....	168
6.3.2	Mehrere Namen deklarieren .....	169
6.3.3	Namen .....	170
6.3.3.1	Schlüsselwörter .....	171

6.3.4	Gültigkeitsbereiche .....	172
6.3.5	Initialisierung .....	174
6.3.5.1	Fehlende Initialisierer .....	177
6.3.5.2	Initialisierungslisten .....	178
6.3.6	Einen Typ herleiten: <b>auto</b> und <b>decltype()</b> .....	179
6.3.6.1	Der Typspezifizierer <b>auto</b> .....	179
6.3.6.2	<b>auto</b> und <b>{}</b> -Listen .....	180
6.3.6.3	Der Spezifizierer <b>decltype()</b> .....	181
6.4	Objekte und Werte .....	182
6.4.1	L-Werte und R-Werte .....	182
6.4.2	Lebensdauer von Objekten .....	183
6.5	Typalias .....	184
6.6	Ratschläge .....	185
<b>7</b>	<b>Zeiger, Arrays und Referenzen .....</b>	<b>187</b>
7.1	Einführung .....	187
7.2	Zeiger .....	187
7.2.1	<b>void*</b> .....	188
7.2.2	<b>nullptr</b> .....	189
7.3	Arrays .....	190
7.3.1	Array-Initialisierer .....	191
7.3.2	Stringlitterale .....	192
7.3.2.1	Rohe Zeichen-Strings .....	194
7.3.2.2	Größere Zeichensätze .....	195
7.4	Zeiger auf Arrays .....	196
7.4.1	Navigieren in Arrays .....	198
7.4.2	Mehrdimensionale Arrays .....	200
7.4.3	Arrays übergeben .....	201
7.5	Zeiger und <b>const</b> .....	203
7.6	Zeiger und Besitz .....	205
7.7	Referenzen .....	206
7.7.1	L-Wert-Referenzen .....	208
7.7.2	R-Wert-Referenzen .....	211
7.7.3	Referenzen auf Referenzen .....	214
7.7.4	Zeiger und Referenzen .....	215
7.8	Ratschläge .....	217
<b>8</b>	<b>Strukturen, Unions und Aufzählungen .....</b>	<b>219</b>
8.1	Einführung .....	219
8.2	Strukturen .....	219
8.2.1	Layout einer Struktur .....	221
8.2.2	Namen von Strukturen .....	222
8.2.3	Strukturen und Klassen .....	224

8.2.4	Strukturen und Arrays	225
8.2.5	Typäquivalenz	227
8.2.6	Plain Old Data	228
8.2.7	Felder	230
8.3	Unions	231
8.3.1	Unions und Klassen	233
8.3.2	Anonyme Unions	234
8.4	Aufzählungen	237
8.4.1	Aufzählungsklassen	237
8.4.2	Einfache Aufzählungen	241
8.4.3	Unbenannte Aufzählungen	243
8.5	Ratschläge	243
<b>9</b>	<b>Anweisungen</b>	<b>245</b>
9.1	Einführung	245
9.2	Zusammenfassung der Anweisungen	245
9.3	Deklarationen als Anweisungen	247
9.4	Auswahanweisungen	248
9.4.1	<b>if</b> -Anweisungen	248
9.4.2	<b>switch</b> -Anweisungen	250
9.4.2.1	Deklarationen in <b>case</b> -Zweigen	252
9.4.3	Deklarationen in Bedingungen	252
9.5	Schleifenanweisungen	253
9.5.1	Bereichsbasierte <b>for</b> -Anweisungen	254
9.5.2	<b>for</b> -Anweisungen	255
9.5.3	<b>while</b> -Anweisungen	256
9.5.4	<b>do</b> -Anweisungen	257
9.5.5	Schleifen verlassen	257
9.6	<b>goto</b> -Anweisungen	258
9.7	Kommentare und Einrückungen	259
9.8	Ratschläge	261
<b>10</b>	<b>Ausdrücke</b>	<b>263</b>
10.1	Einführung	263
10.2	Ein Taschenrechner	263
10.2.1	Der Parser	264
10.2.2	Eingabe	268
10.2.3	Low-level-Eingabe	272
10.2.4	Fehlerbehandlung	274
10.2.5	Das Rahmenprogramm	274
10.2.6	Header	275
10.2.7	Befehlszeilenargumente	276
10.2.8	Eine Anmerkung zum Stil	277



10.3	Zusammenfassung der Operatoren .....	278
10.3.1	Ergebnisse .....	282
10.3.2	Reihenfolge der Auswertung .....	283
10.3.3	Operatorrangfolge .....	284
10.3.4	Temporäre Objekte .....	285
10.4	Konstante Ausdrücke .....	287
10.4.1	Symbolische Konstanten .....	289
10.4.2	<b>const</b> -Typen in konstanten Ausdrücken .....	290
10.4.3	Literale Typen .....	290
10.4.4	Referenzargumente .....	291
10.4.5	Adresse konstanter Ausdrücke .....	292
10.5	Implizite Typkonvertierung .....	292
10.5.1	Heraufstufungen .....	293
10.5.2	Konvertierungen .....	293
10.5.2.1	Integrale Konvertierungen .....	294
10.5.2.2	Gleitkommakonvertierungen .....	294
10.5.2.3	Zeiger- und Referenzkonvertierungen .....	295
10.5.2.4	Zeiger-auf-Member-Konvertierungen .....	295
10.5.2.5	Boolesche Konvertierungen .....	295
10.5.2.6	Gleitkomma-Ganzzahl-Konvertierungen .....	296
10.5.3	Übliche arithmetische Konvertierungen .....	297
10.6	Ratschläge .....	297
<b>11</b>	<b>Auswahloperationen .....</b>	<b>299</b>
11.1	Diverse Operatoren .....	299
11.1.1	Logische Operatoren .....	299
11.1.2	Bitweise logische Operatoren .....	299
11.1.3	Bedingte Ausdrücke .....	301
11.1.4	Inkrementieren und Dekrementieren .....	301
11.2	Freispeicher .....	303
11.2.1	Speicherverwaltung .....	305
11.2.2	Arrays .....	308
11.2.3	Speicherplatz beschaffen .....	309
11.2.4	Überladen von <b>new</b> .....	310
11.2.4.1	<b>nothrow new</b> .....	312
11.3	Listen .....	313
11.3.1	Implementierungsmodell .....	313
11.3.2	Qualifizierte Listen .....	315
11.3.3	Unqualifizierte Listen .....	315
11.4	Lambda-Ausdrücke .....	317
11.4.1	Implementierungsmodelle .....	318
11.4.2	Alternativen für Lambda-Ausdrücke .....	319
11.4.3	Erfassung .....	321
11.4.3.1	Lambda und Lebensdauer .....	323

11.4.3.2	Namen von Namespaces	324
11.4.3.3	Lambda und <b>this</b>	324
11.4.3.4	Veränderbare Lambda-Ausdrücke	324
11.4.4	Aufruf und Rückgabe	325
11.4.5	Der Typ eines Lambda-Ausdrucks	325
11.5	Explizite Typumwandlung	326
11.5.1	Konstruktion	328
11.5.2	Benannte Typumwandlungen	329
11.5.3	C-Typumwandlungen	331
11.5.4	Funktionale Typumwandlung	331
11.6	Ratschläge	332
<b>12</b>	<b>Funktionen</b>	<b>333</b>
12.1	Funktionsdeklarationen	333
12.1.1	Warum Funktionen?	334
12.1.2	Bestandteile einer Funktionsdeklaration	334
12.1.3	Funktionsdefinitionen	335
12.1.4	Werte zurückgeben	337
12.1.5	Inline-Funktionen	339
12.1.6	<b>constexpr</b> -Funktionen	340
12.1.6.1	<b>constexpr</b> und Referenzen	341
12.1.6.2	Bedingte Auswertung	342
12.1.7	<b>[[noreturn]]</b> -Funktionen	342
12.1.8	Lokale Variablen	343
12.2	Argumentübergabe	344
12.2.1	Referenzargumente	345
12.2.2	Array-Argumente	347
12.2.3	Listenargumente	349
12.2.4	Nicht angegebene Anzahl von Argumenten	350
12.2.5	Standardargumente	354
12.3	Überladene Funktionen	356
12.3.1	Automatische Auflösung von Überladungen	356
12.3.2	Überladen und Rückgabotyp	358
12.3.3	Überladen und Gültigkeitsbereiche	359
12.3.4	Auflösung für mehrere Argumente	360
12.3.5	Manuelle Auflösung von Überladungen	360
12.4	Vor- und Nachbedingungen	361
12.5	Zeiger auf Funktion	363
12.6	Makros	367
12.6.1	Bedingte Übersetzung	370
12.6.2	Vordefinierte Makros	371
12.6.3	Pragmas	372
12.7	Ratschläge	372

<b>13</b>	<b>Ausnahmenbehandlung</b>	<b>375</b>
13.1	Fehlerbehandlung	375
13.1.1	Ausnahmen	376
13.1.2	Herkömmliche Fehlerbehandlung	378
13.1.3	Durchhangeln	379
13.1.4	Alternative Ansichten von Ausnahmen	380
13.1.4.1	Asynchrone Ereignisse	380
13.1.4.2	Ausnahmen, die keine Fehler sind	380
13.1.5	Wann Sie keine Ausnahmen verwenden können	381
13.1.6	Hierarchische Fehlerbehandlung	382
13.1.7	Ausnahmen und Effizienz	384
13.2	Ausnahmegarantien	386
13.3	Ressourcenverwaltung	388
13.3.1	<b>finally</b>	391
13.4	Invarianten erzwingen	393
13.5	Ausnahmen auslösen und abfangen	398
13.5.1	Ausnahmen auslösen	398
13.5.1.1	<b>noexcept</b> -Funktionen	400
13.5.1.2	Der Operator <b>noexcept</b>	400
13.5.1.3	Ausnahmespezifikationen	401
13.5.2	Ausnahmen abfangen	402
13.5.2.1	Ausnahmen erneut auslösen	403
13.5.2.2	Jede Ausnahme abfangen	404
13.5.2.3	Mehrere Handler	405
13.5.2.4	<b>try</b> -Blöcke in Funktionen	405
13.5.2.5	Beendigung	407
13.5.3	Ausnahmen und Threads	409
13.6	Eine <b>vector</b> -Implementierung	410
13.6.1	Ein einfacher <b>vector</b>	410
13.6.2	Speicher explizit darstellen	414
13.6.3	Zuweisung	417
13.6.4	Größe ändern	419
13.6.4.1	<b>reserve()</b>	420
13.6.4.2	<b>resize()</b>	421
13.6.4.3	<b>push_back()</b>	421
13.6.4.4	Abschließende Gedanken	422
13.7	Ratschläge	423
<b>14</b>	<b>Namespaces</b>	<b>425</b>
14.1	Kompositionsprobleme	425
14.2	Namespaces	426
14.2.1	Explizite Qualifizierung	428
14.2.2	<b>using</b> -Deklarationen	429
14.2.3	<b>using</b> -Direktiven	430

14.2.4	Argumentabhängige Namensauflösung	431
14.2.5	Namespaces sind offen	434
14.3	Modularisierung und Schnittstellen	435
14.3.1	Namespaces als Module	436
14.3.2	Implementierungen	438
14.3.3	Schnittstellen und Implementierungen	440
14.4	Komposition mit Namespaces	442
14.4.1	Komfort vs. Sicherheit	442
14.4.2	Namespace-Alias	443
14.4.3	Namespaces zusammensetzen	444
14.4.4	Komposition und Auswahl	445
14.4.5	Namespaces und Überladen	446
14.4.6	Versionsverwaltung	448
14.4.7	Verschachtelte Namespaces	450
14.4.8	Unbenannte Namespaces	451
14.4.9	C-Header	452
14.5	Ratschläge	453
<b>15</b>	<b>Quelldateien und Programme</b>	<b>455</b>
15.1	Separate Übersetzung	455
15.2	Binden	456
15.2.1	Dateilokale Namen	459
15.2.2	Header-Dateien	460
15.2.3	Die Eine-Definition-Regel	462
15.2.4	Header der Standardbibliothek	464
15.2.5	Binden mit Nicht-C++-Code	465
15.2.6	Binden und Zeiger auf Funktionen	467
15.3	Header-Dateien verwenden	468
15.3.1	Organisation mit einzeltem Header	468
15.3.2	Organisation mit mehreren Header-Dateien	472
15.3.2.1	Andere Taschenrechnermodule	475
15.3.2.2	Header verwenden	477
15.3.3	Include-Wächter	478
15.4	Programme	479
15.4.1	Initialisierung von nichtlokalen Variablen	479
15.4.2	Initialisierung und Nebenläufigkeit	480
15.4.3	Programmbeendigung	481
15.5	Ratschläge	483
<b>Teil III</b>	<b>– Abstraktionsmechanismen</b>	<b>485</b>
<b>16</b>	<b>Klassen</b>	<b>487</b>
16.1	Einführung	487
16.2	Grundlagen von Klassen	488

16.2.1	Member-Funktionen .....	489
16.2.2	Standardmäßiges Kopieren .....	490
16.2.3	Zugriffskontrolle .....	491
16.2.4	<b>class</b> und <b>struct</b> .....	492
16.2.5	Konstruktoren .....	494
16.2.6	Explizite Konstruktoren .....	496
16.2.7	Klasseninterne Initialisierer .....	498
16.2.8	Klasseninterne Funktionsdefinitionen .....	499
16.2.9	Veränderlichkeit .....	500
16.2.9.1	Konstante Member-Funktionen .....	500
16.2.9.2	Physische und logische Konstanz .....	501
16.2.9.3	<b>mutable</b> .....	502
16.2.9.4	Veränderlichkeit über Indirektion .....	502
16.2.10	Selbstreferenz .....	503
16.2.11	Member-Zugriff .....	505
16.2.12	Statische Member .....	506
16.2.13	Member-Typen .....	508
16.3	Konkrete Klassen .....	509
16.3.1	Member-Funktionen .....	512
16.3.2	Hilfsfunktionen .....	515
16.3.3	Überladene Operatoren .....	516
16.3.4	Der Stellenwert von konkreten Klassen .....	517
16.4	Ratschläge .....	518
<b>17</b>	<b>Konstruieren, Aufräumen, Kopieren und Verschieben .....</b>	<b>521</b>
17.1	Einführung .....	521
17.2	Konstruktoren und Destruktoren .....	523
17.2.1	Konstruktoren und Invarianten .....	524
17.2.2	Destruktoren und Ressourcen .....	525
17.2.3	Basis- und Member-Destruktoren .....	526
17.2.4	Konstruktoren und Destruktoren aufrufen .....	527
17.2.5	Virtuelle Destruktoren .....	528
17.3	Initialisierung von Klassenobjekten .....	529
17.3.1	Initialisierung ohne Konstruktoren .....	529
17.3.2	Initialisierung mithilfe von Konstruktoren .....	531
17.3.2.1	Initialisierung durch Konstruktoren .....	533
17.3.3	Standardkonstruktoren .....	534
17.3.4	Initialisierungslisten-Konstruktoren .....	536
17.3.4.1	Mehrdeutigkeiten bei Initialisierungslisten-Konstruktoren auflösen .....	537
17.3.4.2	Initialisierungslisten verwenden .....	538
17.3.4.3	Direkte und Kopierinitialisierung .....	539
17.4	Initialisierung von Mitgliedern und Basisklassen .....	541
17.4.1	Member-Initialisierung .....	541
17.4.1.1	Member-Initialisierung und -Zuweisung .....	542

17.4.2	Basisklassen-Initialisierer	543
17.4.3	Konstruktoren delegieren	543
17.4.4	Klasseninterne Initialisierer	544
17.4.5	Initialisierer statischer Member	546
17.5	Kopieren und verschieben	547
17.5.1	Kopieren	548
17.5.1.1	Vorsicht vor Standardkonstruktoren	550
17.5.1.2	Kopieren von Basisklassen	550
17.5.1.3	Was Kopieren bedeutet	551
17.5.1.4	Slicing	554
17.5.2	Verschieben	555
17.6	Standardoperationen generieren	559
17.6.1	Explizite Standardoperationen	560
17.6.2	Standardoperationen	561
17.6.3	Standardoperationen verwenden	562
17.6.3.1	Standardkonstruktoren	562
17.6.3.2	Invarianten bewahren	562
17.6.3.3	Ressourceninvarianten	563
17.6.3.4	Partiell spezifizierte Invarianten	564
17.6.4	Gelöschte Funktionen	566
17.7	Ratschläge	568
<b>18</b>	<b>Überladen von Operatoren</b>	<b>571</b>
18.1	Einführung	571
18.2	Operatorfunktionen	573
18.2.1	Binäre und unäre Operatoren	574
18.2.2	Vordefinierte Bedeutungen für Operatoren	575
18.2.3	Operatoren und benutzerdefinierte Typen	576
18.2.4	Objekte übergeben	576
18.2.5	Operatoren in Namespaces	578
18.3	Ein Typ für komplexe Zahlen	580
18.3.1	Member- und Nicht-Member-Operatoren	580
18.3.2	Arithmetik mit gemischten Datentypen	581
18.3.3	Konvertierungen	582
18.3.3.1	Konvertierung von Operanden	584
18.3.4	Literale	585
18.3.5	Zugriffsfunktionen	586
18.3.6	Hilfsfunktionen	587
18.4	Typumwandlung	588
18.4.1	Konvertierungsoperatoren	588
18.4.2	Explizite Konvertierungsoperatoren	590
18.4.3	Mehrdeutigkeiten	591
18.5	Ratschläge	593

<b>19</b>	<b>Spezielle Operatoren</b>	<b>595</b>
19.1	Einführung	595
19.2	Spezielle Operatoren	595
19.2.1	Indizierung	595
19.2.2	Funktionsaufruf	596
19.2.3	Dereferenzieren	598
19.2.4	Inkrementieren und Dekrementieren	600
19.2.5	Allokation und Deallokation	602
19.2.6	Benutzerdefinierte Literale	604
19.3	Eine <b>String</b> -Klasse	607
19.3.1	Wesentliche Operationen	608
19.3.2	Zugriff auf Zeichen	608
19.3.3	Darstellung	609
19.3.3.1	Ergänzende Funktionen	611
19.3.4	Member-Funktionen	612
19.3.5	Hilfsfunktionen	615
19.3.6	Unsere <b>String</b> -Klasse verwenden	617
19.4	Friends	617
19.4.1	Friends finden	619
19.4.2	Friends und Member	620
19.5	Ratschläge	622
<b>20</b>	<b>Abgeleitete Klassen</b>	<b>625</b>
20.1	Einführung	625
20.2	Abgeleitete Klassen	626
20.2.1	Member-Funktionen	629
20.2.2	Konstruktoren und Destruktoren	630
20.3	Klassenhierarchien	631
20.3.1	Typfelder	631
20.3.2	Virtuelle Funktionen	634
20.3.3	Explizite Qualifizierung	637
20.3.4	Überschreiben steuern	637
20.3.4.1	<b>override</b>	639
20.3.4.2	<b>final</b>	640
20.3.5	Member der Basisklasse verwenden	642
20.3.5.1	Konstruktoren vererben	643
20.3.6	Lockerung bei Rückgabetypen	645
20.4	Abstrakte Klassen	647
20.5	Zugriffskontrolle	649
20.5.1	Geschützte Member	653
20.5.1.1	Geschützte Member verwenden	654
20.5.2	Zugriff auf Basisklassen	654
20.5.2.1	Mehrfachvererbung und Zugriffskontrolle	655
20.5.3	<b>using</b> -Deklarationen und Zugriffskontrolle	656

20.6	Zeiger auf Member .....	657
20.6.1	Zeiger auf Funktions-Member .....	657
20.6.2	Zeiger auf Daten-Member .....	659
20.6.3	Basis- und abgeleitete Member .....	660
20.7	Ratschläge .....	661
<b>21</b>	<b>Klassenhierarchien .....</b>	<b>663</b>
21.1	Einführung .....	663
21.2	Klassenhierarchien entwerfen .....	663
21.2.1	Implementierungsvererbung .....	664
21.2.1.1	Kritische Betrachtungen .....	667
21.2.2	Schnittstellenvererbung .....	668
21.2.3	Alternative Implementierungen .....	670
21.2.3.1	Kritische Betrachtungen .....	673
21.2.4	Objekterstellung örtlich begrenzen .....	674
21.3	Mehrfachvererbung .....	675
21.3.1	Mehrfachschnittstellen .....	676
21.3.2	Mehrere Implementierungsklassen .....	676
21.3.3	Auflösung von Mehrdeutigkeiten .....	678
21.3.4	Eine Basisklasse wiederholt verwenden .....	682
21.3.5	Virtuelle Basisklassen .....	683
21.3.5.1	Virtuelle Basisklassen konstruieren .....	686
21.3.5.2	Member einer virtuellen Klasse nur einmal aufrufen .....	687
21.3.6	Replizierte und virtuelle Basisklassen .....	689
21.3.6.1	Funktionen virtueller Basisklassen überschreiben .....	691
21.4	Ratschläge .....	692
<b>22</b>	<b>Laufzeit-Typinformationen .....</b>	<b>693</b>
22.1	Einführung .....	693
22.2	Navigation in der Klassenhierarchie .....	693
22.2.1	<b>dynamic_cast</b> .....	695
22.2.1.1	<b>dynamic_cast</b> in Referenz .....	697
22.2.2	Mehrfachvererbung .....	698
22.2.3	<b>static_cast</b> und <b>dynamic_cast</b> .....	700
22.2.4	Eine Schnittstelle zurückgewinnen .....	701
22.3	Doppelte Bindung und Besucher .....	705
22.3.1	Doppelte Bindung .....	706
22.3.2	Besucher .....	708
22.4	Konstruktion und Destruktion .....	710
22.5	Typidentifizierung .....	711
22.5.1	Erweiterte Typinformationen .....	713
22.6	RTTI – richtig und falsch eingesetzt .....	714
22.7	Ratschläge .....	716



<b>23</b>	<b>Templates</b>	<b>719</b>
23.1	Einführung und Überblick	719
23.2	Ein einfaches String-Template	722
23.2.1	Ein Template definieren	724
23.2.2	Template-Instanziierung	725
23.3	Typprüfung	726
23.3.1	Typäquivalenz	728
23.3.2	Fehlererkennung	729
23.4	Member von Klassen-Templates	730
23.4.1	Daten-Member	730
23.4.2	Member-Funktionen	731
23.4.3	Member-Typalias	731
23.4.4	Statische Member	732
23.4.5	Member-Typen	732
23.4.6	Member-Templates	733
23.4.6.1	Templates und Konstruktoren	734
23.4.6.2	Templates und <b>virtual</b>	735
23.4.6.3	Verschachtelungen	735
23.4.7	Friends	738
23.5	Funktions-Templates	739
23.5.1	Argumente von Funktions-Templates	741
23.5.2	Funktions-Template-Argumente herleiten	742
23.5.2.1	Referenzherleitung	743
23.5.3	Überladen von Funktions-Templates	745
23.5.3.1	Mehrdeutigkeiten auflösen	746
23.5.3.2	Fehler bei Argumentersetzung	747
23.5.3.3	Überladen und Ableitung	749
23.5.3.4	Überladen und nicht hergeleitete Parameter	749
23.6	Template-Alias	750
23.7	Quellcodeorganisation	751
23.7.1	Binden	753
23.8	Ratschläge	754
<b>24</b>	<b>Generische Programmierung</b>	<b>757</b>
24.1	Einführung	757
24.2	Algorithmen und Lifting	758
24.3	Konzepte	762
24.3.1	Ein Konzept erkennen	763
24.3.2	Konzepte und Einschränkungen	766
24.4	Konzepte konkret machen	768
24.4.1	Axiome	772
24.4.2	Konzepte mit mehreren Argumenten	773
24.4.3	Wertkonzepte	775

24.4.4	Einschränkungsüberprüfungen .....	775
24.4.5	Überprüfung von Template-Definitionen .....	777
24.5	Ratschläge .....	779
<b>25</b>	<b>Spezialisierung .....</b>	<b>781</b>
25.1	Einführung .....	781
25.2	Template-Parameter und -Argumente .....	782
25.2.1	Typen als Argumente .....	782
25.2.2	Werte als Argumente .....	784
25.2.3	Operationen als Argumente .....	786
25.2.4	Templates als Argumente .....	788
25.2.5	Template-Standardargumente .....	789
25.2.5.1	Standardargumente bei Funktions-Templates .....	790
25.3	Spezialisierung .....	791
25.3.1	Schnittstellenspezialisierung .....	794
25.3.1.1	Implementierungsspezialisierung .....	795
25.3.2	Das primäre Template .....	795
25.3.3	Reihenfolge der Spezialisierung .....	797
25.3.4	Funktions-Template-Spezialisierung .....	798
25.3.4.1	Spezialisierung und Überladen .....	798
25.3.4.2	Spezialisierung, die kein Überladen ist .....	799
25.4	Ratschläge .....	800
<b>26</b>	<b>Instanziierung .....</b>	<b>801</b>
26.1	Einführung .....	801
26.2	Template-Instanziierung .....	802
26.2.1	Wann wird Instanziierung gebraucht? .....	803
26.2.2	Manuelle Kontrolle der Instanziierung .....	804
26.3	Namensbindung .....	805
26.3.1	Abhängige Namen .....	807
26.3.2	Bindung am Punkt der Definition .....	809
26.3.3	Bindung am Punkt der Instanziierung .....	810
26.3.4	Mehrere Punkte der Instanziierung .....	812
26.3.5	Templates und Namespaces .....	814
26.3.6	Zu aggressive ADL .....	815
26.3.7	Namen aus Basisklassen .....	817
26.4	Ratschläge .....	819
<b>27</b>	<b>Templates und Hierarchien .....</b>	<b>821</b>
27.1	Einführung .....	821
27.2	Parametrisierung und Hierarchie .....	822
27.2.1	Generierte Typen .....	824
27.2.2	Template-Konvertierungen .....	826

27.3	Hierarchien von Klassen-Templates .....	827
27.3.1	Templates als Schnittstellen .....	829
27.4	Template-Parameter als Basisklassen .....	829
27.4.1	Datenstrukturen zusammensetzen .....	830
27.4.2	Klassenhierarchien linearisieren .....	834
27.5	Ratschläge .....	839
<b>28</b>	<b>Metaprogrammierung .....</b>	<b>841</b>
28.1	Einführung .....	841
28.2	Typfunktionen .....	843
28.2.1	Typalias .....	846
28.2.1.1	Wann man keinen Alias verwendet .....	847
28.2.2	Typprädikate .....	848
28.2.3	Eine Funktion auswählen .....	850
28.2.4	Traits .....	850
28.3	Steuerungsstrukturen .....	852
28.3.1	Auswahl .....	852
28.3.1.1	Zwischen zwei Typen auswählen .....	853
28.3.1.2	Übersetzungszeit versus Laufzeit .....	853
28.3.1.3	Zwischen mehreren Typen auswählen .....	855
28.3.2	Iteration und Rekursion .....	856
28.3.2.1	Rekursion mit Klassen .....	857
28.3.3	Wann man Metaprogrammierung verwendet .....	857
28.4	Bedingte Definition: <b>Enable_if</b> .....	859
28.4.1	<b>Enable_if</b> verwenden .....	860
28.4.2	<b>Enable_if</b> implementieren .....	862
28.4.3	<b>Enable_if</b> und Konzepte .....	863
28.4.4	Weitere Beispiele mit <b>Enable_if</b> .....	863
28.5	Eine Liste zur Übersetzungszeit: <b>Tuple</b> .....	866
28.5.1	Eine einfache Ausgabefunktion .....	868
28.5.2	Elementzugriff .....	869
28.5.2.1	Konstante Tupel .....	871
28.5.3	<b>make_tuple</b> .....	872
28.6	Variadische Templates .....	873
28.6.1	Eine typsichere <b>printf()</b> -Funktion .....	873
28.6.2	Technische Details .....	876
28.6.3	Weiterleitung .....	878
28.6.4	Der Typ <b>tuple</b> der Standardbibliothek .....	879
28.7	Beispiel: SI-Einheiten .....	883
28.7.1	Einheiten .....	883
28.7.2	Größen .....	884
28.7.3	Literale für Einheiten .....	886
28.7.4	Hilfsfunktionen .....	888
28.8	Ratschläge .....	889

<b>29</b>	<b>Ein Matrix-Design</b>	<b>891</b>
29.1	Einführung	891
29.1.1	Einfache Anwendungen von <b>Matrix</b>	891
29.1.2	Anforderungen an <b>Matrix</b>	893
29.2	Ein <b>Matrix</b> -Template	894
29.2.1	Konstruktion und Zuweisung	896
29.2.2	Indizierung und Slicing	897
29.3	Arithmetische <b>Matrix</b> -Operationen	900
29.3.1	Skalaroperationen	901
29.3.2	Addition	901
29.3.3	Multiplikation	903
29.4	<b>Matrix</b> -Implementierung	905
29.4.1	<b>slice()</b>	905
29.4.2	<b>Matrix</b> -Slices	905
29.4.3	<b>Matrix_ref</b>	907
29.4.4	<b>Matrix</b> -Listeninitialisierung	908
29.4.5	<b>Matrix</b> -Zugriff	910
29.4.6	Nulldimensionale <b>Matrix</b>	913
29.5	Lineare Gleichungen lösen	914
29.5.1	Das klassische gaußsche Eliminationsverfahren	915
29.5.2	Pivotisierung	916
29.5.3	Testen	917
29.5.4	Verschmolzene Operationen	918
29.6	Ratschläge	921
<b>Teil IV – Die Standardbibliothek</b>		<b>923</b>
<b>30</b>	<b>Überblick über die Standardbibliothek</b>	<b>925</b>
30.1	Einführung	925
30.1.1	Komponenten der Standardbibliothek	926
30.1.2	Design einschränkungen	927
30.1.3	Beschreibungsstil	929
30.2	Header	929
30.3	Sprachunterstützung	934
30.3.1	Unterstützung für Initialisierungslisten	934
30.3.2	Unterstützung für bereichsbasierte <b>for</b> -Anweisung	935
30.4	Fehlerbehandlung	935
30.4.1	Ausnahmen	936
30.4.1.1	Die Hierarchie der Standardausnahmen	937
30.4.1.2	Weiterleitung von Ausnahmen	938
30.4.1.3	<b>terminate()</b>	941
30.4.2	Assertionen	941
30.4.3	<b>system_error</b>	942
30.4.3.1	Fehlercodes	943

30.4.3.2	Fehlerkategorien .....	945
30.4.3.3	<b>system_error</b> -Ausnahme .....	946
30.4.3.4	Potenziell portable Fehlerbedingungen .....	947
30.4.3.5	Fehlercodes zuordnen .....	947
30.4.3.6	<b>errc</b> -Fehlercodes .....	949
30.4.3.7	<b>future_errc</b> -Fehlercodes .....	952
30.4.3.8	<b>io_errc</b> -Fehlercodes .....	952
30.5	Ratschläge .....	952
<b>31</b>	<b>STL-Container .....</b>	<b>955</b>
31.1	Einführung .....	955
31.2	Überblick über Container .....	955
31.2.1	Containerdarstellung .....	958
31.2.2	Anforderungen an die Elemente .....	960
31.2.2.1	Vergleiche .....	961
31.2.2.2	Andere relationale Operatoren .....	962
31.3	Operatoren im Überblick .....	963
31.3.1	Member-Typen .....	966
31.3.2	Konstruktoren, Destruktor und Zuweisungen .....	967
31.3.3	Größe und Kapazität .....	969
31.3.4	Iteratoren .....	969
31.3.5	Elementzugriff .....	971
31.3.6	Stack-Operationen .....	971
31.3.7	Listenoperationen .....	972
31.3.8	Andere Operationen .....	973
31.4	Container .....	974
31.4.1	<b>vector</b> .....	974
31.4.1.1	<b>vector</b> und Wachstum .....	974
31.4.1.2	<b>vector</b> und Verschachtelung .....	976
31.4.1.3	<b>vector</b> und Arrays .....	978
31.4.1.4	<b>vector</b> und <b>string</b> .....	978
31.4.2	Listen .....	979
31.4.3	Assoziative Container .....	981
31.4.3.1	Geordnete assoziative Container .....	982
31.4.3.2	Ungeordnete assoziative Container .....	986
31.4.3.3	Ungeordnete Maps konstruieren .....	987
31.4.3.4	Hash- und Gleichheitsfunktionen .....	989
31.4.3.5	Lastfaktor und Buckets .....	992
31.5	Containeradapter .....	993
31.5.1	<b>stack</b> .....	994
31.5.2	<b>queue</b> .....	996
31.5.3	<b>priority_queue</b> .....	996
31.6	Ratschläge .....	997

<b>32 STL-Algorithmen</b> .....	<b>1001</b>
32.1 Einführung .....	1001
32.2 Algorithmen .....	1001
32.2.1 Sequenzen .....	1002
32.3 Richtlinienargumente .....	1003
32.3.1 Komplexität .....	1005
32.4 Nichtmodifizierende Sequenzalgorithmen .....	1006
32.4.1 <b>for_each()</b> .....	1006
32.4.2 Sequenzprädikate .....	1006
32.4.3 <b>count()</b> .....	1007
32.4.4 <b>find()</b> .....	1007
32.4.5 <b>equal()</b> und <b>mismatch()</b> .....	1008
32.4.6 <b>search()</b> .....	1009
32.5 Modifizierende Sequenzalgorithmen .....	1010
32.5.1 <b>copy()</b> .....	1011
32.5.2 <b>unique()</b> .....	1012
32.5.3 <b>remove()</b> , <b>reverse()</b> und <b>replace()</b> .....	1013
32.5.4 <b>rotate()</b> , <b>random_shuffle()</b> und <b>partition()</b> .....	1014
32.5.5 Permutationen .....	1015
32.5.6 <b>fill()</b> .....	1016
32.5.7 <b>swap()</b> .....	1017
32.6 Sortieren und Suchen .....	1018
32.6.1 Binäre Suche .....	1021
32.6.2 <b>merge()</b> .....	1022
32.6.3 Mengenalgorithmen .....	1023
32.6.4 Heaps .....	1024
32.6.5 <b>lexicographical_compare()</b> .....	1026
32.7 Minimum und Maximum .....	1026
32.8 Ratschläge .....	1028
<b>33 STL-Iteratoren</b> .....	<b>1029</b>
33.1 Einführung .....	1029
33.1.1 Iteratormodell .....	1029
33.1.2 Iteratorkategorien .....	1031
33.1.3 Iterator-Traits .....	1032
33.1.4 Iteratoroperationen .....	1035
33.2 Iteratoradapter .....	1036
33.2.1 Reverse-Iteratoren .....	1036
33.2.2 Einfügeiteratoren .....	1039
33.2.3 Verschiebeiteratoren .....	1040
33.3 Bereichszugriffsfunktionen .....	1041
33.4 Funktionsobjekte .....	1042
33.5 Funktionsadapter .....	1043

33.5.1	<b>bind()</b>	1044
33.5.2	<b>mem_fn()</b>	1046
33.5.3	<b>function</b>	1046
33.6	Ratschläge	1049
<b>34</b>	<b>Speicher und Ressourcen</b>	<b>1051</b>
34.1	Einführung	1051
34.2	„Beinahe-Container“	1051
34.2.1	<b>array</b>	1052
34.2.2	<b>bitset</b>	1055
34.2.2.1	Konstruktoren	1056
34.2.2.2	<b>bitset</b> -Operationen	1058
34.2.3	<b>vector&lt;bool&gt;</b>	1060
34.2.4	Tupel	1061
34.2.4.1	<b>pair</b>	1061
34.2.4.2	<b>tuple</b>	1063
34.3	Ressourcenverwaltungszeiger	1065
34.3.1	<b>unique_ptr</b>	1066
34.3.2	<b>shared_ptr</b>	1069
34.3.3	<b>weak_ptr</b>	1073
34.4	Allokatoren	1076
34.4.1	Der Standard-Allokator	1077
34.4.2	Allokator-Traits	1079
34.4.3	Zeiger-Traits	1080
34.4.4	Allokatoren mit eigenem Gültigkeitsbereich	1080
34.5	Die Schnittstelle zur Garbage Collection	1082
34.6	Nichtinitialisierter Speicher	1085
34.6.1	Temporäre Puffer	1086
34.6.2	<b>raw_storage_iterator</b>	1086
34.7	Ratschläge	1088
<b>35</b>	<b>Utilities</b>	<b>1089</b>
35.1	Einführung	1089
35.2	Zeit	1089
35.2.1	<b>duration</b>	1090
35.2.2	<b>time_point</b>	1093
35.2.3	Zeitgeber	1095
35.2.4	Zeit-Traits	1096
35.3	Rationale Arithmetik zur Übersetzungszeit	1097
35.4	Typfunktionen	1099
35.4.1	Typ-Traits	1099
35.4.2	Typgeneratoren	1104
35.5	Kleinere Utilities	1109

35.5.1	<b>move()</b> and <b>forward()</b> .....	1109
35.5.2	<b>swap()</b> .....	1110
35.5.3	Relationale Operatoren .....	1111
35.5.4	Vergleichen und Hashing von <b>type_info</b> .....	1112
35.6	Ratschläge .....	1113
<b>36</b>	<b>Strings</b> .....	<b>1115</b>
36.1	Einführung .....	1115
36.2	Zeichenklassifizierung .....	1115
36.2.1	Klassifizierungsfunktionen .....	1115
36.2.2	Zeichen-Traits .....	1117
36.3	Strings .....	1118
36.3.1	<b>string</b> im Vergleich zu C-Strings .....	1119
36.3.2	Konstruktoren .....	1121
36.3.3	Fundamentale Operationen .....	1123
36.3.4	String-Ein-/Ausgabe .....	1125
36.3.5	Numerische Konvertierungen .....	1125
36.3.6	STL-ähnliche Operationen .....	1127
36.3.7	Die <b>find</b> -Familie .....	1130
36.3.8	Teilstrings .....	1132
36.4	Ratschläge .....	1133
<b>37</b>	<b>Reguläre Ausdrücke</b> .....	<b>1135</b>
37.1	Reguläre Ausdrücke .....	1135
37.1.1	Notation regulärer Ausdrücke .....	1136
37.2	<b>regex</b> .....	1141
37.2.1	Übereinstimmungsergebnisse .....	1143
37.2.2	Formatierung .....	1146
37.3	Funktionen für reguläre Ausdrücke .....	1147
37.3.1	<b>regex_match()</b> .....	1147
37.3.2	<b>regex_search()</b> .....	1149
37.3.3	<b>regex_replace()</b> .....	1150
37.4	Iteratoren für reguläre Ausdrücke .....	1152
37.4.1	<b>regex_iterator</b> .....	1152
37.4.2	<b>regex_token_iterator</b> .....	1153
37.5	<b>regex_traits</b> .....	1156
37.6	Ratschläge .....	1157
<b>38</b>	<b>E/A-Streams</b> .....	<b>1159</b>
38.1	Einführung .....	1159
38.2	Die E/A-Stream-Hierarchie .....	1161
38.2.1	Datei-Streams .....	1162
38.2.2	String-Streams .....	1164



38.3	Fehlerbehandlung .....	1166
38.4	Ein-/Ausgabeoperationen .....	1168
38.4.1	Eingabeoperationen .....	1168
38.4.1.1	Formatierte Eingabe .....	1169
38.4.1.2	Unformatierte Eingabe .....	1170
38.4.2	Ausgabeoperationen .....	1171
38.4.2.1	Virtuelle Ausgabefunktionen .....	1173
38.4.3	Manipulatoren .....	1174
38.4.4	Stream-Status .....	1175
38.4.5	Formatierung .....	1179
38.4.5.1	Formatierungsstatus .....	1180
38.4.5.2	Standardmanipulatoren .....	1182
38.4.5.3	Benutzerdefinierte Manipulatoren .....	1185
38.5	Stream-Iteratoren .....	1187
38.6	Puffer .....	1188
38.6.1	Ausgabe-Streams und -puffer .....	1192
38.6.2	Eingabe-Streams und -puffer .....	1193
38.6.3	Pufferiteratoren .....	1194
38.6.3.1	<b>istreambuf_iterator</b> .....	1195
38.6.3.2	<b>ostreambuf_iterator</b> .....	1196
38.7	Ratschläge .....	1196
<b>39</b>	<b>Locales .....</b>	<b>1199</b>
39.1	Kulturelle Unterschiede behandeln .....	1199
39.2	Die Klasse <b>locale</b> .....	1202
39.2.1	Benannte Locales .....	1204
39.2.1.1	Neue Locales konstruieren .....	1207
39.2.2	Strings vergleichen .....	1208
39.3	Die Klasse <b>facet</b> .....	1209
39.3.1	Auf Facetten in einem Locale zugreifen .....	1210
39.3.2	Eine einfache benutzerdefinierte Facette .....	1211
39.3.3	Locales und Facetten verwenden .....	1214
39.4	Standardfacetten .....	1215
39.4.1	String-Vergleich .....	1217
39.4.1.1	Benannte <b>collate</b> -Facetten .....	1220
39.4.2	Numerische Formatierung .....	1220
39.4.2.1	Numerische Interpunktion .....	1221
39.4.2.2	Numerische Ausgabe .....	1222
39.4.2.3	Numerische Eingabe .....	1225
39.4.3	Formatierung von Geldbeträgen .....	1226
39.4.3.1	Interpunktion bei Geldbeträgen .....	1227
39.4.3.2	Ausgabe von Geldbeträgen .....	1230
39.4.3.3	Eingabe von Geldbeträgen .....	1231

39.4.4	Datum und Uhrzeit formatieren .....	1232
39.4.4.1	<b>time_put</b> .....	1232
39.4.4.2	<b>time_get</b> .....	1233
39.4.5	Zeichenklassifizierung .....	1235
39.4.6	Zeichencodes konvertieren .....	1239
39.4.7	Meldungen .....	1243
39.4.7.1	Meldungen von anderen Facetten verwenden .....	1246
39.5	Komfortschnittstellen .....	1247
39.5.1	Zeichenklassifizierung .....	1247
39.5.2	Zeichenkonvertierungen .....	1248
39.5.3	String-Konvertierungen .....	1248
39.5.4	Pufferkonvertierungen .....	1250
39.6	Ratschläge .....	1250
<b>40</b>	<b>Numerische Berechnungen .....</b>	<b>1253</b>
40.1	Einführung .....	1253
40.2	Numerische Grenzen .....	1253
40.2.1	Makros für Grenzwerte .....	1256
40.3	Mathematische Standardfunktionen .....	1257
40.4	Komplexe Zahlen .....	1259
40.5	Ein numerisches Array: <b>valarray</b> .....	1260
40.5.1	Konstruktoren und Zuweisungen .....	1261
40.5.2	Indizierung .....	1263
40.5.3	Operationen .....	1264
40.5.4	Slices .....	1267
40.5.5	<b>slice_array</b> .....	1269
40.5.6	Verallgemeinerte Slices .....	1270
40.6	Verallgemeinerte numerische Algorithmen .....	1271
40.6.1	<b>accumulate()</b> .....	1272
40.6.2	<b>inner_product()</b> .....	1273
40.6.3	<b>partial_sum()</b> und <b>adjacent_difference()</b> .....	1274
40.6.4	<b>iota()</b> .....	1275
40.7	Zufallszahlen .....	1275
40.7.1	Zufallszahlenmodule .....	1278
40.7.2	Zufallsgerät .....	1280
40.7.3	Verteilungen .....	1281
40.7.4	Zufallszahlen im Stil von C .....	1285
40.8	Ratschläge .....	1286
<b>41</b>	<b>Nebenläufigkeit .....</b>	<b>1287</b>
41.1	Einführung .....	1287
41.2	Speichermodelle .....	1289
41.2.1	Speicherstellen .....	1290

41.2.2	Umordnung von Befehlen .....	1291
41.2.3	Speicherordnung .....	1292
41.2.4	Data Races .....	1293
41.3	Atomare Datentypen .....	1295
41.3.1	Atomare Typen .....	1298
41.3.2	Flags und Fences .....	1303
41.3.3	Atomare Flags .....	1303
41.3.3.1	Fences .....	1304
41.4	<b>volatile</b> .....	1304
41.5	Ratschläge .....	1305
<b>42</b>	<b>Threads und Tasks .....</b>	<b>1307</b>
42.1	Einführung .....	1307
42.2	Threads .....	1307
42.2.1	Identität .....	1309
42.2.2	Konstruktion .....	1310
42.2.3	Zerstörung .....	1311
42.2.4	<b>join()</b> .....	1312
42.2.5	<b>detach()</b> .....	1313
42.2.6	Namespace <b>this_thread</b> .....	1315
42.2.7	Einen Thread vorzeitig beenden .....	1316
42.2.8	<b>thread_local</b> -Daten .....	1316
42.3	Data Races vermeiden .....	1318
42.3.1	Mutexe .....	1319
42.3.1.1	<b>mutex</b> und <b>recursive_mutex</b> .....	1320
42.3.1.2	<b>mutex</b> -Fehler .....	1322
42.3.1.3	<b>timed_mutex</b> und <b>recursive_timed_mutex</b> .....	1323
42.3.1.4	<b>lock_guard</b> und <b>unique_lock</b> .....	1324
42.3.2	Mehrere Sperren .....	1328
42.3.3	<b>call_once()</b> .....	1329
42.3.4	Bedingungsvariablen .....	1330
42.3.4.1	<b>condition_variable_any</b> .....	1335
42.4	Task-basierte Nebenläufigkeit .....	1336
42.4.1	<b>future</b> und <b>promise</b> .....	1337
42.4.2	<b>promise</b> .....	1338
42.4.3	<b>packaged_task</b> .....	1339
42.4.4	<b>future</b> .....	1342
42.4.5	<b>shared_future</b> .....	1345
42.4.6	<b>async()</b> .....	1346
42.4.7	Ein paralleles <b>find()</b> -Beispiel .....	1349
42.5	Ratschläge .....	1353

<b>43 Die C-Standardbibliothek</b> .....	<b>1355</b>
43.1 Einführung .....	1355
43.2 Dateien .....	1355
43.3 Die <b>printf()</b> -Familie .....	1356
43.4 C-Strings .....	1361
43.5 Speicher .....	1362
43.6 Datum und Uhrzeit .....	1364
43.7 Diverses .....	1367
43.8 Ratschläge .....	1369
<b>44 Kompatibilität</b> .....	<b>1371</b>
44.1 Einführung .....	1371
44.2 C++11-Erweiterungen .....	1372
44.2.1 Sprachfeatures .....	1372
44.2.2 Komponenten der Standardbibliothek .....	1373
44.2.3 Veraltete Features .....	1374
44.2.4 Umgang mit älteren C++-Implementierungen .....	1375
44.3 C/C++-Kompatibilität .....	1376
44.3.1 C und C++ sind Geschwister .....	1376
44.3.2 „Stillschweigende“ Unterschiede .....	1378
44.3.3 C-Code, der kein C++ ist .....	1379
44.3.3.1 Probleme mit „klassischem C“ .....	1382
44.3.3.2 C-Features, die von C++ nicht übernommen wurden .....	1382
44.3.4 C++-Code, der kein C ist .....	1382
44.4 Ratschläge .....	1384
<b>Index</b> .....	<b>1387</b>

# Vorwort

*All problems in computer science  
can be solved by another level of indirection,  
except for the problem of too many layers of indirection.*

*- David J. Wheeler*

C++ fühlt sich wie eine neue Sprache an. Das heißt, ich kann in C++11 meine Ideen klarer, einfacher und direkter ausdrücken als ich es in C++98 konnte. Darüber hinaus werden die Programme durch den Compiler besser überprüft und laufen auch schneller.

In diesem Buch strebe ich nach *Vollständigkeit*. Ich beschreibe alle Sprachfeatures und Komponenten der Standardbibliothek, die ein professioneller Programmierer wahrscheinlich braucht. Für jedes dieser Elemente gebe ich Folgendes an:

- *Grundprinzip:* Welche Probleme soll es lösen helfen? Welche Prinzipien liegen dem Design zugrunde? Was sind die grundsätzlichen Beschränkungen?
- *Spezifikation:* Wie ist es definiert? Dabei ist die Detailebene auf den erfahrenen Programmierer ausgerichtet; der angehende Sprachanwalt kann den vielen Verweisen zum ISO-Standard folgen.
- *Beispiele:* Wie kann man es an sich und in Kombination mit anderen Features sinnvoll einsetzen? Wie sehen die Schlüsseltechniken und Idioms aus? Welche Bedeutung hat es für Wartbarkeit und Performance?

Die Verwendung von C++ hat sich im Lauf der Jahre drastisch gewandelt und das Gleiche gilt auch für die Sprache selbst. Aus dem Blickwinkel eines Programmierers sind die meisten Änderungen als Verbesserungen einzustufen. Das aktuelle ISO-Standard-C++ (ISO/IEC 14882-2011, üblicherweise C++11 genannt) ist ein weit besseres Tool für Qualitätssoftware als die vorherigen Versionen. Worin äußert sich dies? Welche Programmierstile und -techniken unterstützt modernes C++? Welche Features der Sprache und der Standardbibliothek unterstützen diese Techniken? Was sind die Bausteine von elegantem, korrektem, wartbarem und effizientem C++-Code? Dies sind die Schlüsselfragen, die in diesem Buch beantwortet werden. Viele Antworten sind nicht die gleichen, wie Sie sie für Vintage-C++ der Jahre 1985, 1995 oder 2005 finden: Der Fortschritt ist nicht aufzuhalten.

C++ ist eine universelle Programmiersprache, die den Entwurf und die Verwendung von typereichen, kompakten Abstraktionen betont. Die Sprache ist besonders für ressourcenbeschränkte Anwendungen geeignet, wie man sie beispielsweise in Softwareinfrastrukturen findet. Es lohnt sich für den Programmierer, Zeit in das Erlernen der Techniken für qualitätsgerechten Code zu investieren. C++ ist eine Sprache für jemanden, der Programmierung ernst nimmt. Unsere Zivilisation hängt entscheidend von Software ab; es wäre also besser, wenn es sich dabei um Qualitätssoftware handelt.

Es gibt Milliarden von Zeilen in C++-Code. Dabei wird besonderer Wert auf Stabilität gelegt, sodass der C++-Code von 1985 und 1995 immer noch funktioniert und auch für weitere Jahrzehnte funktionieren wird. Allerdings können Sie mit modernem C++ besser arbeiten; wenn Sie sich an die älteren Stile klammern, werden Sie Code von geringerer Qualität und schlechterer Leistung schreiben. Die Betonung von Stabilität bedeutet auch, dass standard-konformer Code, den Sie heute schreiben, auch noch in einigen Jahrzehnten funktionieren wird. Der gesamte Code in diesem Buch ist zum 2011-ISO-C++-Standard konform.

Dieses Buch richtet sich an drei Leserkreise:

- C++-Programmierer, die wissen möchten, was der neueste ISO-C++-Standard zu bieten hat,
- C-Programmierer, die daran interessiert sind, was C++ über C hinaus bietet, und
- Programmierer, die von Anwendungssprachen wie zum Beispiel Java, C#, Python und Ruby kommen und nach etwas suchen, das „näher an der Maschine“ dran ist – etwas, was flexibler ist, was eine bessere Prüfung zur Übersetzungszeit realisiert oder was eine bessere Performance bietet.

Natürlich gibt es keine klare Trennung zwischen diesen drei Gruppen – ein professioneller Softwareentwickler beherrscht schließlich mehr als nur eine Programmiersprache.

In diesem Buch wird davon ausgegangen, dass die Leser Programmierer sind. Wenn Sie fragen: „Was ist eine **for**-Schleife?“ oder „Was ist ein Compiler?“, dann ist dieses Buch (noch) nichts für Sie; stattdessen empfehle ich mein „*Principles and Practice Using C++ to get started with programming and C++*“. Darüber hinaus nehme ich an, dass die Leser bereits eine gewisse Reife als Softwareentwickler haben. Wenn Sie fragen: „Warum sich mit Testen abmühen?“ oder sagen „Alle Sprachen sind prinzipiell gleich; zeige mir nur die Syntax“, vielleicht auch überzeugt davon sind, dass es genau eine Sprache gibt, die für sämtliche Aufgaben ideal ist, dann ist dieses Buch ebenfalls nichts für Sie.

Welche Features bietet C++11 gegenüber C++98 und darüber hinaus? Ein Maschinenmodell, das für moderne Computer mit jeder Menge Parallelität geeignet ist. Sprach- und Standardbibliotheksinstrumente für nebenläufige Programmierung auf Systemniveau (z.B. mithilfe von Mehrkernprozessoren). Verarbeitung regulärer Ausdrücke, Ressourcenverwaltungszeiger, Zufallszahlen, verbesserte Container (einschließlich Hashtabellen) und vieles mehr. Allgemeine und einheitliche Initialisierung, eine einfachere **for**-Anweisung, Verschiebesemantik, grundlegende Unicode-Unterstützung, Lambda-Ausdrücke, allgemeine konstante Ausdrücke, Kontrolle über Standardwerte von Klassen, variadische Templates, benutzerdefinierte Literale und mehr. Denken Sie bitte daran, dass diese Bibliotheken und Sprachfeatures dafür da sind, Programmieretechniken für die Entwicklung von Qualitätssoftware zu unterstützen. Kombinieren Sie diese – wie Bausteine aus einem Baukasten –, um ein konkretes Problem zu lösen, anstatt sie einzeln in relativer Isolation zu verwenden. Ein Computer ist eine universelle Maschine und mit C++ machen Sie dessen Kapazität nutzbar. Insbesondere ist C++ konzeptionell darauf ausgelegt, genügend flexibel und allgemein zu sein, um mit zukünftigen Problemen umgehen zu können, von denen die Designer der Sprache noch nicht einmal geträumt haben.

## Danksagung

Außer den Leuten, die in den Danksagungen der vorherigen Ausgaben erwähnt wurden, möchte ich Pete Becker, Hans-J. Boehm, Marshall Clow, Jonathan Coe, Lawrence Crowl, Walter Daugherty, J. Daniel Garcia, Robert Harle, Greg Hickman, Howard Hinnant, Brian Kernighan, Daniel Krügler, Nevin Liber, Michel Michaud, Gary Powell, Jan Christiaan van Winkel und Leor Zolman danken. Ohne ihre Hilfe wäre dieses Buch mit Sicherheit nicht so gut geworden.

Dank auch an Howard Hinnant für die Beantwortung vieler Fragen zur Standardbibliothek. Andrew Sutton ist der Autor der Origin-Bibliothek, die als Testumgebung für die Emulation von Konzepten in den Template-Kapiteln dient, und der Matrix-Bibliothek, die Thema von Kapitel 29 ist. Die Open-Source-Bibliothek Origin finden Sie im Web, wenn Sie nach „Origin“ und „Andrew Sutton“ suchen.

Ein Dank geht auch an meine Design-Diplomanden, die mehr Probleme mit den „Tour-Kapiteln“ als sonst jemand gefunden haben.

Hätte ich sämtlichen Hinweisen meiner Rezensenten folgen können, wäre das Buch noch besser, doch auch Hunderte Seiten länger geworden. Jeder Fachlektor hat zusätzliche technische Details vorgeschlagen, erweiterte Beispiele und viele nützliche Entwicklungskonventionen; von jedem neuen Rezensenten (oder Lehrer) kamen Vorschläge für ergänzende Beispiele; und viele Rezensenten haben (zu Recht) eingeschätzt, dass das Buch zu umfangreich werden könnte.

Danken möchte ich der Computer Science Department der Princeton University und speziell Prof. Brian Kernighan, die mich für einen Teil des Sabbaticals aufgenommen haben, was mir Zeit verschaffte, dieses Buch zu schreiben. Aus dem gleichen Grund sei dem Computer Lab der Cambridge University und speziell Prof. Andy Hopper gedankt.

*College Station, Texas*

*Bjarne Stroustrup*

*For every complex problem,  
there is an answer that is  
clear, simple, and wrong.*

*- H. L. Mencken*

## ■ 26.1 Einführung

Zu den größten Stärken von Templates zählt ihre Flexibilität, mit der sich Code arrangieren lässt. Der Compiler kombiniert Code (Informationen) aus

- der Template-Definition und ihrer lexikalischen Umgebung,
- den Template-Argumenten und ihrer lexikalischen Umgebung und
- der Einsatzumgebung des Templates,

um eine eindrucksvolle Codequalität zu produzieren. Der Schlüssel zur resultierenden Performance liegt darin, dass sich der Compiler den Code aus diesen Umgebungen gleichzeitig ansehen und ihn anhand aller verfügbaren Informationen verflechten kann. Das Problem dabei ist, dass Code in einer Template-Definition nicht so örtlich begrenzt ist, wie wir es (unter sonst gleichen Umständen) gern hätten. Manchmal ist nicht ganz klar, worauf sich ein Name, der in einer Template-Definition verwendet wird, bezieht:

- Ist es ein lokaler Name?
- Ist es ein Name, der mit einem Template-Argument verbunden ist?
- Ist es ein Name von einer Basisklasse in einer Hierarchie?
- Ist es ein Name aus einem benannten Namespace?
- Ist es ein globaler Name?

Dieses Kapitel diskutiert Fragen, die sich auf *Namensbindung* beziehen, und betrachtet die Konsequenzen für Programmierstile.

- Templates wurden in §3.4.1 und §3.4.2 eingeführt.
- Kapitel 23 gibt eine ausführliche Einführung in Templates und die Verwendung von Template-Argumenten.
- Kapitel 24 führt generische Programmierung und den Kerngedanken der Konzepte ein.
- Kapitel 25 beschäftigt sich mit Details von Klassen-Templates und Funktions-Templates und führt den Begriff der Spezialisierung ein.
- Kapitel 27 diskutiert die Beziehung zwischen Templates und Klassenhierarchien (die generische und objektorientierte Programmierung unterstützen).



- Kapitel 28 stellt Templates als Sprache für das Generieren von Klassen und Funktionen in den Mittelpunkt.
- Kapitel 29 präsentiert ein größeres Beispiel, wie sich die Sprachmittel und Programmier-techniken kombiniert einsetzen lassen.

## ■ 26.2 Template-Instanziierung

Für eine gegebene Template-Definition und eine Verwendung dieses Templates ist es Aufgabe der Implementierung, korrekten Code zu generieren. Aus einem Klassen-Template und einem Satz von Template-Argumenten muss der Compiler die Definition einer Klasse und die Definitionen ihrer Member-Funktionen, die im Programm verwendet werden (und nur diese; §26.2.1) generieren. Aus einer Template-Funktion und einem Satz von Template-Argumenten muss eine Funktion generiert werden. Dieser Vorgang ist die sogenannte *Template-Instanziierung*.

Die generierten Klassen und Funktionen heißen *Spezialisierungen*. Wenn wir zwischen generierten Spezialisierungen und explizit durch den Programmierer geschriebenen Spezialisierungen (§25.3) unterscheiden müssen, sprechen wir von *generierten Spezialisierungen* bzw. *expliziten Spezialisierungen*. Eine explizite Spezialisierung wird oftmals auch als *benutzerdefinierte Spezialisierung* oder einfach *Benutzerspezialisierung* bezeichnet.

Um Templates in nichttrivialen Programmen zu verwenden, muss ein Programmierer die Grundlagen beherrschen, wie Namen, die in einer Template-Definition erscheinen, an Deklarationen gebunden werden und wie sich der Quellcode organisieren lässt (§23.7).

Standardmäßig generiert der Compiler Klassen und Funktionen aus den verwendeten Templates entsprechend den Regeln für Namensbindung (§26.3). Das heißt, ein Programmierer muss nicht explizit angeben, welche Versionen von welchen Templates generiert werden müssen. Dies ist auch sinnvoll, weil der Programmierer gar nicht ohne Weiteres genau wissen kann, welche Versionen eines Templates erforderlich sind. Oftmals werden Templates, von denen der Programmierer nie etwas gehört hat, in der Implementierung von Bibliotheken verwendet, und manchmal werden Templates, die dem Programmierer bekannt sind, mit unbekanntem Template-Argumenttypen verwendet. Zum Beispiel ist `map` (§4.4.3, §31.4.3) der Standardbibliothek in Form eines Rot-Schwarz-Baums als Template mit Datentypen und Operationen implementiert, die außer vielleicht einem neugierigen Benutzer praktisch niemandem bekannt sind. Im Allgemeinen lässt sich der erforderliche Satz der generierten Funktionen nur in Erfahrung bringen, wenn man die in Codebibliotheken verwendeten Templates rekursiv untersucht. Für derartige Analysen sind Computer besser geeignet als der Mensch.

Andererseits ist es für einen Programmierer manchmal wichtig, spezifisch festlegen zu können, wo Code aus einem Template generiert werden soll (§26.2.2). Auf diese Weise kann der Programmierer den Kontext der Instanziierung detailliert steuern.

## 26.2.1 Wann wird Instanziierung gebraucht?

Eine Spezialisierung eines Klassen-Templates muss nur generiert werden, wenn die Definition der Klasse gebraucht wird (§14.7.1). Um speziell einen Zeiger auf eine bestimmte Klasse zu deklarieren, ist die eigentliche Definition einer Klasse nicht erforderlich. Zum Beispiel:

```
class X;
X* p;      // OK: keine Definition von X erforderlich
X a;      // Fehler: Definition von X notwendig
```

Diese Unterscheidung kann ausschlaggebend sein, wenn man Template-Klassen definiert. Eine Template-Klasse wird *nicht* instanziiert, außer wenn ihre Definition tatsächlich benötigt wird. Zum Beispiel:

```
template<typename T>
class Link {
    Link* suc;      // OK: Definition von Link (noch) nicht erforderlich
    // ...
};

Link<int>* pl;     // Instanziierung von Link<int> (noch) nicht erforderlich

Link<int> lnk;    // jetzt müssen wir Link<int> instanziiieren
```

Ein Platz, wo ein Template verwendet wird, definiert einen Punkt der Instanziierung (§26.3.3).

Eine Implementierung instanziiert eine Template-Funktion nur, wenn diese Funktion verwendet wurde. Mit „verwendet“ meinen wir „aufgerufen oder ihre Adresse ermitteln lassen“. Insbesondere zieht die Instanziierung eines Klassen-Templates nicht die Instanziierung aller seiner Member-Funktionen nach sich. Damit ist der Programmierer sehr flexibel, wenn er eine Template-Klasse definiert. Sehen Sie sich dazu folgenden Code an:

```
template<typename T>
class List {
    // ...
    void sort();
};

class Glob {
    // ... keine Vergleichsoperatoren ...
};

void f(List<Glob>& lb, List<string>& ls)
{
    ls.sort();
    // ... Operationen auf lb, aber nicht lb.sort() verwenden ...
}
```

Hier wird **List<string>::sort()** instanziiert, **List<Glob>::sort()** jedoch nicht. Dies verringert die Menge des generierten Codes und erspart es uns, das Programm neu entwerfen zu müssen. Wäre **List<Glob>::sort()** generiert worden, müssten wir entweder die von **List::sort()** benötigten Operationen zu **Glob** hinzufügen, die Funktion **sort()** redefinieren, damit sie kein Member mehr von **List** ist (ohnehin das bessere Design), oder einen anderen Container für **Glob**-Typen verwenden.

## 26.2.2 Manuelle Kontrolle der Instanziierung

Die Sprache setzt für eine Template-Instanziierung keine explizite Benutzeraktion voraus. Allerdings bietet sie zwei Mechanismen, damit der Benutzer bei Bedarf die Kontrolle übernehmen kann. Denn manchmal möchte er

- den Vorgang beim Übersetzen und Binden optimieren, indem redundante Instanziierungen eliminiert werden, oder
- genau wissen, welcher Punkt der Instanziierung verwendet wird, um Überraschungen zu vermeiden, die sich aus komplizierten Umgebungen bei der Namensbindung ergeben.

Eine explizite Instanziierungsanforderung (oftmals einfach *explizite Instanziierung* genannt) ist eine Deklaration einer Spezialisierung, die mit dem Schlüsselwort **template** als Präfix (ohne nachfolgendes **<**) versehen ist:

```
template class vector<int>;           // Klasse
template int& vector<int>::operator[] (int); // Member-Funktion
template int convert<int,double>(double); // Nicht-Member-Funktion
```

Eine Template-Deklaration beginnt mit **template<**, während ein reines **template** eine Instanziierungsanforderung einleitet. Beachten Sie, dass **template** als Präfix vor einer vollständigen Deklaration steht; es genügt nicht, nur einen Namen anzugeben:

```
template vector<int>::operator[]; // Syntaxfehler
template convert<int,double>;    // Syntaxfehler
```

Wie in Aufrufen von Template-Funktionen können die Template-Argumente, die aus den Funktionsargumenten hergeleitet werden können, entfallen (§23.5.1). Zum Beispiel:

```
template int convert<int,double>(double); // OK (redundant)
template int convert<int>(double);       // OK
```

Wird ein Klassen-Template explizit instanziiert, wird auch jede Member-Funktion instanziiert.

Die Instanziierungsanforderungen können sich erheblich auf die Bindungszeit und die Effizienz der Neukompilierung auswirken. Ich habe Beispiele gesehen, in denen ein Bundling der meisten Template-Instanziierungen zu einer einzigen Übersetzungseinheit die Übersetzungszeit von mehreren Stunden auf einige Minuten gedrückt hat.

Wenn zwei Definitionen für dieselbe Spezialisierung existieren, ist das ein Fehler. Es spielt keine Rolle, ob solche Mehrfachspezialisierungen benutzerdefiniert sind (§25.3), implizit generiert (§23.2.2) oder explizit angefordert werden. Allerdings ist ein Compiler nicht verpflichtet, mehrere Instanziierungen in getrennten Übersetzungseinheiten zu diagnostizieren. Eine intelligente Implementierung kann damit redundante Instanziierungen ignorieren und so Probleme vermeiden, die mit der Komposition von Programmen aus Bibliotheken mithilfe expliziter Instanziierung zusammenhängen. Implementierungen müssen aber nicht unbedingt intelligent sein. Benutzer von „weniger intelligenten“ Implementierungen müssen Mehrfachinstanziierungen vermeiden. Wenn sie dies nicht tun, lässt sich ihr Programm im ungünstigsten Fall nicht binden; stillschweigende Bedeutungsänderungen gibt es nicht.

Als Ergänzung zu expliziten Instanzierungsanforderungen bietet die Sprache explizite Anforderungen, *nicht* zu instanzieren (normalerweise **extern templates** genannt). Diese Option bietet sich an, wenn eine Spezialisierung explizit zu instanzieren und ihre Instanzierung in anderen Übersetzungseinheiten mit **extern template** zu unterdrücken ist. Dies spiegelt das klassische Paradigma einer Definition und vieler Deklarationen (§15.2.3) wider. Zum Beispiel:

```
#include "MyVector.h"

extern template class MyVector<int>; // unterdrückt implizite Instanzierung
                                     // an anderer Stelle explizit instanzieren

void foo(MyVector<int>& v)
{
    // ... den Vektor hier verwenden ...
}
```

Das „an anderer Stelle“ könnte etwa so aussehen:

```
#include "MyVector.h"

template class MyVector<int>; // in dieser Übersetzungseinheit instanzieren;
                               // diesen Punkt der Instanzierung verwenden
```

Außer Spezialisierungen für alle Member einer Klasse zu generieren, bestimmt die explizite Instanzierung auch einen einzelnen Punkt der Instanzierung, sodass andere Punkte der Instanzierung (§26.3.3) ignoriert werden können. Diese nutzt man beispielsweise, um explizite Instanzierung in einer gemeinsamen Bibliothek zu platzieren.

## ■ 26.3 Namensbindung

Definieren Sie Template-Funktionen, um Abhängigkeiten von nichtlokalen Informationen zu minimieren. Denn Templates sind dafür vorgesehen, Funktionen und Klassen basierend auf unbekanntem Typen und unbekanntem Kontexten zu generieren. Jede unauffällige Kontextabhängigkeit zeigt sich wahrscheinlich als Problem – und zwar für den Programmierer, der eigentlich gar nicht an den Implementierungsdetails des Templates interessiert ist. Die allgemeine Regel, globale Namen weitestgehend zu vermeiden, sollte in Template-Code besonders ernst genommen werden. Dementsprechend versuchen wir, Template-Definitionen möglichst eigenständig abzufassen und so viel wie möglich von dem, was andernfalls im globalen Kontext erscheinen würde, in Form von Template-Argumenten bereitzustellen (z. B. Traits; §28.2.4, §33.1.3). Verwenden Sie Konzepte, um Abhängigkeiten von Template-Argumenten zu dokumentieren (§24.3).

Allerdings ist es durchaus üblich, dass bestimmte nichtlokale Namen verwendet werden müssen, um die eleganteste Formulierung eines Templates zu erreichen. So schreibt man häufiger einen Satz von kooperierenden Template-Funktionen als lediglich nur eine selbstständige Funktion. Derartige Funktionen können manchmal Klassen-Member sein, sind es jedoch nicht immer. Manchmal sind nichtlokale Funktionen die beste Wahl. Typische Beispiele dafür sind die **swap()**- und **less()**-Aufrufe von **sort()** (§25.3.4). Die Algorithmen der Standardbibliothek verkörpern ein Beispiel im großen Stil (Kapitel 32). Wenn etwas nicht-

lokal sein muss, bevorzugen Sie einen benannten Namespace gegenüber dem globalen Gültigkeitsbereich. Damit wird auch eine gewisse Lokalität bewahrt.

Operationen mit konventionellen Namen und konventioneller Semantik wie zum Beispiel `+`, `*`, `[]` und `sort()` sind eine weitere Quelle für nichtlokale Namen, die in einer Template-Definition verwendet werden. Sehen Sie sich dazu folgenden Code an:

```
bool tracing;

template<typename T>
T sum(std::vector<T>& v)
{
    T t {};
    if (tracing)
        cerr << "sum(" << &v << ")\n";
    for (int i = 0; i!=v.size(); ++i)
        t = t + v[i];
    return t;
}
// ...

#include<quad.h>

void f(std::vector<Quad>& v)
{
    Quad c = sum(v);
}
```

Die harmlos aussehende Template-Funktion `sum()` hängt von mehreren Namen ab, die in ihrer Definition nicht explizit angegeben sind, wie zum Beispiel `tracing`, `cerr` und dem Operator `+`. In diesem Beispiel ist `+` in `<quad.h>` definiert:

```
Quad operator+(Quad,Quad);
```

Vor allem aber befindet sich nichts, was sich auf `Quad` bezieht, im Gültigkeitsbereich, wenn `sum()` definiert wird, und beim Verfasser von `sum()` kann man nicht davon ausgehen, dass er die Klasse `Quad` kennt. So kann der Operator `+` im Programmtext erst hinter `sum()` und sogar zu einem späteren Zeitpunkt definiert werden. Die sogenannte *Namensbindung* sucht die Deklaration für jeden Namen, der explizit oder implizit in einem Template verwendet wird. Allgemein haftet der Template-Namensbindung das Problem an, dass drei Kontexte an einer Template-Instanziierung beteiligt sind und sich nicht sauber trennen lassen:

1. Der Kontext der Template-Definition
2. Der Kontext der Argumenttypdeklaration
3. Der Kontext, in dem das Template verwendet wird

Wenn wir ein Funktions-Template definieren, sollte genügend Kontext verfügbar sein, damit die Template-Definition in Form ihrer tatsächlichen Argumente sinnvoll sein kann, ohne am Punkt der Verwendung „versehentlich“ etwas aus der Umgebung aufzugreifen. Um uns dabei zu helfen, trennt die Sprache Namen, die in einer Template-Definition verwendet werden, in zwei Kategorien:

1. *Abhängige Namen*: Namen, die von einem Template-Parameter abhängen. Derartige Namen werden am Punkt der Instanziierung (§26.3.3) gebunden. Im `sum()`-Beispiel



```
int gg(Quad);
int zz = ff(Quad{2});
```

Würde die Funktion `gg(Quad{1})` als abhängig betrachtet, wäre ihre Bedeutung für einen Leser der Template-Definition höchst mysteriös. Wenn ein Programmierer möchte, dass `gg(Quad)` aufgerufen wird, sollte die Deklaration von `gg(Quad)` vor der Definition von `ff()` erscheinen, sodass sich `gg(Quad)` im Gültigkeitsbereich befindet, wenn `ff()` analysiert wird. Dies ist genau die gleiche Regel, die auch für Definitionen von Nicht-Template-Funktionen (§26.3.2) gilt.

Standardmäßig wird von einem abhängigen Namen angenommen, dass er etwas benennt, was kein Typ ist. Um also einen abhängigen Namen als Typ zu verwenden, müssen Sie dies mit dem Schlüsselwort **typename** ausdrücken. Zum Beispiel:

```
template<typename Container>
void fct(Container& c)
{
    Container::value_type v1 = c[7];    // Syntaxfehler: value_type wird als Name
                                        // eines Nicht-Typs angenommen
    typename Container::value_type v2 = c[9];    // OK: value_type benennt
                                                // vermutlich einen Typ
    auto v3 = c[11];                    // OK: Typ vom Compiler herausfinden lassen
    // ...
}
```

Wir können dieses umständliche **typename** vermeiden, wenn wir einen Typalias einführen (§23.6). Zum Beispiel:

```
template<typename T>
using Value_type = typename T::value_type;

template<typename Container>
void fct2(Container& c)
{
    Value_type<Container> v1 = c[7];    // OK
    // ...
}
```

Analog dazu müssen wir bei der Benennung eines Member-Templates nach einem `.` (Punkt), `->` oder `::` das Schlüsselwort **template** angeben. Zum Beispiel:

```
class Pool {    // ein Allokator
public:
    template<typename T> T* get();
    template<typename T> void release(T*);
    // ...
};

template<typename Alloc>
void f(Alloc& all)
{
    int* p1 = all.get<int>();            // Syntaxfehler: für get wird angenommen,
                                        // dass ein Nicht-Template benannt wird
    int* p2 = all.template get<int>();  // OK: für get() wird angenommen,
```

```

// ...
}
// dass ein Template benannt wird

void user(Pool& pool)
{
    f(pool);
    // ...
}

```

Verglichen mit **typename** (um explizit auszudrücken, dass ein Name einen Typ benennt) ist es eher selten, mit **template** explizit auszudrücken, dass ein Name ein Template benennt. Beachten Sie die unterschiedlichen Positionen des Schlüsselworts, das die Mehrdeutigkeit beseitigt: **typename** erscheint vor dem qualifizierten Namen und **template** unmittelbar vor dem Template-Namen.

### 26.3.2 Bindung am Punkt der Definition

Wenn der Compiler eine Template-Definition sieht, ermittelt er, welche Namen abhängig sind (§26.3.1). Ist ein Name abhängig, wird die Suche nach seiner Deklaration auf den Zeitpunkt der Instanziierung verschoben (§26.3.3).

Namen, die nicht von einem Template-Argument abhängen, werden wie Namen behandelt, die nicht in Templates vorkommen; sie müssen sich am Punkt der Definition im Gültigkeitsbereich (§6.3.4) befinden. Zum Beispiel:

```

int x;

template<typename T>
T f(T a)
{
    ++x;      // OK: x ist im Gültigkeitsbereich
    ++y;      // Fehler: kein y im Gültigkeitsbereich und y hängt nicht von T ab
    return a; // OK: a ist abhängig
}

int y;

int z = f(2);

```

Eine einmal gefundene Deklaration wird auch verwendet, selbst wenn später vielleicht eine „bessere“ Deklaration gefunden werden könnte. Zum Beispiel:

```

void g(double);
void g2(double);

template<typename T>
int ff(T a)
{
    g2(2); // g2(double) aufrufen
    g3(2); // Fehler: kein g3() im Gültigkeitsbereich
    g(a);  // g(double) aufrufen, g(int) nicht im Gültigkeitsbereich
    // ...
}

```



```
void g(int);
void g3(int);

int x = ff(5);
```

Von **ff(5)** wird hier **g(double)** aufgerufen. Die Definition von **g(int)** erscheint zu spät, um betrachtet zu werden – genau als wäre **ff()** kein Template oder **g** hätte eine Variable benannt.

### 26.3.3 Bindung am Punkt der Instanziierung

Der Kontext, in dem die Bedeutung eines abhängigen Namens ermittelt wird (§26.3.1), ergibt sich aus der Verwendung eines Templates für eine gegebene Menge von Argumenten. Dies ist der sogenannte *Punkt der Instanziierung* für diese Spezialisierung (§iso.14.6.4.1). Jede Verwendung eines Templates für eine gegebene Menge von Template-Argumenten definiert einen Punkt der Instanziierung. Für ein Funktions-Template befindet sich dieser Punkt im nächsten globalen oder durch einen Namespace definierten Gültigkeitsbereich, der seine Verwendung umschließt, unmittelbar nach der Deklaration, die diese Verwendung enthält. Zum Beispiel:

```
void g(int);

template<typename T>
void f(T a)
{
    g(a);          // g wird am Punkt der Instanziierung gebunden
}
void h(int i)
{
    extern void g(double);
    f(i);
}
// Punkt der Instanziierung für f<int>
```

Der Punkt der Instanziierung für **f<int>()** liegt *außerhalb* von **h()**. Dies ist wichtig, um sicherzustellen, dass die in **f()** aufgerufene Funktion **g()** die globale Funktion **g(int)** und nicht die lokale Funktion **g(double)** ist. Ein nicht qualifizierter Name, der in einer Template-Definition verwendet wird, kann niemals an einen lokalen Namen gebunden werden. Lokale Namen zu ignorieren, ist unabdingbar, um viele makroartige Verhaltensweisen zu verhindern.

Damit rekursive Aufrufe möglich sind, liegt der Punkt der Instanziierung für ein Funktions-Template *nach* der Deklaration, die es instanziiert. Zum Beispiel:

```
void g(int);

template<typename T>
void f(T a)
{
    g(a);          // g wird am Punkt der Instanziierung gebunden
    if (a>1) h(T(a-1)); // h wird am Punkt der Instanziierung gebunden
}

```

```
enum Count { one=1, two, three }

void h(Count i)
{
    f(i);
}
// Punkt der Instanziierung für f<int>
```

Hier ist es notwendig, dass der Punkt der Instanziierung *nach* der Definition von **h()** erscheint, um den (indirekt rekursiven) Aufruf **h(T(a-1))** zu ermöglichen.

Für eine Template-Klasse oder einen Klassen-Member liegt der Punkt der Instanziierung unmittelbar *vor* der Deklaration, die ihre/seine Verwendung enthält.

```
template<typename T>
class Container {
    vector<T> v;           // Elemente
    // ...
public:
    void sort();         // Elemente sortieren
    // ...
};

// Punkt der Instanziierung von Container<int>
void f()
{
    Container<int> c;     // Punkt der Verwendung
    c.sort();
}
```

Hätte der Punkt der Instanziierung nach **f()** gelegen, würde der Aufruf **c.sort()** scheitern, um die Definition von **Container<int>** zu finden.

Wenn man Abhängigkeiten mithilfe von Template-Argumenten explizit darstellt, wird der Template-Code verständlicher und man kann sogar auf lokale Informationen zugreifen. Zum Beispiel:

```
void fff()
{
    struct S { int a,b; };
    vector<S> vs;
    // ...
}
```

Hier ist **S** ein lokaler Name, doch da wir ihn als explizites Argument verwenden, anstatt zu versuchen, seinen Namen in der Definition von **vector** zu vergraben, haben wir keine potenziell überraschenden Feinheiten zu erwarten.

Warum vermeiden wir dann nicht komplett nichtlokale Namen in Template-Definitionen? Dies würde sicherlich das technische Problem mit der Namenssuche lösen, doch wir wollen – wie bei normalen Funktions- und Klassendefinitionen – in der Lage sein, „andere Funktionen und Typen“ ungehindert in unserem Code zu verwenden. Wenn man jede Abhängigkeit in ein Argument umwandelt, führt das zu sehr chaotischem Code. Zum Beispiel:

```

template<typename T>
void print_sorted(vector<T>& v)
{
    sort(v.begin(),v.end());
    for (const auto& x : v)
        cout << x << '\n';
}

void use(vector<string>& vec)
{
    // ...
    print_sorted(vec);    // mit std::sort sortieren, dann mit std::cout ausgeben
}

```

Hier verwenden wir lediglich zwei nichtlokale Namen (**sort** und **cout**, die beide aus der Standardbibliothek stammen). Um diese zu eliminieren, müssten wir Parameter hinzufügen:

```

template<typename T, typename S>
void print_sorted(vector<T>& v, S sort, ostream& os)
{
    sort(v.begin(),v.end());
    for (const auto& x : v)
        os << x << '\n';
}

void fct(vector<string>& vec)
{
    // ...
    using Iter = decltype(vec.begin());    // Iteratortyp von vec
    print_sorted(vec, std::sort<Iter>, std::cout);
}

```

In diesem trivialen Fall ist ziemlich viel Code erforderlich, um die Abhängigkeit vom globalen Namen **cout** zu beseitigen. Wie aber **sort()** veranschaulicht hat, kann der Code durch zusätzliche Parameter im Allgemeinen wesentlich weitschweifiger werden, ohne dass er dadurch verständlicher wird.

Wären die Regeln für die Namensbindung bei Templates radikal restriktiver als die Regeln für Nicht-Template-Code, wäre es zudem eine vollkommen andere Herausforderung, Template-Code zu schreiben als Nicht-Template-Code. Templates und Nicht-Template-Code würden nicht mehr so einfach und problemlos zusammenarbeiten.

### 26.3.4 Mehrere Punkte der Instanziierung

Eine Template-Spezialisierung kann generiert werden

- an jedem Punkt der Instanziierung (§26.3.3),
- an jedem darauffolgenden Punkt in einer Übersetzungseinheit oder
- in einer Übersetzungseinheit, die speziell für das Generieren von Spezialisierungen erstellt wurde.

Dies spiegelt drei offensichtliche Strategien wider, nach denen eine Implementierung Spezialisierungen generieren kann:

1. Eine Spezialisierung generieren, wenn der erste Aufruf erscheint
2. Alle Spezialisierungen, die für eine Übersetzungseinheit erforderlich sind, am Ende der Übersetzungseinheit generieren
3. Alle Spezialisierungen, die für das Programm erforderlich sind, generieren, nachdem jede Übersetzungseinheit des Programms abgearbeitet wurde

Alle drei Strategien weisen Stärken und Schwächen auf und es sind auch Kombinationen dieser Strategien möglich.

Ein Template, das mehrfach mit derselben Menge von Template-Argumenten verwendet wird, hat entsprechend viele Instanzierungspunkte. Ein Programm ist unzulässig, wenn es möglich ist, zwei unterschiedliche Bedeutungen zu konstruieren, indem zwei verschiedene Instanzierungspunkte ausgewählt werden. Wenn sich also die Bindungen eines abhängigen oder eines unabhängigen Namens unterscheiden können, ist das Programm unzulässig. Zum Beispiel:

```
void f(int);           // hier geht es mir um int-Werte

namespace N {
    class X { };
    char g(X,int);
}

template<typename T>
char ff(T t, double d)
{
    f(d);             // f wird an f(int) gebunden
    return g(t,d);   // g könnte an g(X,int) gebunden werden
}

auto x1 = ff(N::X{},1.1); // ff<N::X,double>; kann g an N::g(X,int) binden,
                          // wobei 1.1 zu 1 eingeschränkt wird

namespace N {         // N erneut öffnen, um doubles zu berücksichtigen
    double g(X,double);
}

auto x2 = ff(N::X{},2.2); // ff<N::X,double>; bindet g an N::g(X,double);
                          // die beste Übereinstimmung
```

Für **ff()** haben wir zwei Instanzierungspunkte. Für den ersten Aufruf könnten wir die Spezialisierung bei der Initialisierung von **x1** generieren und **g(N::X, int)** aufrufen lassen. Alternativ dazu könnten wir warten und die Spezialisierung am Ende der Übersetzungseinheit generieren, sodass **g(N::X, double)** aufgerufen wird. Folglich ist der Aufruf **ff(N::X{}, 1.1)** ein Fehler.

Es gilt als nachlässige Programmierung, eine überladene Funktion zwischen zwei ihrer Deklarationen aufzurufen. In einem großen Programm hätte ein Programmierer keinen Grund, dort ein Problem zu vermuten. In diesem konkreten Fall könnte ein Compiler die Mehrdeutigkeit abfangen. Ähnliche Probleme können allerdings auch in getrennten Übersetzungseinheiten auftreten und dann wird die Erkennung wesentlich schwerer (sowohl für Compiler als auch für Programmierer). Von einer Implementierung darf nicht erwartet werden, dass sie derartige Probleme abfängt.

Um überraschende Namensbindungen zu vermeiden, ist es am besten, Kontextabhängigkeiten in Templates zu begrenzen.

### 26.3.5 Templates und Namespaces

Wenn eine Funktion aufgerufen wird, lässt sich ihre Deklaration auch dann finden, wenn sie sich nicht im Gültigkeitsbereich befindet, sofern sie im selben Namespace wie eines ihrer Argumente deklariert ist (§14.2.4). Dies ist wichtig für Funktionen, die in Template-Definitionen aufgerufen werden, weil es sich hierbei um den Mechanismus handelt, durch den abhängige Funktionen während der Instanziierung gefunden werden. Die Bindung von abhängigen Namen erfolgt (§iso.14.6.4.2), indem

1. die Namen im Gültigkeitsbereich an dem Punkt, an dem das Template definiert wird, und
2. die Namen im Namespace eines Arguments eines abhängigen Aufrufs (§14.2.4)

betrachtet werden. Zum Beispiel:

```
namespace N {
    class A { /* ... */ };
    char f(A);
}

char f(int);

template<typename T>
char g(T t)
{
    return f(t);    // f() abhängig von T auswählen
}

char f(double);

char c1 = g(N::A());    // bewirkt, dass N::f(N::A) aufgerufen wird
char c2 = g(2);        // bewirkt, dass f(int) aufgerufen wird
char c3 = g(2.1);     // bewirkt, dass f(int) aufgerufen wird; f(double) wird
                       // nicht berücksichtigt
```

Hier ist **f(t)** zweifellos abhängig, sodass wir **f** am Punkt der Definition nicht binden können. Um eine Spezialisierung für **g<N::A>(N::A)** zu generieren, sucht die Implementierung im Namespace **N** nach **f()**-Funktionen und findet **N::f(N::A)**.

Die Funktion **f(int)** wird gefunden, weil sie sich im Gültigkeitsbereich am Punkt der Definition des Templates befindet. Dagegen wird **f(double)** nicht gefunden, weil sie sich am Punkt der Definition des Templates nicht im Gültigkeitsbereich befindet (§iso.14.6.4.1) und eine argumentabhängige Suche (§14.2.4) findet keine globale Funktion, die nur Argumente integrierter Typen übernimmt. Meiner Ansicht nach kann man das leicht vergessen.

### 26.3.6 Zu aggressive ADL

Die argumentabhängige Suche (Argument-Dependent Lookup, ADL) ist sehr hilfreich, um Weitschweifigkeit zu vermeiden (§14.2.4). Zum Beispiel:

```
#include <valarray>    // Hinweis: kein "using namespace std;"

valarray<double> fct(valarray<double> v1, valarray<double> v2, double d)
{
    return v1+d*v2;    // OK wegen ADL
}
```

Ohne argumentabhängige Suche würde der Additionsoperator **+** für **valarray** nicht gefunden werden. Wie es aussieht, bemerkt der Compiler, dass das erste Argument an **+** ein **valarray** ist, das in **std** definiert ist. Deshalb sucht er nach dem Operator **+** in **std** und findet ihn (in **<valarray>**).

Allerdings kann die ADL in Verbindung mit unbeschränkten Templates „zu aggressiv“ sein. Sehen Sie sich dazu folgenden Code an:

```
#include<vector>
#include<algorithm>
// ...

namespace User {
    class Customer { /* ... */ };
    using Index = std::vector<Customer*>;

    void copy(const Index&, Index&, int deep);    // tiefe oder flache Kopie,
                                                // je nach dem Wert von deep

    void algo(Index& x, Index& y)
    {
        // ...
        copy(x,y,false);    // Fehler
    }
}
```

Es besteht die begründete Vermutung, dass der Autor von **User** für **User::algo()** den Aufruf **User::copy()** vorgesehen hat. Dieser findet allerdings nicht statt. Der Compiler stellt fest, dass **Index** eigentlich ein **vector** ist (in **std** definiert), sucht nach einer relevanten Funktion in **std** und findet in **<algorithm>**:

```
template<typename In, typename Out>
Out copy(In,In,Out);
```

Offensichtlich stellt dieses allgemeine Template eine perfekte Übereinstimmung für **copy(x,y,false)** dar. Andererseits kann die Funktion **copy()** in **User** nur mit einer Konvertierung von **bool** nach **int** aufgerufen werden. Für dieses Beispiel (wie auch für äquivalente Beispiele) ist die Auflösung durch den Compiler eine Überraschung für die meisten Programmierer und eine Quelle für sehr undurchsichtige Bugs. Vermutlich ist es als Designfehler der Sprache anzusehen, mithilfe von ADL völlig allgemeine Templates zu suchen. Immerhin erfordert **std::copy()** ein Paar Iteratoren (und nicht einfach zwei Argumente desselben Typs, wie die beiden **Index**-Argumente). Der Standard besagt das, der Code aber

nicht. Viele derartige Probleme lassen sich mithilfe von Konzepten lösen (§24.3, §24.3.2). Wenn zum Beispiel der Compiler erkannt hätte, dass `std::copy()` zwei Iteratoren verlangt, wäre das Ergebnis ein klar erkennbarer Fehler gewesen.

```
template<typename In, typename Out>
Out copy(In p1, In p2, Out q)
{
    static_assert(Input_iterator<In>(), "copy(): In ist kein Eingabeiterador");
    static_assert(Output_iterator<Out>(), "copy(): Out ist kein Ausgabeiterador");
    static_assert(Assignable<Value_type<Out>, Value_type<In>>(),
                  "copy(): Werttypkonflikt");
    // ...
}
```

Noch besser wäre es gewesen, wenn der Compiler erkannt hätte, dass `std::copy()` nicht einmal ein gültiger Kandidat für diesen Aufruf ist, und `User::copy()` aufgerufen worden wäre. Zum Beispiel (§28.4):

```
template<typename In, typename Out,
         typename = Enable_if(<Input_iterator<In>()
                               && Output_iterator<Out>()
                               && Assignable<Value_type<Out>, Value_type<In>>())>>
Out copy(In p1, In p2, Out q)
{
    // ...
}
```

Unglücklicherweise sind viele derartige Templates in Bibliotheken untergebracht, die ein Benutzer nicht modifizieren kann (z. B. die Standardbibliothek).

Am besten vermeidet man völlig allgemeine (vollkommen unbeschränkte) Funktions-Templates in Headern, die auch Typdefinitionen enthalten können. Allerdings ist das schwer durchzusetzen. Falls Sie ein solches Template benötigen, lohnt es sich oftmals, es mit einer Einschränkungüberprüfung zu schützen.

Was kann ein Benutzer tun, wenn eine Bibliothek unbeschränkte Templates enthält, die Probleme verursachen? Häufig wissen wir, aus welchem Namespace unsere Funktion kommen sollte, sodass wir ihn explizit angeben können. Zum Beispiel:

```
void User::algo(Index& x, Index& y)
{
    User::copy(x,y,false); // OK
    // ...
    std::swap(*x[i],*x[j]); // OK: nur std::swap wird betrachtet
}
```

Möchten wir den Namespace nicht spezifizieren, aber sicherstellen, dass eine bestimmte Version einer Funktion durch Überladen von Funktionen betrachtet wird, können wir eine `using`-Deklaration (§14.2.2) verwenden. Zum Beispiel:

```
template<typename Range, typename Op>
void apply(const Range& r, Op f)
{
    using std::begin;
    using std::end;
    for (auto& x : r)
        f(x);
}
```

Nun befinden sich die Standardfunktionen `begin()` und `end()` in der überladenen Menge, die von der bereichsbasierten `for`-Schleife verwendet wird (außer wenn `Range` über Member `begin()` und `end()` verfügt; §9.5.1).

### 26.3.7 Namen aus Basisklassen

Besitzt ein Klassen-Template eine Basisklasse, kann sie auf Namen aus dieser Basisklasse zugreifen. Wie bei anderen Namen gibt es zwei unterschiedliche Möglichkeiten:

- Die Basisklasse hängt von einem Template-Argument ab.
- Die Basisklasse hängt nicht von einem Template-Argument ab.

Der zweite Fall ist einfach und lässt sich behandeln wie Basisklassen in Klassen, die keine Templates sind. Zum Beispiel:

```
void g(int);

struct B {
    void g(char);
    void h(char);
};

template<typename T>
class X : public B {
public:
    void h(int);
    void f()
    {
        g(2);      // B::g(char) aufrufen
        h(2);      // X::h(int) aufrufen
    }
    // ...
};
```

Wie immer überdecken lokale Namen andere Namen, sodass `h(2)` an `X::h(int)` gebunden und `B::h(char)` niemals betrachtet wird. Analog dazu wird der Aufruf `g(2)` an `B::g(char)` gebunden, ohne irgendwelche Funktionen zu berücksichtigen, die außerhalb von `X` deklariert sind. Das heißt, die globale Funktion `g()` wird überhaupt nicht betrachtet.

Für Basisklassen, die von einem Template-Parameter abhängen, müssen wir etwas sorgfältiger sein und explizit angeben, was wir möchten. Sehen Sie sich dazu folgenden Code an:

```
void g(int);

struct B {
    void g(char);
    void h(char);
};

template<typename T>
class X : public T {
public:
    void f()
    {
        g(2);      // ::g(int) aufrufen
    }
};
```



```

    }
    // ...
};
void h(X<B> x)
{
    x.f();
}

```

Warum ruft **g(2)** nicht **B::g(char)** auf (wie im vorherigen Beispiel)? Weil **g(2)** nicht vom Template-Parameter **T** abhängt. Die Funktion wird demzufolge am Punkt der Definition gebunden; Namen vom Template-Argument **T** (das hier als Basisklasse verwendet wird) sind (noch) nicht bekannt und werden deshalb nicht betrachtet. Sollen Namen aus einer abhängigen Klasse berücksichtigt werden, müssen wir die Abhängigkeit klarmachen. Dazu haben wir drei Möglichkeiten:

- einen Namen mit einem abhängigen Typ qualifizieren (z. B. **T::g**)
- angeben, dass der Name auf ein Objekt dieser Klasse verweist (z. B. **this->g**)
- den Namen mit einer **using**-Deklaration in den Gültigkeitsbereich bringen (z. B. **using T::g**).

Zum Beispiel:

```

void g(int);
void g2(int);

struct B {
    using Type = int;
    void g(char);
    void g2(char);
};

template<typename T>
class X : public T {
public:
    typename T::Type m;    // OK
    Type m2;              // Fehler (kein Type im Gültigkeitsbereich)

    using T::g2();        // T::g2() in den Gültigkeitsbereich bringen

    void f()
    {
        this->g(2);       // T::g aufrufen
        g(2);            // ::g(int) aufrufen; überrascht?
        g2(2);           // T::g2 aufrufen
    }
    // ...
};

void h(X<B> x)
{
    x.f();
}

```

Nur am Punkt der Instanziierung wissen wir, ob das für den Parameter **T** verwendete Argument (hier **B**) die erforderlichen Namen besitzt.

Man kann leicht vergessen, Namen von einer Basisklasse zu qualifizieren, und der qualifizierte Code sieht oftmals ein wenig weitschweifig und unordentlich aus. Die Alternative wäre aber, dass ein Name in einer Template-Klasse abhängig vom Template-Argument manchmal an einen Member der Basisklasse und manchmal an eine globale Entität gebunden würde. Das ist ebenfalls nicht ideal und die Sprachregel unterstützt die Faustregel, dass eine Template-Definition so eigenständig wie möglich sein sollte (§26.3).

Es kann recht lästig sein, den Zugriff auf abhängige Member der Basisklasse zu qualifizieren. Allerdings helfen explizite Qualifizierungen bei der Programmpflege, sodass sich der ursprüngliche Autor nicht über die zusätzliche Tipparbeit beschweren sollte. Häufig tritt dieses Problem auf, wenn eine ganze Klassenhierarchie in Templates überführt wird. Zum Beispiel:

```
template<typename T>
class Matrix_base {    // Speicher für Matrizen, Operationen aller Elemente
    // ...
    int size() const { return sz; }
protected:
    int sz;           // Anzahl der Elemente
    T* elem;         // Matrix-Elemente
};

template<typename T, int N>
class Matrix : public Matrix_base<T> {    // N-dimensionale Matrix
    // ...
    T* data()      // Zeiger auf Elementspeicher zurückgeben
    {
        return this->elem;
    }
};
```

Hier ist die Qualifizierung `this->` erforderlich.

## ■ 26.4 Ratschläge

1. Lassen Sie den Compiler/die Implementierung Spezialisierungen je nach Bedarf generieren; §26.2.1.
2. Instanzieren Sie explizit, wenn Sie genaue Kontrolle über die Instanzierungsumgebung benötigen; §26.2.2.
3. Instanzieren Sie explizit, wenn Sie die zum Generieren von Spezialisierungen benötigte Zeit optimieren wollen; §26.2.2.
4. Vermeiden Sie in einer Template-Definition subtile Kontextabhängigkeiten; §26.3.
5. Die in einer Template-Definition verwendeten Namen müssen sich im Gültigkeitsbereich befinden oder über argumentabhängige Suche (ADL) auffindbar sein; §26.3, §26.3.5.
6. Halten Sie den Bindungskontext zwischen Instanzierungspunkten unverändert; §26.3.4.
7. Vermeiden Sie völlig allgemeine Templates, die durch ADL gefunden werden können; §26.3.6.

8. Verwenden Sie Konzepte und/oder `static_assert`, um die Verwendung ungeeigneter Templates zu vermeiden; §26.3.6.
9. Begrenzen Sie die Reichweite der ADL mit `using`-Deklarationen; §26.3.6.
10. Qualifizieren Sie Namen aus einer Template-Basisklasse je nach Bedarf mit `->` oder `T::`; §26.3.7.

# Index

## Symbole

## (Makrooperator) 369  
& (Adresse) 187  
& (Und) 299  
\* (Dereferenzierung) 187  
. (Punktoperator) 220, 505  
... (Ellipse) 92  
[] (Indexoperator) 99, 190, 727  
^ (Exklusiv-Oder, XOR) 299  
{ } (Argumente an Konstruktor) 532  
| (Oder) 299  
~ (Komplement) 299  
+= (Verkettung) 99  
<< (basic\_string) 1125  
<< (Linksschieben) 29  
>> (Operator) 505  
>> (basic\_string) 1125  
>> (Rechtsschieben) 299

## A

ABA-Problem 1300  
abgeleitete Klassen 625  
Abhängigkeiten  
– Standardargumente 355  
– Variablen 344  
ABI (Application Binary Interface) 229, 577, 1083  
Ableitung  
– Darstellung 627  
– geschützte 655  
– Klassenhierarchien 77  
– öffentliche 654  
– private 655  
– protected 655

– public 654  
– Templates 821  
abort() 1367  
abs() 1257, 1258  
Absolutwert 1257  
Abstract Syntax Tree (AST) 708  
abstrakt  
– Klassen 74, 647  
– Syntaxbäume 708  
– Typen 73  
Abstraktionsmechanismen 53  
accumulate() 140, 1271, 1272  
acos() 1257  
acquire 1296  
Adapter  
– bind() 1044  
– Container- 957, 993  
– Funktions- 1043  
– Iteratoren 1036  
– mem\_fn() 1043  
– not1() 1043  
– not2() 1043  
– regex\_iterator 1152  
– regex\_token\_iterator 1153  
– Zufallszahlenmodule 1279  
add\_const 1104  
add\_cv 1104  
add\_pointer 1106  
add\_volatile 1104  
Addition  
– complex 1259  
– Matrix 901  
– Zeiger 199  
address\_family\_not\_supported 949  
address\_in\_use 949

- address\_not\_available 949
- Ad-hoc-Konzepte 767
- adjacent\_difference() 1272, 1274
- adjacent\_find() 1008
- adjustfield 1180
- ADL (Argument-Dependent Lookup) 815
- advance() 1036
- Akkumulationsalgorithmus 759
- Aktivierungsblock 183
- <algorithm> 930, 1001
- Algorithmen 112, 758, 1001
  - adjacent\_find() 1008
  - Akkumulation 759
  - all\_of() 1006
  - any\_of() 1006
  - binäre Suche 1021
  - binary\_search() 1021
  - Container 119
  - copy() 1011
  - copy\_backward() 1011
  - copy\_if() 1011
  - copy\_n() 1011
  - count() 1007
  - count\_if() 1007
  - equal() 1008
  - fill() 1016
  - fill\_n() 1016
  - find() 1007
  - find\_end() 1008
  - find\_first\_of() 1007
  - find\_if() 1007
  - find\_if\_not() 1007
  - for\_each() 1006
  - Funktionsobjekte 1042
  - Funktions-Templates 758
  - generate() 1016
  - Heap- 1024
  - includes() 1023
  - inplace\_merge() 1022
  - is\_heap() 1025
  - is\_heap\_until() 1025
  - is\_partitioned() 1015
  - is\_sorted() 1018
  - is\_sorted\_until() 1018
  - iter\_swap() 1017
  - Komplexität 1005
  - lexicographical\_compare() 1026
  - Lifting 758
  - make\_heap() 1025
  - max() 1027
  - max\_element() 1027
  - Maximum 1026
  - Mengen 1023
  - merge() 980, 1022
  - min() 1026
  - min\_element() 1027
  - Minimum 1026
  - minmax() 1027
  - minmax\_element() 1027
  - Mischen 1014
  - mismatch() 1008
  - modifizierende 1010
  - move() 1011
  - move\_backward() 1011
  - mutierende 1010
  - nichtmodifizierende 1006
  - none\_of() 1006
  - nth\_element() 1018
  - numerische 1271
  - Operatoren, relationale 962
  - pair() 1027
  - partial\_sort() 1018
  - partial\_sort\_copy() 1018
  - partition() 1014, 1015
  - partition\_copy() 1015
  - Partitionieren 1015
  - partition\_point() 1015
  - Permutationen 1015
  - pop\_heap() 1025
  - push\_heap() 1025
  - R 1277
  - random\_shuffle() 1014
  - remove() 1013
  - replace() 1013
  - reverse() 1013
  - rotate() 1014
  - search() 1009
  - search\_n() 1009
  - Sequenzen 1001
  - Sequenzen teilen 1015
  - set\_difference() 1023
  - set\_intersection() 1023
  - set\_symmetric\_difference() 1024
  - set\_union() 1023
  - sort() 1018
  - sort\_heap() 1025
  - Sortieren 1018

- Speicher, temporärer 1086
- stable\_partition() 1015
- stable\_sort() 1018
- Standard- 118
- Stichproben 1277
- Suchen 1018
- swap() 1017
- swap\_ranges() 1017
- transform() 1010
- try\_lock() 1329
- Überblick 119
- uninitialized\_copy() 1016
- uninitialized\_copy\_n() 1016
- uninitialized\_fill() 1016
- uninitialized\_fill\_n() 1016
- unique() 1012
- unique\_copy() 1012
- Alias 93
  - <fstream> 1163
  - Namespaces 443
  - nicht verwenden 847
  - Templates 750
  - Typen 184, 508
- aligned\_storage 1106
- aligned\_union 1106
- alignment\_of 1103
- alignof() 165
- all 1203
- all() 1059
- allocate\_shared() 1072
- allocation unit (Zuordnungseinheit) 231
- allocator\_traits 1079
- allocator\_type 966
- all\_of() 1006
- Allokation 602
- Allokatoren 1076
  - Arenen 1076
  - mit eigenem Gültigkeitsbereich 1080
  - Platzierungsoperator new 311
  - scoped\_allocator\_adaptor 1081
  - stack 995
  - Standard- 1077
  - Zeiger-Traits 1080
- Allzweckprogrammiersprache 11
- already\_connected 949
- always\_noconv() 1240
- and 282
- AND 1058
- and\_eq 282
- Anführungszeichen
  - doppelte 156
  - einfache 156
  - Strings 194
  - unescaped 194
- anonyme Unions 236, 611
- Anpassungspunkte 798
- Anweisungen 245
  - Auswahl- 248
  - bereichsbasierte for- 51, 935
  - Blöcke 246
  - Deklarationen 246
  - do 257
  - for 255
  - for, bereichsbasiert 254
  - goto 258
  - if 248
  - leere 246
  - Schleifen 253
  - switch 250
  - throw 62
  - Verbund- 246
  - while 50, 256
  - Zusammenfassung 245
- Anwendungsoperator 344
- any() 1059
- any\_of() 1006
- APL
  - iota 1275
- app 1176
- append()
  - basic\_string 1128
- Application Binary Interface (ABI) 229, 577
- apply()
  - valarray 1265
- Archetyp 778
- Arenen
  - Allokatoren 1076
  - placement new 311
  - Platzierungsoperator new 311
- argc 276, 479
- Argumente
  - argc 479
  - argv 479
  - Arrays 347
  - Auflösungsregeln 360
  - formale 344
  - Funktions-Templates 741
  - Konstruktoren 494

- Lambda-Ausdrücke 325
- Listen 349
- main() 276
- Makros 368
- Operationen 786
- Referenzen 291, 345
- Richtlinien 1003
- Standard- 354
- Templates 788
- übergeben 126
- argumentabhängige Suche 815
- Argument-Dependent Lookup (ADL) 815
- argument-dependent lookup (argument-  
abhängige Namensauflösung) 432
- Argumentersetzung 747
- argument\_list\_too\_long 949
- argument\_out\_of\_domain 949
- argv 276, 479
- argv[0] 352
- Arithmetik 46
  - gemischte Datentypen 581
  - Operationen 141
  - ratio 1097
  - Vektoren 143
- arithmetische Typen 151
- Arkuskosinus 1257
- Arkussinus 1257
- Arkustangens 1257
- ARM (The Annotated C++ Reference Manual) 30
- array 226, 1051, 1052
- <array> 930
- Arrays 190
  - Argumente 347
  - assoziative 111, 930
  - decay 1104
  - initialisieren 191
  - mehrdimensionale 191, 200
  - new 308
  - Row-major Order 1267
  - Schleifen 51
  - Strukturen 225
  - übergeben 201
  - valarray 1260
  - Zeiger 196
  - Zerfall 825, 1104
  - Zugriff 198
  - zugrunde liegende 314
- Array\_type 844
- asctime() 1365
- asin() 1257
- asinh() 1257
- assert() 394, 941
- Assert 396
- Assertionen 394, 941
  - <cassert> 394
  - implementierungsabhängige Features 148
  - statische 64
- assign() 1117
  - basic\_string 1128
- associated types (zugeordnete Typen) 731
- assoc\_laguerre() 1258
- assoc\_legendre() 1258
- assoziative Container 981
- AST (Abstract Syntax Tree) 708
- async() 133, 936, 1336, 1347
- at() 106
  - basic\_string 1124
- atan() 1257
- atan2() 1257, 1265
- atanh() 1257
- ate 1176
- atexit() 482
- atof() 1362
- atoi() 1362
- atol() 1362
- atoll() 1362
- atomar
  - Operationen 1292, 1298
  - Typen 1298
- atomic 1298
- <atomic> 933
- atomic\_compare\_exchange\_weak() 1303
- atomic\_flag 1303
- atomic\_flag\_clear() 1303
- atomic\_flag\_clear\_explicit() 1303
- ATOMIC\_FLAG\_INIT 1304
- atomic\_flag\_test\_and\_set() 1303
- atomic\_flag\_test\_and\_set\_explicit() 1303
- atomic\_init() 1302
- atomic\_is\_lock\_free() 1302
- atomic\_load() 1303
- atomic\_signal\_fence() 1304
- atomic\_store() 1302
- atomic\_thread\_fence() 1304
- at\_quick\_exit() 482
- Attribute
  - [[carries\_dependency]] 342, 1297
  - [[noreturn]] 342

- Aufrufoperator 344
- Aufzählungen 57, 237
  - einfache 241
  - mask 1235
  - unbenannte 243
  - Zeichenklassen 1235
- Aufzählungsklassen 57, 237
  - später definieren 240
- Ausdrücke 245, 263
  - Adresse konstanter 292
  - bedingte 301
  - const-Typen 290
  - Klammern 284
  - konstante 287
  - Kurzschlussauswertung 284
  - primäre 267
  - reguläre 139
  - Reihenfolge der Auswertung 283
  - vollständige 285
- Ausdruckssequenzierung 283
- Ausdrucks-Templates 920
- Ausgabe
  - Geldbeträge 1230
  - numerische 1222
- Ausgabeoperationen 1171
  - virtuelle 1173
- Ausgabepuffer 1192
- ausgerichtet 221
- Aushungern 1319
- Auslassungszeichen 92
- Ausnahmen 375, 376, 936
  - abfangen 398, 402
  - alternative Ansichten 380
  - asynchrone Ereignisse 380
  - auslösen 398
  - bad\_alloc 402
  - basic\_ios 1167
  - basic\_string 1119
  - Benutzercode 409
  - delete 312
  - Division durch null 283
  - Effizienz 384
  - erneut auslösen 403
  - Garantien 386, 387
  - gültiger Zustand 386
  - Handler für unerwartete 401
  - Handler, mehrere 405
  - Hierarchie 937
  - invalid\_argument 1127
  - istream 1226
  - keine Fehler 380
  - length\_error 1122
  - longjmp() 388
  - nested\_exception 939
  - new 312
  - noexcept 385, 400
  - nothrow 312
  - out\_of\_range 106, 1127
  - POD 401
  - runtime\_error 1202
  - setstate() 1226
  - set\_terminate() 407
  - system\_error 946, 1322
  - terminate() 941
  - throw 62
  - Überlauf 283
  - unerwartete 941
  - Unterlauf 283
  - veraltete 1375
  - vermeiden 312
  - Weiterleitung 938
  - what() 937
- ausnahmensicher 386
- Ausnahmensicherheitsgarantien 375
- Ausnahmespezifikationen 401
- Ausrichtung 165, 1106
  - Container 968
- Auswahl 852
  - Komposition 445
  - Typen 853
- Auswahanweisungen 248
- Auswertungen
  - bedingte 342
  - partielle 1044
- auto 48, 179
  - Container 970
  - {}-Listen 180
- awk 1141
- Axiome 772
  
- B**
- back() 996, 1124
- back\_insert\_iterator 1036, 1039
- Backslashes 156
  - reguläre Ausdrücke 1139
  - Strings 194
  - Stringliterale 139



- Backspace 156
- Backtick 1146
- bad() 1163, 1177
- bad\_address 949
- bad\_alloc 402
- badbit 1176
- bad\_file\_descriptor 949
- bad\_message 949
- Barton-Nackman-Trick 834
- basefield 1180
- basic 1141
- basic\_filebuf 1163, 1240
- basic\_ios 1162, 1167, 1175
  - Operationen 1177
- basic\_istream 1160
- basic\_istream 1168
- basic\_regex 1142
- basic\_streambuf 1189
  - Operationen 1189
- basic\_string 723, 1051, 1118
  - Ausnahmen 1119
  - Ein-/Ausgabe 1125
  - Ersetzen 1129
  - Indizierung 1124
  - Iteratoren 1127
  - Konstruktoren 1121
  - Operationen 1123
  - Teilstrings 1132
  - Zuweisungen 1128
- basic\_stringstream 1164
- Basisklassen 74
  - Initialisierer 543
  - Konstruieren virtueller 686
  - Kopieren 550
  - replizierte 689
  - Template-Parameter 829
  - using 642
  - Verband 686
  - virtuelle 683
  - wiederholt verwenden 682
  - Zugriff 654
- Bäume
  - abstrakte Syntax- 920
- bedingte Ausdrücke 301
- Bedingungen 152
  - Deklarationen 252
- Bedingungsvariablen 1330
  - condition\_variable\_any 1335
  - Nachrichtenfluss steuern 1332
- Befehle
  - umordnen 1291
- Befehlszeilenargumente 276
- before() 713
- beg 1177
- begin() 108, 1041, 1127, 1265
  - bereichsbasierte for-Anweisung 1041
  - Iteratoren 1041
- Beinahe-Container 955, 957, 1051
- Beispiele
  - Compiler 304
  - SI-Einheiten 883
  - Taschenrechner 263
  - vector 410
- Bell 156
- benutzerdefinierte Literale 604
- benutzerdefinierte Typen 53, 151
- Benutzeroberflächen
  - grafische 95
- Benutzerspezialisierungen 791, 802
- Bereiche
  - halboffene 1029
  - Zugriffsfunktionen 1041
- Bereichsauflösungsoperator 173
- bereichsbasierte for-Anweisung 51, 934, 935
  - begin() 1041
  - end() 1041
- Bereichsüberprüfungen 106
  - Arrays 190
  - Slices 1269
- bernoulli\_distribution() 1282
- Bernoulli-Verteilungen 1282
- Besitz
  - strenger 1066
  - Zeiger 1065
- Besucher 708, 836
- beta() 1258
- Bezeichner 170
  - Groß-/Kleinschreibung 170
  - Iteratoren 929
- Beziehungen
  - Standard- 826
  - Templates 826
- Bibliotheken 95
  - BLAS (Basic Linear Algebra Subprograms) 1267
  - Standardbibliothek 96
- bidirectional\_iterator\_tag 1032

- Binärbäume
    - ausgeglichene 767
    - balancierte 767
  - binary 1176
  - binary\_search() 1021
  - bind() 1044
    - Wrapper 1045
    - Zufallszahlen 1276
  - bind1st() 1044
  - Binden 753
  - Binder 456, 1044
  - Bindung 456
    - doppelte 706, 1174
    - dynamische 705
    - externe 457
    - interne 457
    - Konstanten 1381
    - Konventionen 467
    - Nicht-C++-Code 465
    - Punkt der Definition 809
    - Punkt der Instanziierung 810
    - static 457, 460
  - Bindungsblöcke 466
  - Bindungsstärke 281
  - bit\_and 1043
  - bitand 282
  - Bitarrays 158
  - Bitfelder 230, 293
    - Kosten 1290
  - bit\_or 1043
  - bitor 282
  - Bitposition 1056
  - bitset 936, 1051, 1055
    - Konstruktoren 1056
    - Operationen 1058
  - <bitset> 930
  - bit\_xor 1043
  - BLAS (Basic Linear Algebra Subprograms) 1267
  - Blöcke 172, 246
  - Body (Textkörper) 318
  - bool 1384
  - \_Bool 1384
  - boolalpha 1180, 1182
  - Boost 35
  - boundary character 1139
  - break
    - Schleifen 257
    - switch 251
  - broken\_pipe 949
  - broken\_promise 946, 1344
  - bsearch() 930, 1367
  - Buckets 992
- ## C
- C
    - kein C++ 1379
  - C++
    - C-Makros 1380
    - Entwurf 10
    - Schlüsselwörter 171, 1380
  - C++11
    - Erweiterungen 1372
    - Features 1372
  - Callbacks 317
  - calloc() 1363
  - call\_once 343
  - call\_once() 936, 1329
  - cancel 1316
  - capacity() 1123
  - capture list (Erfassungsliste) 91
  - Carriage Return 156
  - carries\_dependency 342, 1297
  - case
    - brake 251
    - Deklarationen 252
    - return 251
  - CAS-Operationen 1300
  - <cassert> 394, 931
  - Casting (Typumwandlung, explizite) 17, 330
  - catch(...) 409
  - cauchy\_distribution() 1284
  - cbegin() 1127
  - <ccomplex> 933
  - <cctype> 931
  - ceil() 1257
  - ceil() 1127
  - cerr 100, 1161
  - <cerrno> 931, 949
  - <cfenv> 933
  - <cfloat> 294, 932
  - char 153
    - Vorzeichen 155
  - char16\_t 153
  - char32\_t 153
  - CHAR\_BIT 1256
  - CHAR\_MAX 1256
  - CHAR\_MIN 1256

- char\_traits 1117
- C-Header 452
- check\_bounds() 911
- check\_non\_jagged() 909
- chi\_squared\_distribution() 1283
- <chrono> 129, 930, 1089
  - duration 1090
- cin 1161
- <cinttypes> 933
- class 492
  - Template-Parameter 782
- clear() 1123, 1177, 1303
- <climits> 932, 1256
- <clocale> 932
- clock() 1364
- clock\_t 1364
- clog 1161
- close()
  - get() 1243
- Closure (Funktionsabschluss) 319, 325, 919
- <cmath> 140, 932, 1257
- codecvt 1215, 1239
- <codecvt> 932
- codecvt\_utf8 1248
- Codepunkte 196
- Codeseiten 1205
- collate 1141, 1203, 1215
  - Facetten, benannte 1220
- common\_type 1106
- Common\_type 902
- compare() 1117, 1132
- Compare-and-Swap 1299
- compare\_exchange\_strong() 1298
- compare\_exchange\_weak() 1298
- comp\_ellint\_1() 1258
- comp\_ellint\_2() 1258
- comp\_ellint\_3() 1258
- Compiler
  - Beispiel 304
- compl 282
- complex 69
  - Hilfsfunktionen 587
  - Konvertierung, einschränkende 1259
  - Zugriffsfunktionen 586
- <complex> 140, 932, 933, 1384
- <complex.h> 1384
- composition closure objects 920
- compositors 920
- concurrency (Parallelität) 1287
  - conditional 845, 1106
  - Conditional 852
  - condition\_variable 936, 1330
  - <condition\_variable> 129, 933
  - condition\_variable\_any 1335
  - connection\_aborted 949
  - connection\_already\_in\_progress 949
  - connection\_refused 949
  - connection\_reset 949
  - const 70, 203, 287, 335
    - Daten-Member von Templates 731
    - Member-Funktionen 500
  - const\_cast 330
  - constexpr 203, 287, 335, 339, 340
    - bedingte Auswertung 342
    - Daten-Member von Templates 731
    - Einschränkungsüberprüfungen 768
    - Nebeneffekte 341
    - ODR 341
    - Referenzen 341
    - Zeiger 342
  - const\_iterator 966
  - const\_local\_iterator 966
  - const\_pointer 966
  - const\_pointer\_cast() 1073
  - const\_reference 966
  - const\_reverse\_iterator 966
  - const wchar\_t[] 195
  - consume 1296
  - Container 71, 104, 974
    - Algorithmen 119
    - Allokatoren 1076
    - array 1052
    - assoziative 930, 981
    - Beinahe- 957, 1051
    - bitset 1055
    - Darstellung 958
    - Destruktoren 967
    - Duplikate entfernen 1012
    - Elemente 106, 960
    - Elementzugriff 971
    - geordnete assoziative 981
    - Größe 969
    - Hashtabellen 986
    - Initialisierung 72
    - Iteratoren 969
    - Kapazität 969
    - Konstruktoren 967, 982
    - Kopieren 81

- Lastfaktor 992
  - Listen 979
  - Listenoperationen 972
  - Member-Typen 966
  - Operatoren 963
  - Ordnungsfunktionen 110
  - Stack-Operationen 971
  - Standard-Allokatoren 1077
  - STL- 974
  - Überblick 110, 955
  - ungeordnete assoziative 981, 986
  - unordered\_map 987
  - vector 104, 974
  - vergleichen 973
  - Verschieben 83
  - vertauschen 973
  - Zuweisungen 967
  - Containeradapter 957, 993
    - stack 994
  - continue 258
  - controls 693
  - converted() 1249
  - copy() 1011, 1118
    - basic\_string 1124
  - copy\_backward() 1011
  - copyfmt() 1178
  - copy\_if() 1011
  - copy\_n() 1011
  - Copy-On-Write 553
  - cos() 1257
  - count() 1007, 1059
  - count\_if() 1007
  - cout 100, 1161
  - \_\_cplusplus 371, 1384
  - crbegin() 1127
  - crend() 1127
  - Crosscast 694
  - cross\_device\_link 949
  - CRTP (Curiously Recurring Template Pattern) 834
  - <csetjmp> 932
  - cshift()
    - valarray 1264
  - <csignal> 932
  - C-Standardbibliothek
    - C-Strings 1361
    - Datum und Uhrzeit 1364
    - Konvertierungen 1362
    - Speicher 1362
  - <cstdalign> 933
  - <cstdarg> 351, 932
  - <cstdbool> 933
  - <cstdint> 932
  - <cstdint> 158, 932
  - <cstdio> 931
  - <cstdlib> 930, 931, 932, 1258, 1285, 1367
  - c\_str() 1124
  - C-Strings 53, 1361
    - konvertieren 931
    - Konvertierungen 1362
  - <cstring> 931
  - <ctgmath> 933
  - ctime() 1365
  - <ctime> 930, 932
  - ctype 1203, 1215, 1235
  - <cuchar> 931
  - cur 1177
  - Curiously Recurring Template Pattern (CRTP) 834
  - current\_exception() 409, 938
  - curr\_symbol() 1228
  - Currying 1044
  - <cwchar> 931
  - <cwctype> 931
  - cyl\_bessel\_i() 1258
  - cyl\_bessel\_j() 1258
  - cyl\_bessel\_k() 1258
  - cyl\_neumann() 1258
- ## D
- Dämonen 1313
  - Darstellung
    - Container 958
  - data()
    - basic\_string 1124
  - Data Races 125, 148, 343, 1293
    - globale Variablen 459
    - nichtlokaler Speicher 1317
    - vermeiden 1318
  - \_\_DATE\_\_ 371
  - Dateien 1355
    - Header- 460
    - Modi 1163
    - Modus 1356
    - öffnen 1355
    - schließen 1355
    - <stdio> 1355

- Daten
  - gemeinsam nutzen 127
- Daten-Member
  - Templates 730
- Datenstrukturen
  - zusammensetzen 830
- Datentypen
  - Arithmetik mit gemischten 581
- dateorder() 1233
- Datum und Uhrzeit
  - <chrono> 1089
  - clock\_t 1364
  - <ctime> 1364
  - dateorder() 1233
  - duration\_cast 1093
  - Epochen 1093
  - Formatierung 1232, 1366
  - Formatmodifizierer 1367
  - Funktionen 1364
  - strftime() 1366
  - time\_get 1233
  - time\_put 1232
  - time\_t 1364
  - tm 1232, 1364
  - Zeitdauern 1090
  - Zeitpunkte 1093
- DBL\_EPSILON 1257
- DBL\_MAX 294, 1257
- DBL\_MIN 1257
- Deadlocks 343, 1322
- Deallokation 602
- dec 1180, 1183
- decay 1104
- decimal\_point() 1221, 1227, 1228
- declare\_no\_pointers() 1085
- decltype() 179, 181, 852
- declval() 1107, 1109
- deep copy (tiefe Kopie) 552
- default
  - switch 251
  - =default 560
- defaultfloat 1183
- deferred 1343, 1347
- #define 368
- Definitionen 58, 166
  - bedingte 859
  - Deklarationen 166
  - Enable\_if 859
  - klasseninterne 499
  - Deklarationen 46, 58, 166, 245, 246
    - Bedingungen 252
    - case 252
    - Definitionen 166
    - friend 617
    - Struktur 168
    - using 429
  - Deklarationsoperatoren 52
  - Deklaratoren 168
  - Dekrementieren 301
    - Zeiger 600
  - delegierende Konstruktoren 544
  - delete
    - nothrow 312
  - Deleter 1066, 1070
  - deprecated 1374
  - Deque
    - stack 995
  - <deque> 930
  - Dereferenzieren 187
  - Dereferenzierungsoperator 598
  - Designeinschränkungen 927
  - destination\_address\_required 950
  - Destruktion 710
  - Destruktoren 71, 523, 630
    - aufrufen 527
    - Container 967
    - Ressourcen 525
    - Threads 1311
    - virtual 528
  - detach() 1308, 1313
  - device\_or\_resource\_busy 950, 1323
  - Dezimalpunkt 1221
  - Dezimaltrennzeichen 1221
    - decimal\_point() 1227
    - frac\_digits() 1229
  - dictionary (Wörterbuch) 109
  - difference\_type 966
  - difftime() 1364
  - directory\_not\_empty 950
  - direkte Initialisierung 497
  - Direktiven
    - #include 455
    - #pragma 372
    - using 61, 430
  - discard\_block\_engine 1279
  - discrete\_distribution() 1284
  - disguised pointers (getarnte Zeiger) 1083
  - diskriminierte Unions 235

distance() 1036  
distribution 141  
div() 1258  
divides 1043  
Division  
– complex 1259  
– durch null 267  
do 257  
Domänenfehler 1258  
doppelte Bindung 1174  
doppelt überprüfte Sperrung 1301  
dot\_product() 904  
double-checked locking 1301  
double-dispatch (doppelte Bindung) 705  
Downcast 694  
Dreiecksmatrix 914  
Duck-Typing 758  
– Typargumente 783  
Duplikate  
– entfernen 1012  
duration 1089, 1090  
duration\_cast 1093  
Durchhangeln 379  
dynamic\_cast 330, 695, 936  
– nullptr 695  
– Referenzen 697  
– Schnittstellen 701  
– static\_cast 700  
– Typ ermitteln 693  
dynamic\_pointer\_cast() 1073  
dynamische Bindung 705  
dynamische Polymorphie 626  
dynamischer Speicher 54, 303

## E

E2BIG 949  
EACCESS 951  
EADDRINUSE 949  
EADDRNOTAVAIL 949  
EAFNOSUPPORT 949  
EAGAIN 951  
EALREADY 949  
E/A-Streams 1159  
eback() 1191  
EBADF 949  
EBADMSG 949  
EBCDIC 154  
EBUSY 950

ECANCELED 951  
ECHILD 950  
Echtzeituhren 1095  
ECMAScript 1141  
ECONNABORTED 949  
ECONNREFUSED 949  
ECONNRESET 949  
EDEADLK 951  
EDESTADDRREQ 950  
EDOM 949, 1258  
EEXIST 950  
EFAULT 949  
EFBIG 950  
Effizienz  
– Ausnahmen 384  
egptr() 1191  
egrep 1141  
EHOSTUNREACH 950  
EIDRM 950  
EILSEQ 950  
Ein-/Ausgabe  
– basic\_string 1125  
– Manipulatoren 1182  
– Operationen 1168  
– ungepufferte 1189  
Eine-Definition-Regel 341, 462, 492  
Eingabe 268  
– formatierte 1169  
– Geldbeträge 1231  
– Integer-Wert-Eingabebox 663  
– numerische 1225  
– unformatierte 1170  
Eingabeiteratoren 774  
Eingabeoperationen 1168  
Eingabepuffer 1193  
Einheiten  
– Literale 886  
– SI- 883  
– Unit 883  
EINPROGRESS 951  
Einrückungen 259  
einschränkende Konvertierungen 47, 152, 293,  
358  
– complex 1259  
– {}-Initialisierer 533  
– Listeninitialisierung 175  
Einschränkungen 767  
– Design- 927  
– Konzepte 766

- Einschränkungsüberprüfungen 775
- EINTR 950
- EINVAL 950
- EIO 950
- EISCONN 949
- EISDIR 950
- Elemente *siehe auch* Member
  - Container 106
- Elementzugriff 971
- ellint\_1() 1258
- ellint\_2() 1258
- ellint\_3() 1258
- Ellipse 92, 876
- ELOOP 951
- EMFILE 951
- EMLINK 951
- empty() 1123
- EMSGSIZE 950
- enable\_if 748, 859, 1106
- Enable\_if 859
  - Beispiele 863
  - Implementierung 862
  - Konzepte 863
  - verwenden 860
- ENAMETOOLONG 950
- encoding() 1240, 1241
- end 1177
- end() 108, 1041, 1127, 1265
  - bereichsbasierte for-Anweisung 1041
  - Iteratoren 1041
- #endif 370
- endl 1184
- ends 1184
- End Of File 1166
- Endlosschleifen 256
  - for 266
  - while 266
- ENETDOWN 950
- ENETRESET 950
- ENETUNREACH 950
- ENFILE 951
- engine 141
- Engine (Zufallszahlenmodul) 1276
- ENOBUFFS 950
- ENODATA 950
- ENODEV 950
- ENOENT 950
- ENOEXEC 950
- ENOLCK 950
- ENOLINK 950
- ENOMEM 951
- ENOMSG 950
- ENOPROTOPT 950
- ENOSPC 950
- ENOSR 950
- ENOSTR 951
- ENOSYS 950
- ENOTCONN 951
- ENOTDIR 950
- ENOTEMPTY 950
- ENOTRECOVERABLE 951
- ENOTSOCK 951
- ENOTSUP 951
- ENOTTY 950
- entropy() 1280
- Entwurfsmuster
  - Besucher 708, 836
  - Visitor 708
- enum class 57, 237
- Enumerationen 57
- Enumeratoren 57, 237
  - errc 945
- ENXIO 950
- eof() 1118, 1177
- EOF 1125
- eofbit 1176
- EOPNOTSUPP 951
- EOVERFLOW 951
- EOWNERDEAD 951
- EPERM 951
- EPIPE 949
- Epochen 1093
- eptr() 1192
- EPROTO 951
- EPROTONOSUPPORT 951
- EPROTOTYPE 951
- eq() 1117
- eq\_int\_type() 1117
- equal() 1008
- equal\_to 1009, 1042
- Equality\_comparable 770
- ERANGE 951, 1127, 1258, 1362
- erase()
  - basic\_string 1128
- Ereignisse
  - asynchrone 380
  - register\_callback() 1179
  - Threads 129

- Erfassung 321
- Erfassungslisten 91, 318
- EROFS 951
- errno 943
  - <cerrno> 949
  - Enumeratoren 945
  - Fehlercodes 949
- errno 378, 1127, 1362
- error 1241
- error\_category 943
- error\_code 943
- error\_condition 943
- Ersetzungsfehler 748
- Escapezeichen 193
- ESPIPE 950
- ESRCH 950
- ETIME 951
- ETIMEDOUT 951
- ETXTBSY 951
- event 1179
- event\_callback 1179
- EWOLDBLOCK 951
- exception 404
  - <exception> 931, 932
  - set\_terminate() 407
- exception\_ptr 938
- exceptions() 1167, 1178
- exchange() 1298
- EXCLUSIVE OR 1058
- EXDEV 949
- executable\_format\_error 950
- exit() 339, 1367
- Exklusiv-Oder (XOR) 299
- exp() 1257
- expint() 1258
- explicit 496, 583
  - Konvertierungsoperatoren 590
- exponential\_distribution() 1283
- export 1375, 1380
- extended 1141
- extent 1103
- extern 465
- externe Bindung 457
- extreme\_value\_distribution() 1283

## F

- Fabrik 674
- facet 1203, 1209
- Facetten 1201
  - benutzerdefinierte 1211
  - codecvt 1239
  - collate 1203
  - ctype 1235
  - hash() 1218
  - Interpunktion 1227
  - Kategorien 1203
  - money\_get 1227, 1231
  - money\_put 1227, 1230
  - num\_get 1225
  - numpunct 1203, 1221
  - Standard- 1215
  - time\_put 1232
  - transform() 1218
  - verwenden 1214
  - Zugriff auf 1210
- factory 674
- fail() 1177
- failbit 1176
- Fakultät
  - Templates 856
- false 151
- falsename() 1221
- fclose() 1355
- Features 1374
  - deprecated 1374
- Fehler
  - ±1 614
  - Argumentersetzung 747
  - Ausnahmen 380
  - Bereichs- 1258
  - Domänen- 1258
  - EDOM 1258
  - ERANGE 1258
  - errno 378
  - Ersetzungs- 748
  - future-Operationen 1344
  - Heisenbugs 1315
  - Mutexe 1322
  - Off-by-One- 614
  - Runden bei Zeitdauern 1097
- ±1-Fehler 614
- Fehlerbehandlung 61, 375, 935
  - herkömmliche 378



- hierarchische 382
- Streams 1166
- Taschenrechner 274
- Fehlercodes 943
  - errc 949
  - future\_errc 952
  - io\_errc 952
  - potenziell portable 947
  - zuordnen 947
- Fehlererkennung
  - Templates 729
- Fehlerkategorien 945
- Felder 230
- Fences 1303
- fetch\_add() 1301, 1302
- fetch\_and() 1301
- fetch\_or() 1301
- fetch\_sub() 1301, 1302
- fetch\_xor() 1301
- \_\_FILE\_\_ 371, 397
- file\_exists 950
- file\_too\_large 950
- filename\_too\_long 950
- fill() 1016, 1178
- fill\_n() 1016
- final 335, 640
- finally 391
- find() 1007, 1118
  - Familie 1130
  - parallel 1349
- find\_end() 1008
- find\_first\_not\_of() 1131
- find\_first\_of() 1007, 1131
- find\_if() 1007
- find\_if\_not() 1007
- find\_last\_not\_of() 1131
- find\_last\_of() 1131
- first 1062
- fisher\_f\_distribution() 1284
- fixed 1180, 1183
- flache Kopie 552
- flags() 1180
- Flags 1303
- flip() 1058
- floatfield 1180
- floating-point types 160
- floor() 1257
- FLT\_DIG 1256
- FLT\_MAX 294, 1256
- FLT\_MAX\_10\_EXP 1256
- FLT\_MIN 1256
- flush 1184
- flush() 1172
- fmod() 1257
- fmtflags 1180
- fold 1273
- fopen() 1355
- for 255
  - bereichsbasiert 51, 254, 607, 934, 935
  - Endlosschleifen 266
- for\_each() 1006
- Form Feed 156
- format\_default 1146
- format\_first\_only 1146
- format\_no\_copy 1146
- format\_sed 1146
- Formate
  - allgemeine 1181
  - feste 1181
  - Genauigkeit 1181
  - wissenschaftliche 1181
- Formatierung
  - Datum und Uhrzeit 1232
  - Geldbeträge 1226
  - numerische 1220
  - printf() 1357
  - reguläre Ausdrücke 1146
  - Streams 1179
  - strftime() 1366
- Formularvorschub 156, 1115
- Fortran
  - Indizierung 898
- forward() 1109, 1110
- forward\_as\_tuple() 1064
- forward iterators 135
- forward\_iterator\_tag 1032
- forward\_list 979
- <forward\_list> 930
- fprintf() 1356
- frac\_digits() 1228, 1229
- Fragezeichen 156
- free() 1363
- Freispeicher 54, 303
- Friends 617
  - finden 619
  - Member-Funktionen 619
  - Templates 738
- from\_bytes() 1249

- from\_time\_t() 1095
- front() 996, 1124
- front\_insert\_iterator 1036, 1039
- fstream 1162
  - bad() 1163
- <fstream> 931, 1162
  - Alias 1163
- \_\_func\_\_ 371
- function
  - bind() 1046
- <functional> 930, 1042
- function\_not\_supported 950
- function-style cast (funktionale Notation der Typumwandlung) 332
- Funktionen
  - Anwendungsoperator 344
  - Array-Argumente 347
  - Aufrufoperator 344
  - auswählen 850
  - bedingte Auswertung 342
  - Bereichszugriffs- 1041
  - Binden überladener 1044
  - Bindung 467
  - Callbacks 317
  - check\_non\_jagged() 909
  - <cmath> 1257
  - const 70
  - constexpr 340
  - <cstdlib> 1258
  - Definitionen 58, 335
  - Deklarationen 333
  - diverse 1367
  - ergänzende 611
  - explizite Qualifizierung 637
  - formale Argumente 344
  - friend 617
  - gelöschte 566
  - Gleichheits- 989
  - globale 479
  - Hash- 989
  - inline 335
  - klasseninterne Definitionen 499
  - Klassifizierung 1115
  - komplexe Zahlen 141
  - Konstruktoren 56
  - Konvertierungs- 827
  - lokale 344
  - main() 45, 479
  - make\_pair() 138
  - Member- 335, 489
  - move() 213, 558
  - Nachbedingungen 361
  - noexcept 400
  - noreturn 342
  - operator[] 595
  - Operatoren 573
  - overloading 356
  - Parameter 344
  - pfind() 1349
  - Prädikate 848
  - printf() 1356
  - Prototypen 1379
  - reguläre Ausdrücke 1147
  - rein virtuelle 74, 647
  - rekursive 338
  - Rückgabetypen 358
  - Rückgabewerte 337
  - spezielle mathematische 1258
  - Standardargumente 354
  - Suffixrückgabetypen 337
  - Tabelle für virtuelle 76, 636
  - Templates 88
  - try 405
  - Typ- 135, 843
  - Überladen 356
  - Überladen über Gültigkeitsbereiche 642
  - überschreiben 635
  - verlassen 339
  - virtuelle 74, 76, 634
  - virtuelle Ausgabe- 1173
  - Vorbedingungen 361
  - vtbl 636
  - Weiterleitung 878
  - Zeiger 363
- funktionale Notation 331
- Funktionsabschluss 319, 919
- Funktionsadapter 1043
  - mem\_fn() 1046
- funktionsähnliche Objekte 597
- Funktionsaufrufe 596
- Funktionsdefinitionen 335
- Funktionsdeklarationen 333
- Funktionsobjekte 89, 597, 1042
  - bind() 1044
  - equal\_to 1009
- Funktionsoperatoren 596
- Funktionsparameterpakete 876

- Funktionstabellen
  - virtuelle 76
- Funktions-Templates 739
  - Algorithmen 758
  - Argumente 741
  - Argumente herleiten 742
  - Argumentersetzung 747
  - Standardargumente 790
  - Template-Funktionen 724
  - überladen 745
  - Vererbung 749
- Funktionszeiger 363
  - konvertieren 364
- Funktoren 89
- future 131, 665, 936, 1336, 1337, 1342
  - Zustand beobachten 1343
- <future> 933
- future\_already\_retrieved 1344
- future\_errc 943, 952

## G

- gamma\_distribution() 1283
- Ganzzahl-literale 158
- Garantien
  - Ausnahmen 387
- Garbage Collection 1082
  - ABI (Application Binary Interface) 1083
  - declare\_no\_pointers() 1085
  - konservative 1085
  - sicher abgeleitete Zeiger 1083
  - Verschränkung 553
- gaußsches Eliminationsverfahren 915
- gbump() 1191
- Gebietsschemas *siehe* Locales
- Geldbeträge
  - Ausgabe 1230
  - Eingabe 1231
  - Formatierung 1226
  - Interpunktion 1227
- gemeinsamer Status 1337
- Genauigkeit 1181
  - Zeitgeber 1093
- generate() 1016
- Generatoren
  - Zufallszahlen 1278
- generische Programmierung 757
  - Algorithmen 758
- geometric\_distribution() 1282

- geordnet
  - assoziative Container 981
  - Typen 770
- get() 1063, 1065, 1072, 1343, 1346
  - messages 1243
  - money\_get 1231
  - num\_get 1225
- get\_allocator() 1123
- get\_area (Lesebereich) 1190
- getc() 1361
- getchar() 1361
- get\_date() 1233
- get\_deleter() 1073
- get\_future() 1340
- get\_id() 1308, 1315
- getline() 102
  - basic\_string 1125
- getloc() 1156, 1189
- get\_money() 1184
- get\_monthname() 1233
- get\_obj() 702
- Get/Set-Funktionen 586
- get\_temporary\_buffer() 1086
- get\_terminate() 941
- get\_time() 1184, 1233
- get\_weekday() 1233
- get\_year() 1233
- getarnte Zeiger 1083
- Gleichheit
  - complex 1260
- Gleichheitsfunktionen 989
- Gleichungen
  - gaußsches Eliminationsverfahren 915
  - lineare 914
  - testen 917
- Gleitkomma-division
  - Rest 1257
- Gleitkommaliterale 160
- Gleitkommatypen 160
- Glocke 156
- Glockenkurve 141
- gmtime() 1364
- good() 1177
- goodbit 1176
- goto 258
- gptr() 1191
- Grafische Benutzeroberflächen 95
- Graphical User Interface (GUI) 95, 663
- greater 1042

- greater\_equal 1042
- Greatest Common Denominator (GCD) 1097
- greedy match (gierige Übereinstimmung) 1140
- Grenzen
  - numerische 1253
- Grenzwerte
  - Makros 1256
- Grenzzeichen 1139
- grep 1140, 1141
- Größen
  - Container 969
  - Typen 162
- Groß-/Kleinschreibung 170
  - konvertieren 1116
  - reguläre Ausdrücke 1141
  - Sortieren 1220
  - Vergleiche 961
- grouping() 1221, 1228
- Gruppen
  - reguläre Ausdrücke 1140
- gslice 895, 1260
- gslice\_array 1260
- guard (Wächter) 392
- GUI (Graphical User Interface) 95, 663
- gültiger Zustand 386
- Gültigkeitsbereiche 172
  - globale 429
  - Klassen 1381
  - Strukturen 1381
  - Templates 738
  - Überladen 359
  - Überladen von Funktionen 642
  - using 642
  - verlassen (Thread) 1315

## H

- Handler
  - Ausnahmen, unerwartete 401
  - mehrere 405
  - terminate 407
- hardware\_concurrency() 1308
- Hardwaregrößen 148
- Has\_equal 770
- has\_facet() 1210
- hash
  - bitset 1059
  - Typen 987
- hash() 1218

- Hash
  - Richtlinien 992
- hash\_code() 713
- hasher 966
- Hashfunktionen 110, 989
- Hashtabellen 111, 713, 986
  - Buckets 992
  - Überlauf 957
- has\_virtual\_destructor 1103
- Header
  - C- 452
  - mehrere Dateien 472
  - Standardbibliothek 464, 929
  - Taschenrechner 275
  - verwenden 468, 477
- Header-Dateien 59, 460
  - Vorübersetzung 460
- Heap 54, 303
  - Algorithmen 1024
- Heisenbugs 1315
  - detach() 1315
- Hello, World! (Beispiel) 44
- Heraufstufungen 293
- hermite() 1258
- hex 1180, 1183
- hexfloat 1183
- Hexadezimalzahlen 156
- Hierarchien
  - Ausnahmen 937
  - Templates 827
- high\_resolution\_clock 1089, 1096
- Hilfsfunktionen 888
  - complex 587
  - Date 515
- Hiragana 1239
- Horizontal Tab 156
- Horizontaltabulator 156, 1115
- host\_unreachable 950
- Hyperthreading 1309

## I

- icase 1141
- id 1308
- identifier\_removed 950
- Identität 1309
- if 248
- #ifdef 370
- ifstream 1162

- illegal\_byte\_sequence 950
- imbue() 1156, 1178
- Implementierungen 438
  - alternative 670
  - Enable\_if 862
  - freistehende 149, 927
  - gehostete 149
  - Schnittstellen 440
- implementierungsabhängig 147
- Implementierungsklassen
  - mehrere 676
- Implementierungsmodelle
  - Lambda-Ausdrücke 318
  - Listen 313
- Implementierungsvererbung 79, 649, 664
- in 1176
- in() 1240
- inappropriate\_io\_control\_operation 950
- in\_avail() 1191
- include guards (Include-Wächter) 478
- #include 455
- includes() 1023
- Include-Wächter 367, 478
- independent\_bits\_engine 1279
- Indexoperator [] 190
- indirect\_array 1260
- Indirektion 187
- Indizes 595
- Indizierung
  - basic\_string 1124
  - C-Stil 898
  - Fortran-Stil 898
  - Maske 1264
  - Slices 897
  - valarray 1263
- init() 1178
- Initialisierer 174
  - Basisklassen 543
  - fehlende 177
  - klasseninterne 498, 544, 645
  - statische Member 546
- Initialisierung
  - Basisklassen 541
  - Container 72
  - direkte 497
  - Klassenobjekte 529
  - Konstruktoren 533
  - Kopier- 539
  - Member 541
  - Nebenläufigkeit 480
  - nichtlokaler Variablen 479
  - Notation 494
  - Referenzen 208
  - struct 220
  - thread\_local 1318
  - universelle 532
- Initialisierungslisten 178
  - Member 541
  - verwenden 538
- Initialisierungslisten-Konstruktor 536
- initializer\_list 934
- <initializer\_list> 932, 934
- Inkrementieren 301
  - Zeiger 600
- inline 335, 1380
  - constexpr 339
- Inline-Namespaces 448
- inner\_product() 904, 1271, 1273
- inplace\_merge() 1022
- input\_iterator\_tag 1032
- insert()
  - basic\_string 1128
- insert\_iterator 1036, 1039
  - Operationen 1040
- Insertor 1011, 1039
- Instanziierung
  - explizite 804
  - manuelle Kontrolle 804
  - Punkt der 730
  - Templates 725, 801
  - Template-Funktionen 803
  - Zeitpunkt 803
- int
  - implizites 1382
- INT\_MIN 1256
- integrale Typen 151
- integrierte Typen 53, 151
- intelligente Zeiger 598
- internal 1180, 1183
- Internal Program Representation (IPR) 834
- International 1228
- Internationalisierung 1199
- interne Bindung 457
- Interpunktion
  - Facetten 1227
  - Geldbeträge 1227
  - Zahlen 1221

- interrupt 1316
- interrupted 950
- Interval 1275
- invalid\_argument 950, 1127, 1323
- invalid\_seek 950
- Invarianten 63, 375, 524
  - bewahren 562
  - erzwingen 393
  - partiell spezifizierte 564
  - Ressourcen- 563
  - Ressourcen-Handles 81
- io\_errc 943, 952
- io\_error 950
- <iomanip> 931
  - Manipulatoren 1184
- <ios> 931, 1175
- ios\_base 1175, 1176
  - fmtflags 1180
  - Konstanten 1176
  - Locales 1182
  - Modi 1176
  - Operationen 1179
- <iosfwd> 931, 1161
- iostream 936
  - ios\_base 1176
  - Status 1166
- <iostream> 931
- iota() 1275
- IPR (Internal Program Representation) 834
- is()
  - ctype 1236
- is\_abstract 1101
- is\_a\_directory 950
- isalnum() 273, 1116, 1247
- isalpha() 273, 1115, 1247
- is\_arithmetic 1100
- is\_array 1099
- is\_assignable 1101
- is\_base\_of 1103
- isblank() 1115, 1247
- is\_class 1099
- iscntrl() 1116, 1247
- is\_compound 1100
- is\_const 1100
- is\_constructible 1101
- is\_convertible 1103
- is\_copy\_assignable 863, 1101
- is\_copy\_constructible 1101
- isctype() 1156
- is\_default\_constructible 863, 1101
- is\_destructible 1101
- isdigit() 273, 1115, 1247
- is\_empty 1100
- is\_enum 1099
- is\_error\_code\_enum 949
- is\_floating\_point 1099
- is\_function 1099
- is\_fundamental 1100
- isgraph() 1116, 1247
- is\_heap() 1025
- is\_heap\_until() 1025
- is\_integral 1099
- is\_literal\_type 1100
- is\_lock\_free() 1298
- islower() 1116, 1247
- is\_lvalue\_reference 1099
- is\_member\_function\_pointer 1099
- is\_member\_object\_pointer 1099
- is\_member\_pointer 1100
- is\_move\_assignable 1101
- is\_move\_constructible 1101
- is\_nothrow\_assignable 1102
- is\_nothrow\_constructible 1102
- is\_nothrow\_copy\_assignable 1102
- is\_nothrow\_copy\_constructible 1102
- is\_nothrow\_default\_constructible 1102
- is\_nothrow\_destructible 1103
- is\_nothrow\_move\_assignable 1103
- is\_nothrow\_move\_constructible 1102
- is\_object 1100
- is\_partitioned() 1015
- is\_pod 848, 1100
- is\_pointer 1099
- is\_polymorphic 1100
- is\_polymorphic<T> 843
- isprint() 1116, 1247
- ispunct() 1116, 1247
- is\_reference 1100
- is\_rvalue\_reference 1099
- is\_same 1103
- is\_scalar 1100
- is\_signed 1101
- is\_sorted() 1018
- is\_sorted\_until() 1018
- isspace() 273, 1115, 1247
- is\_standard\_layout 1100
- is\_steady 1095

- istream 101, 1168
    - Ausnahmen 1226
    - Manipulatoren 1184
    - sentry 1168
    - verketten 1169
  - <istream> 931, 1168
  - istreambuf\_iterator 1195
  - istream\_iterator 116
  - istringstream 276, 1164
  - is\_trivial 1100
  - is\_trivially\_assignable 1102
  - is\_trivially\_constructible 1102
  - is\_trivially\_copyable 1100
  - is\_trivially\_copy\_assignable 1102
  - is\_trivially\_copy\_constructible 1102
  - is\_trivially\_default\_constructible 1102
  - is\_trivially\_destructible 1102
  - is\_trivially\_move\_assignable 1102
  - is\_trivially\_move\_constructible 1102
  - is\_union 1099
  - is\_unsigned 1101
  - isupper() 1116, 1247
  - is\_void 1099
  - is\_volatile 1100
  - isxdigit() 1115, 1247
  - ISO
    - 4217 (Währungscodes) 1228
  - ISO/IEC 10646 157
  - ISO/IEC 14882
    - 2011 147
  - iter\_swap() 1017
  - Iteration
    - Klassen 857
    - Maps 987
    - Templates 856
  - iterator 966
  - <iterator> 930, 934, 1032, 1036, 1187
  - Iteratoren 108, 113, 1029
    - Abstand 1034
    - Adapter 1036
    - Ausgabe- 1031
    - basic\_string 1127
    - begin() 1041
    - Bezeichner 929
    - bidirektionale 1031
    - Container 969
    - Eigenschaften 1033
    - Einfüge- 1039
    - Eingabe- 774, 1031
    - end() 1041
    - Ende des Streams 1195
    - Forward 1031
    - Inserter 1039
    - istreambuf\_iterator 1195
    - istream\_iterator 116
    - Kategorien 1031
    - Konzepte 1032
    - list 116
    - make\_move\_iterator() 1040
    - mit wahlfreiem Zugriff 135, 1031
    - Operationen 1035
    - ostreambuf\_iterator 1196
    - ostream\_iterator 116
    - pair 138
    - Puffer 1194
    - Random Access 1031
    - raw\_storage\_iterator 1086
    - regex\_iterator 1152
    - regex\_token\_iterator 1153
    - reguläre Ausdrücke 1152
    - reverse\_iterator 1036
    - Slices 1269
    - Streams 116, 1187
    - Tag-Dispatching 1033
    - Tags 1032
    - Traits 1032
    - Typen 115
    - vector 115
    - Verschiebe- 1040
    - Vorwärts- 135, 1031
  - iterator\_traits 135, 850
  - Iteratoreigenschaften 1033
  - Spezialisierungen 1033
  - itoa() 352
  - lval\_box 663
  - lword() 1179
- ## J
- Java
    - Ratschläge 24
  - JavaScript
    - reguläre Ausdrücke 1136
  - join() 1312
  - thread 1308
  - joinable() 1308, 1311

**K**

- Kanji 1239
- Kapazität
  - Container 969
- Katakana 1239
- Kategorien
  - Facetten 1203
  - Iteratoren 1031
- key\_compare 966
- key\_equal 966
- key\_type 966
- key (Schlüssel) 109
- kill 1316
- kill\_dependency() 1297
- Klammern 284
- Klassen 55, 68, 487
  - abgeleitete 74, 625, 626
  - abstrakte 74, 647
  - Aufzählungs- 57
  - Basis- 74
  - facet 1209
  - Grundlagen 488
  - Gültigkeitsbereiche 1381
  - Invarianten 524
  - Iteration 857
  - lval\_box 663
  - konkrete 68, 509, 517
  - Konstruktoren 494
  - locale 1202
  - Member-Funktionen 489, 629
  - Namespaces 429
  - Rekursion 857
  - sentry 1168
  - Standardelemente 559
  - Strukturen 224
  - Sub- 74, 625
  - Super- 74, 625
  - Unions 233
  - Vec 106
  - vector<bool> 1060
  - Vererbung 74
  - verschachtelte 508
  - Zugriffskontrolle 491
- Klassendefinitionen 492
- Klassendeklarationen 492
- Klassenhierarchien 77, 631
  - Basisklassen wiederholt verwenden 682
  - entwerfen 663
  - Implementierungsvererbung 664
  - linearisieren 834
  - Navigation 693
  - Parameter 822
- Klasseninvarianten 63, 524
- Klassenobjekte
  - Initialisierung 529
- Klassen-Templates
  - Member 730
  - Template-Klassen 724
- Klassifizierung
  - Locale-abhängige 1247
  - Zeichen 1115
- Komfortschnittstellen 1247
- Kommentare 45, 259
- Kompatibilität
  - C/C++ 1376
- Kompilierung
  - separate 59
- komplexe Zahlen 140, 580, 1259
- Komplexität
  - Algorithmen 1005
- Komposition
  - Auswahl 445
  - Namespaces 442, 444
- Kompositoren 920
- konkrete Klassen 68, 509
- konkrete Typen 73, 509, 517
- konservative Collectoren 1085
- Konsistenz
  - sequenzielle 1292
- Konstanten 48, 287
  - Bindung 1381
  - DBL\_MAX 294
  - FLT\_MAX 294
  - Formate 1180
  - Hardwaregrößen 148
  - ios\_base 1176
  - magische Zahlen 289
  - numeric\_limits 148
  - reguläre Ausdrücke 1141
  - symbolische 289
  - Zeichen 101
- Konstanz
  - logische 501
  - physische 501
- Konstruktion 328, 710
  - Locales 1207
  - thread 1310



- Konstruktoren 56, 494, 630
  - Argumente 494
  - Auflösung mehrdeutiger 537
  - aufrufen 527
  - basic\_regex 1142
  - basic\_string 1121
  - bitset 1056
  - Container 967
  - Container, assoziative 982
  - delegieren 543
  - delegierende 544
  - Destruktoren 523
  - explicit 496, 583
  - Initialisierer 494
  - Initialisierung 531, 533
  - Initialisierung von Mitgliedern 541
  - Invarianten 524
  - Klasseninvarianten 524
  - Kopier- 82
  - Regeln 495
  - Standard- 69, 534
  - Strukturen 224
  - Templates 734
  - valarray 1261
  - vererben 643
  - Verschiebe- 84, 522
  - virtuelle 646, 674
  - weiterleitende 544
- kontextuelle Schlüsselwörter
  - final 640
  - override 639
- Kontravarianz 660
- Konvertierungen 293, 582
  - atof() 931
  - atoi() 931
  - auto 48
  - boolesche 295
  - complex 1259
  - c\_str() 1124
  - C-Strings 931, 1362
  - einschränkende 47, 175, 292, 293, 358, 533, 1259
  - ERANGE 1127
  - errno 1127
  - Funktionszeiger 364
  - Gleitkomma/Ganzzahl 296
  - Gleitkommatypen 294
  - Heraufstufungen 293
  - implizite 292
  - in() 1240
  - integrale 294
  - int zu string 353
  - mehrdeutige 591
  - Member-Funktionen 621
  - messages 1243
  - nicht einschränkende 329
  - numerische 1125
  - Operanden 584
  - Operatoren 588
  - Operatoren, explicit 590
  - out() 1241
  - Referenzen 295
  - Regeln 592
  - Stream-Puffer 1250
  - String- 1248
  - Templates 826
  - Uhrzeit 134
  - Zeichen 1248
  - Zeiger 295
  - Zeiger-auf-Member 295
- Konvertierungsfunktionen 827
- Konvertierungsoperatoren 827
- Konzepte 727, 762
  - Ad-hoc- 767
  - Einschränkungen 766
  - Enable\_if 863
  - erkennen 763
  - Iteratorkategorien 1032
  - konkret machen 768
  - mehrere Argumente 773
  - Ordered 765
  - Prädikate 727
  - Wert- 775
- Kopieren 81, 521, 547, 548
  - Basisklassen 550
  - Bedeutung 551
  - beim Schreiben 553
  - Container 81
  - Copy-On-Write 553
  - flaches 552
  - memberweises 81
  - Objekte 490
  - Slicing 554
  - tiefes 552
- Kopierinitialisierung 497, 539
- Kopierkonstruktoren 82
- Kopiersemantik 1262
- Kopierzuweisungen 82, 522

- Kopplung 633
- Kosinus 1257
- Kovarianz
  - Rückgabetypen 645
- kritische Abschnitte 1320
- Kurzschlussauswertung 284
  
- L**
  
- labs() 1258
- laguerre() 1258
- Lambda-Ausdrücke 91, 317
  - Alternativen 319
  - Argumente 325
  - Aufruf 325
  - Einführung 321
  - Erfassung 321
  - Erfassungsliste 318
  - Implementierungsmodelle 318
  - Lebensdauer 323
  - mutable 324
  - Rückgabe 325
  - this 324
  - Typen 325
  - veränderbare 324
- Lambda Introducer (Lambda-Einführung) 321
- Lambdas *siehe* Lambda-Ausdrücke
- Lastfaktor 992
- Laufzeit
  - linear amortisierte 965
  - Übersetzungszeit 853
- Laufzeitperformance 841
- Laufzeittypinformationen
  - RTTI (Run-Time Type Information) 694
- Lazy Evaluation 503
- lazy match (faule Übereinstimmung) 1140
- ldiv() 1258
- Lebensdauer 522
  - Lambda-Ausdrücke 323
  - Objekte 183
- leere Anweisungen 246
- Leerzeichen 1115
- left 1180, 1183
- legendre() 1258
- length() 1118, 1123, 1156, 1240, 1241
- length\_error 1122
- less 1042
- less\_equal 1042
  
- lexicographical\_compare() 1026
- lexikografisch
  - Permutationen 1016
  - Vergleichen 1026
- Lexikografische Ordnung 772
- Lifting 758
- <limits> 135, 144, 294, 932
- \_\_LINE\_\_ 371, 397
- linear amortisierte Zeit 965
- lineare Gleichungen 914
  - testen 917
- Line Feed 156
- Linker 456
- linksassoziativ 281
- Linksschieben 1058
- Lisp
  - Templates 841
- list 107
  - Iteratoren 116
  - Operationen 979
  - splice() 980
- <list> 930
- Listen 313
  - Argumente 349
  - doppelt verkettete 107
  - einfach verkettete 979
  - forward\_list 979
  - Implementierungsmodell 313
  - nichtintrusive 650
  - Operationen 972
  - qualifizierte 315
  - unqualifizierte 315
  - verkettete 979
- Listeninitialisierung 175
- Literale
  - benutzerdefinierte 604, 887
  - Einheiten 886
  - Ganzzahl- 158
  - Gleitkomma- 160
  - Präfixe 161
  - Strings 45
  - Suffixe 161
  - Unicode- 195
  - Zeichen- 156
- llabs() 1258
- lldiv() 1258
- load() 1298
- Loader 456
- local\_iterator 966

- locale 1202
  - Operationen 1182
- <locale> 932, 1202
  - Facetten 1215
- Locales 1159, 1199
  - Ausnahmen 1202
  - benannte 1204
  - Codeseiten 1205
  - Dezimaltrennzeichen 1221
  - facet 1209
  - Facetten 1201
  - Formatierung, numerische 1220
  - Internationalisierung 1199
  - Interpunktion 1221
  - ios\_base 1182
  - konstruieren 1207
  - Lokalisierung 1199
  - messages 1243
  - Microsoft 1205
  - POSIX 1205
  - reguläre Ausdrücke 1143
  - Standardfacetten 1215
  - strftime() 1367
  - Strings vergleichen 1208
  - time\_get 1233
  - verwenden 1214
  - Zeichencodekonvertierungen 1248
  - Zeichenklassifizierung 1235
  - Zeichenkonvertierungen 1248
  - Zugriff auf Facetten 1210
- localtime() 1364
- lock() 1320, 1323, 1327, 1328
- lock\_guard 1319, 1324
- locks (Sperrern) 121, 1288
- log() 140, 1257
- log10() 1257
- Logarithmus
  - dekadischer 1257
  - natürlicher 1257
- logical\_and 1042
- logical\_not 1042
- logical\_or 1042
- logische Konstanz 501
- lognormal\_distribution() 1283
- lokale Namen 343
- Lokalisierung 1199
- longjmp()
  - Ausnahmen 388
- LONG\_MAX 1256

- lookup\_classname() 1156
- lookup\_collatename() 1156
- loop fusion 918
- lt() 1117
- L-Werte 14, 182
- L-Wert-Referenzen 208

## M

- magische Zahlen 289
- main() 45, 479
  - argc 276
  - Argumente 276
  - argv 276
  - Namespaces 437
- make\_exception\_ptr(e) 938
- make\_heap() 1025
- make\_move\_iterator() 1040
- make\_pair() 138, 1062
- make\_ready\_at\_exit() 1340
- make\_shared() 1069, 1072
  - forward() 1110
- make\_signed 1105
- make\_tuple 872
- make\_tuple() 1064
- make\_unique() 744
- make\_unsigned 1105
- Makros 367
  - Argumente 368
  - assert() 394
  - ATOMIC\_FLAG\_INIT 1304
  - \_\_cplusplus 1384
  - C++-Schlüsselwörter 1380
  - <cstdlib> 351
  - #define 368
  - #endif 370
  - \_\_FILE\_\_ 397
  - Grenzwerte 1256
  - #ifdef 370
  - Include-Wächter 367
  - \_\_LINE\_\_ 397
  - NDEBUG 394
  - NULL 190
  - ## (Operator) 369
  - Operatoren 369
  - Präprozessor 370
  - rekursive 368
  - Übersetzung, bedingte 370
  - #undef 370

- vordefinierte 371
- Zugriff auf Argumente 351
- malloc() 1363
- vector 976
- Manipulatoren 1174
- benutzerdefinierte 1185
- <iomanip> 1184
- istream 1184
- <ostream> 1183
- Standard- 1182
- map 109
- Member-Typen 982
- Template-Argumente 982
- <map> 930
- Maps
- ungeordnete 987
- mapped\_type 966
- MapReduce 1352
- mask 1235
- mask\_array 1260
- Maßeinheiten 883
- match\_any 1147
- match\_continuous 1147
- match\_not\_bol 1147
- match\_not\_bow 1147
- match\_not\_eol 1147
- match\_not\_eow 1147
- match\_not\_null 1147
- match\_prev\_avail 1147
- match\_results 1143
- Matching Engine 1141
- Mathematik 140
- mathematische Standardfunktionen 1257
- Matrix
- Addition 901
- Anforderungen 893
- Anwendungen 891
- arithmetische Operationen 900
- Dreiecksmatrix 914
- Implementierung 905
- Indizierung 897
- Konstruktion 896
- lineare Gleichungen 914
- Listeninitialisierung 908
- Multiplikation 903
- Multiplizieren 1273
- nulldimensionale 913
- Pivotelement 916
- Slices 897, 905
- Speicherlayout 906
- Templates 894
- Überblick 894
- Zuweisung 896
- Matrix\_ref 907
- Matrix\_slice 905
- Untermatrizen 907
- max() 1027
- duration\_values 1097
- min() 1092
- valarray 1264
- Zufallszahlen 1281
- max\_element() 1027
- max\_length() 1240
- max\_size() 1123
- Maximum 1026
- Mehrbytezeichen 195
- Mehrdeutigkeiten
- auflösen 746
- Auflösung 678
- Konvertierung 591
- Mehrfachschnittstellen 676
- Mehrfachvererbung 669, 698
- Vorzüge 675
- Zugriffskontrolle 655
- Meldungen 1243
- von anderen Facetten 1246
- Member 55, 219
- abgeleitete 660
- Friends 620
- geschützte 653
- Initialisierung 542
- Initialisierungslisten 541
- Klassen-Templates 730
- protected 653
- Punktoperator 220
- statische 506
- Templates 733
- Typen 732
- Zeiger 657
- Zugriff 505
- Zuweisung 542
- Member-Funktionen 489, 629
- const 500
- Date 512
- Deklaration 335
- friend 619
- konstante 500
- Konvertierung 621

- statische 506
  - Templates 731
  - Member-Klassen 508
  - Member-Typen
    - Container 966
  - memchr() 1363
  - memcmp() 1362
  - memcpy() 1362
  - mem\_fn() 1043, 1046
  - mem\_fun() 1044
  - mem\_fun\_ref() 1044
  - memmove() 1362
  - <memory> 930
    - fill\* 1085
    - unique\_ptr 1066
  - Memory Barrier 1304
  - memory\_order\_acq\_rel 1296
  - memory\_order\_acquire 1296
  - memory\_order\_consume 1296
  - memory\_order\_relaxed 1296
  - memory\_order\_release 1296
  - memory\_order\_seq\_cst 1296
  - memset() 1363
  - Mengen
    - Algorithmen 1023
    - Sequenzen 1023
  - merge() 980, 1022
  - messages 1203, 1215, 1243
  - message\_size 950
  - Metaprogramme 350
  - Metaprogrammierung 135, 841
  - Metazeichen 1136
  - Methoden 634
  - min() 1026
    - duration\_values 1097
    - valarray 1264
    - zero() 1092
    - Zufallszahlen 1281
  - min\_element() 1027
  - Minimum 1026
  - minmax() 1027
  - minmax\_element() 1027
  - minus 1043
  - Mischen 1014
  - mismatch() 1008
  - mixed-mode arithmetic 581
  - Mixin 692
  - mktime() 1364
  - modf() 1257
  - modifizierbarer L-Wert 182
  - Modifizierer
    - remove\_reference 1105
  - Modularisierung 435
  - Modularität 58, 425
  - Module
    - Namespaces 436
  - Modulo 266
  - modulus 1043
  - Modus 1356
  - monetary 1203, 1215
  - money\_get 1215, 1227, 1231
  - money\_punct 1215
    - Member-Funktionen 1228
  - money\_put 1215, 1227, 1230
  - move() 213, 558, 1011, 1072, 1109, 1117, 1121, 1178, 1327, 1339, 1342, 1346
    - swap() 1109
    - thread 1308
  - move\_backward() 1011
  - move\_if\_noexcept() 1109
  - move\_iterator 1036
  - Moved-from-Zustand 548
  - multiplies 1043
  - Multiplikation
    - complex 1259
    - Matrix 903
  - Multiplizieren
    - Matrizen 1273
  - Mustermaschine 1137
  - mutable 502
    - Lambda-Ausdrücke 324
  - mutex 936, 1319
    - Operationen 1320
  - mutex() 1327
  - <mutex> 128, 933
  - Mutexe 1319
    - belegen 1319
    - Fehler 1322
    - freigeben 1319
- ## N
- Nachbedingungen 361
  - Namen 170
    - abhängige 806, 807
    - dateilokale 459
    - Deklarationen 166
    - Groß-/Kleinschreibung 170

- lokale 343
- mehrere deklarieren 169
- Namespaces 324
- unabhängige 807
- verdeckte 173
- vollständig qualifizierte 428
- Namensauflösung
  - argumentabhängige 431
- Namensbereiche *siehe* Namespaces
- Namensbindung 801, 805
- Namenskonflikte 426
- namespace pollution 241
- Namespaces 60, 426
  - Alias 443
  - C-Header 452
  - Gültigkeitsbereiche 429
  - Implementierungen 438
  - Inline- 448
  - Klassen 429
  - Komposition 442, 444
  - main() 437
  - Module 436
  - Namen 324
  - offene 434
  - Operatoren 578
  - placeholders 1044
  - std 929
  - std::chrono 134
  - Templates 814
  - this\_thread 1315
  - Überladen 446
  - unbenannte 451
  - using 61, 429
  - verschachtelte 450
  - zugeordnete 433
- Namespace-Verschmutzung 241
- narrow() 1178, 1236
- narrow\_cast<>() 294
- native\_handle() 1308, 1320, 1323
- native\_handle\_type 1308, 1320, 1323
- NDEBUG 394
- Nebeneffekte
  - constexpr 341
- Nebenläufigkeit 124, 1287
  - Initialisierung 480
  - Speichermodelle 1289
  - Task-basierte 1336
  - Threads 1307
  - Umordnung 1291
- neg\_format() 1228
- negate 1043
- negative\_binomial\_distribution() 1282
- negative\_sign() 1228
- nested\_exception 939
- network\_down 950
- network\_reset 950
- network\_unreachable 950
- new
  - Arrays von Objekten 308
  - Handler 310
  - nothrow 312
  - Überladen 310
- <new> 932, 934
- new T 936
- Newline 156, 193, 1115
- next() 1036
- nicht gefunden 113
- nichtintrusive Listen 650
- noboolalpha 1182
- no\_buffer\_space 950
- no\_child\_process 950
- noconv 1241
- noexcept 335, 385, 400
  - bedingt 400
- no\_link 950
- no\_lock\_available 950
- no\_message 950
- no\_message\_available 950
- none 1203
- none() 1059
- none\_of() 1006
- non-greedy match (zurückhaltende Übereinstimmung) 1140
- no\_protocol\_option 950
- noreturn 335, 342
- normal\_distribution 141
- normal\_distribution() 1283
- Normalverteilungen 1283
- noshowbase 1182
- noshowpoint 1183
- noshowpos 1183
- noskipws 1169, 1183
- no\_space\_on\_device 950
- no\_state 1344
- no\_stream\_resources 950
- nosubs 1141
- no\_such\_device 950
- no\_such\_device\_or\_address 950

no\_such\_file\_or\_directory 950  
 no\_such\_process 950  
 not 282  
 not1() 1043  
 not2() 1043  
 not\_a\_directory 950  
 not\_a\_socket 951  
 not\_a\_stream 951  
 not\_connected 951  
 not\_enough\_memory 951  
 not\_eof() 1118  
 not\_eq 282  
 not\_equal\_to 1042  
 not\_supported 951  
 nothrow 312  
 No-throw-Garantie 387  
 nothrow\_t 312  
 nounitbuf 1183  
 nouppercase 1183  
 now() 1089  
 nth\_element() 1018  
 NULL 53, 190  
 Nullooperationen  
   – Streams 1166  
 Null-Overhead-Prinzip 11  
 nullptr 52, 189, 295  
   – dynamic\_cast 695  
 num\_get 1215, 1225  
 num\_put 1215, 1222  
 numeric 1203, 1215  
 <numeric> 140, 932  
 numeric\_limits 135, 148, 294, 296, 1254  
 numerische Grenzen 1253  
 numpunct 1203, 1215, 1221

## O

Objekte 182  
   – Ausrichtung 165  
   – Besitz 1065  
   – Erstellung örtlich begrenzen 674  
   – Funktions- 597  
   – funktionsähnliche 597  
   – Größe 135  
   – initialisieren 490  
   – Initialisierer 174  
   – kopieren 81, 490  
   – Lebensdauer 183  
   – POD 228

  – sizeof 135  
   – Speicher konvertieren in 549  
   – sperrbare 1326  
   – temporäre 285  
   – übergeben 576  
   – Verschieben 81  
   – verschränkte 552  
   – vertauschen 1017  
 objektorientierte Programmierung 625  
 oct 1180, 1183  
 Oder 299  
 ODR (One Definition Rule) 455  
   – constexpr 341  
 Off-by-one-Fehler 614  
 ofstream 1162  
 ok 1241  
 Oktalzahlen 156  
 once\_flag 1329  
 One Definition Rule (ODR) *siehe*  
   Eine-Definition-Regel  
 On\_heap 845  
 O-Notation 965  
 open()  
   – messages 1243  
 OpenMP 1294  
 Operanden  
   – Konvertierung 584  
 operation\_canceled 951  
 operation\_in\_progress 951  
 operation\_not\_permitted 951, 1323  
 operation\_not\_supported 951  
 operation\_would\_block 951  
 Operationen  
   – acquire 1296  
   – Argumente 786  
   – Arithmetik 141  
   – arithmetische Matrix- 900  
   – atomare 1292, 1295, 1298, 1302  
   – Ausgabe- 1171  
   – basic\_ios 1177  
   – basic\_streambuf 1189  
   – bitset 1058  
   – CAS- 1300  
   – complex 1259  
   – consume 1296  
   – C-Strings 1361  
   – Ein-/Ausgabe 1168  
   – Eingabe- 1168  
   – explizite Standard- 560

- fmtflags 1180
- forward\_list 979
- insert\_iterator 1040
- ios\_base 1179
- Iteratoren 1035
- komplexe Zahlen 141
- Komplexität 964
- list 979
- Listen 972
- locale 1182
- logische 151
- mutex 1320
- regex\_traits 1156
- release 1296
- shared\_ptr 1071
- Skalar- 901
- Standard- 561
- stringstream 1164
- sub\_match 1143
- Synchronisierung 1296
- unterdrücken 86
- valarray 1264
- verketteten 1169
- verschmolzene 918
- operator[] 595
- Operatoren
  - Adress- 187
  - & (Adresse) 187
  - alignof() 165
  - alternative Darstellungen 282
  - arithmetische Konvertierungen 297
  - Auswertung 249
  - benutzerdefinierte Typen 576
  - Bereichsaflösungs- 173
  - binäre 574
  - Bindungsstärke 281
  - bitweise 299
  - Container 963
  - Deklarations- 52
  - Deklaratoren 168
  - Dekrementieren 301
  - Dereferenzierung 187
  - Dereferenzierungs- 598
  - diverse 299
  - dynamic\_cast 695
  - einstellige 574
  - Ergebnistypen 282
  - Funktionen 573
  - Funktions- 596
  - [] (Indexoperator) 190
  - Inkrementieren 301
  - Konvertierungs- 588, 827
  - Literal- 604
  - logische 249, 299
  - Makros 369
  - Member-/Nicht-Member- 580
  - Modulo 266
  - Namespaces 578
  - noexcept 400
  - Pfeil- 598
  - Pfeiloperator (->) 505
  - placement new 311
  - Platzierungs- 310
  - Punktoperator (.) 505
  - Rangfolge 284
  - relationale 1111
  - sizeof 46
  - spezielle 595
  - stream\_iterator 1188
  - typeid 71
  - überladene 516
  - unäre 574
  - vordefinierte Bedeutungen 575
  - Zusammenfassung 278
  - zweistellige 574
- Optimierung
  - leere Basisklasse 832, 868, 1077
  - valarray 1266
- optimize 1141
- or 282
- OR 1058
- Ordered 765
- Ordnung
  - lexikografische 772
  - strenge schwache 766
  - totale 963
- Ordnungsfunktionen 110
- or\_eq 282
- Organisation
  - mehrere Header-Dateien 472
  - Quellcode 751
- ostream 1192
- <ostream> 931
  - Manipulatoren 1183
- ostream\_iterator 116
- ostreambuf\_iterator 1196
- ostringstream 1164
- out 1176



- out() 1241
  - codecvt 1240
- out\_of\_range 106, 399, 1127
- output\_iterator\_tag 1032
- overflow() 1192
- overloading (Überladen) 356
- override 75, 335, 636, 639
- owner\_before() 1072
- owner\_dead 951
- owns\_lock() 1327
  
- P**
- packaged\_task 409, 936, 1336, 1339
- pair 138, 1051, 1061
  - first 1062
  - second 1062
- pair() 1027
- Parallelität 1287
- param() 1281
- param\_type 1281
- Parameter 344
- Parameterpakete 875
  - Funktionen 876
- Parser 264
- partial 1241
- partial\_sort() 1018
- partial\_sort\_copy() 1018
- partial\_sum() 1271, 1274
- partielle Auswertung 1044
- partition() 1014, 1015
- partition\_copy() 1015
- partition\_point() 1015
- Partitionieren 1015
- Pattern Matcher 1137
- Pattern Matcher (Mustermaschine) 1137
- pbackfail() 1191
- pbase() 1192
- pbump() 1192
- permission\_denied 951
- Permutationen 1015
- Pfeiloperator (->) 505, 598
- pfind() 1349
- physische Konstanz 501
- physische Struktur 456
- piecewise\_constant\_distribution() 1285
- piecewise\_linear\_distribution() 1285
- Pivotelement 916
- Pivotisierung 916
  
- placeholders 1044
- placement new (Platzierungsoperator new) 311, 528
- Plain Old Data (POD) 228, 848
- Platzhalter 1044
- Platzierungsoperatoren 310, 528
- Platzierungssyntax 311
- plus 1043
- POD 228, 848, 1117
  - Ausnahmen 401
  - (Plain Old Data) 228
- pointer 966
- poisson\_distribution() 1283
- Poisson-Verteilungen 1283
- policy objects 90
- Polymorphe Typen 74
- Polymorphie 636, 821
  - dynamische 626
  - Laufzeit 626, 675
  - parametrische 757, 821
  - statische 626
  - Templates 757
  - Übersetzungszeit 626, 781
  - zur Laufzeit 821
  - zur Übersetzungszeit 821
- pop\_back() 527
  - basic\_string 1124
- pop\_heap() 1025
- Population 1277
- Portabilität 927
  - reguläre Ausdrücke 1138
  - Zeichenklassen 1138
- Portierbarkeit 44
- pos\_format() 1228
- positive\_sign() 1228
- POSIX
  - Locales 1205
- Potenzfunktion 1257
- pow() 1257, 1265
- pptr() 1192
- Prädikate 90, 118, 152, 848
  - check\_bounds() 911
  - Konzepte 727
  - Sequenzen 1006
  - Typeigenschaften 863, 1100
  - Typrelationen 1103
- Prädikatenlogik erster Ordnung 773
- Präfixe 161
- #pragma 372

- Pragmas 372
- Präprozessor 370
- precision() 1181
- prev() 1036
- primäre Ausdrücke 267
- primäres Template 795
- primäre Typprädikate 1099
- printf() 1356
  - Formatierung 1357
  - Typprüfung 1359
  - typsicher 873
- Prioritätswarteschlangen 996
- priority\_queue 996
- private 491, 649
- Probleme
  - ABA- 1300
- Programme 479
  - ausführbare 44
  - beenden 481
  - Befehlszeilenargumente 276
  - exit() 481
  - Hello, World! (Beispiel) 44
  - main() 45, 479
  - Quelldateien 44
- Programmiersprachen 1275
  - Entwurf 10
- Programmierstile 12
- Programmierung
  - generative 841
  - generische 757
  - MapReduce 1352
  - Mehrebenen- 841
  - Meta- 841
  - objektorientierte 625
  - prozedurale 43
  - Template-Meta- 841
  - wertorientierte 517
  - Zwei-Ebenen- 841
- promise 131, 936, 1336, 1337, 1338
- promise\_already\_satisfied 1344
- promotions (Heraufstufungen) 293
- protected 649
- protocol\_error 951
- protocol\_not\_supported 951
- Proxy
  - vector<bool> 1061
- Prozesse 1288
- pt() 1340
- pubimbue() 1189

- public 491, 649
- pubseekoff() 1189
- pubseekpos() 1189
- pubsetbuf() 1189
- pubsync() 1189
- Puffer 1188
  - Ausgabe- 1192
  - Eingabe- 1193
  - Iteratoren 1194
  - temporäre 1086
  - Überlauf 1173
  - Unterlauf 1173
- Punkt der Instanziierung 730, 810
- Punktoperator (.) 220, 505
- Punktprodukt 1274
- push\_back() 72, 419, 421, 527
  - basic\_string 1124
- push\_heap() 1025
- put() 1172, 1222
  - Geldbeträge 1230
  - Zeitpunkte 1232
- put area (Schreibbereich) 1190
- putc() 1361
- putchar() 1361
- put\_money() 1184
- pwd() 1179

## Q

- qsort() 930, 1367
- Quadratwurzel 1257
- Qualifizierung
  - explizite 428, 637
- Quantity 884
- Quellcode
  - Organisation 751
- Quelldateien 44
- Quellezeichensatz 149, 282, 1238
- queue 129, 996
- <queue> 930, 996
- quick\_exit() 482

## R

- R (Algorithmus) 1277
- Race Conditions 129, 1290
- RAII (Resource Acquisition Is Initialization) 375, 390, 526
  - lock\_guard 1326

- rand() 1367
- Rand\_double 1277
- Rand\_int 1277
- <random> 141, 932, 1275
- random-access iterators 135
- random\_access\_iterator\_tag 1032
- random\_device 1275, 1280
- random\_shuffle() 1014
- range-based for-loop 254
  - siehe auch* for, bereichsbasiert
- Rangfolge
  - Operatoren 284
- rank 1103
- ratio 1097
- <ratio> 930, 1097
- Ratschläge 37
- raw\_storage\_iterator 1036, 1086
- rbegin() 1127
- rdbuf() 1178, 1250
- rdstate() 1177
- read\_only\_file\_system 951
- ready 1343
- realloc() 1363
  - vector 976
- rechtsassoziativ 281
- Rechtsschieben 1058
- recursive descent 264
- recursive\_mutex 1319
- recursive\_timed\_mutex 1319, 1323
- reduce 1273
- reduction 1273
- reference 966
- Referenzen 206
  - Argumente 291, 345
  - constexpr 341
  - dynamic\_cast 697
  - Initialisierung 208
  - Konvertierungen 295
  - R-Wert- 211
  - auf Referenzen 214
  - Wrapper 1045, 1311
  - Zeiger 215
- Referenzkollaps 214
- Regeln
  - Eine-Definition- 341
  - Konstruktoren 495
  - Konvertierungen 292, 592
  - Übereinstimmung, längste 281, 1137
- regex 936, 1141
  - <regex> 139, 931
  - regex\_iterator 1135, 1152
  - regex\_match() 1135, 1147
  - regex\_replace() 1135, 1150
  - regex\_search() 1135, 1149
  - regex\_token\_iterator 1135, 1153
  - regex\_traits 1156
  - register\_callback() 1179
  - reguläre Ausdrücke
    - Backslashes 1139
    - ECMA 1136
    - Formatierung 1146
    - Formatierungsoptionen 1146
    - Funktionen 1147
    - Grenzzeichen 1139
    - grep 1140
    - Groß-/Kleinschreibung 1141
    - Gruppen 1140
    - Iteratoren 1152
    - JavaScript 1136
    - Konstanten 1141
    - Locales 1143
    - match\_results 1143
    - Metazeichen 1136
    - Portabilität 1138
    - regex 1141
    - regex\_iterator 1152
    - regex\_match() 1147
    - regex\_search() 1149
    - regex\_token\_iterator 1153
    - smatch 1135
    - sub\_match 1143
    - Token-Splitting 1155
    - Wiederholungen 1136
    - Zeichenklassen 1137
    - Zustandsmaschinen 1136
- Reguläre Ausdrücke 139
- reguläre Typen 765
- reinterpret\_cast 330
  - Funktionszeiger 364
- rein virtuelle Funktionen 74, 647
- Rekursion
  - Klassen 857
  - Templates 856
- rekursiver Abstieg 264
- rekursive Funktionen 338
- release 1296
- release() 1327
- remove() 1013

- remove\_all\_extents 1105
  - remove\_const 1104
  - remove\_cv 1104
  - remove\_extent 1105
  - remove\_pointer 1106
  - remove\_reference 1105
  - remove\_volatile 1104
  - rend() 1127
  - Rep (Representation, Darstellung) 1090
  - replace() 99, 1013, 1129
  - Requesting\_element() 911
  - Requesting\_slice() 911
  - reserve() 420, 1123
  - reset() 1072, 1340
    - bitset 1058
    - Zufallszahlen 1281
  - resetiosflags() 1184
  - resize() 419, 421, 1123
    - valarray 1265
  - resource\_deadlock\_would\_occur 951, 1323
  - resource\_unavailable\_try\_again 951, 1323
  - Ressourcen
    - Destruktoren 525
    - freigeben 389
    - Verwaltung 121, 388
  - Ressourcenbelegung ist Initialisierung 72, 122, 306, 375, 390, 526
  - Ressourcen-Handles 1066
    - vector 978
  - Ressourceninvarianten 563
  - Ressourcensicherheit 86
  - Ressourcenverwaltung 85, 522
  - result\_of 1106
  - Result\_of 1107
  - result\_out\_of\_range 951
  - result\_type 1281
  - rethrow\_exception() 938
  - rethrow\_if\_nested() 939
  - return 337
    - Schleifen 257
    - switch 251
  - return\_temporary\_buffer() 1086
  - reverse() 1013
  - reverse\_iterator 966, 1036
  - Reverse-Iteratoren 1036
  - rfind() 1130
  - Richtlinien
    - Argumente 1003
    - async() 1347
    - Hash 992
    - Start- 1347
  - Richtlinienobjekte 90
  - riemann\_zeta() 1258
  - right 1180, 1183
  - roher Speicher 1085
  - rohe Stringlitterale 139, 194
    - reguläre Ausdrücke 1139
  - Romaji 1239
  - rotate() 1014
  - Rot-Schwarz-Bäume 109, 981
  - round() 328, 1257
  - Row-major Order 906, 1267
  - RTTI
    - richtig und falsch verwendet 714
    - Run-Time Type Information 694
    - switch 714
  - Rückgabetypen 337
    - kovariante 645
    - Lockerung 645
    - Überladen 358
  - Rückgabewerte 337
    - return 337
  - Rückruffunktionen 317
  - Rückwärtsschritt 156
  - Runden 328
    - 4/5-Rundung 1257
  - 4/5-Rundung 1257
  - Rundungsfehler 1097
  - runtime\_error 399, 1202
  - R-Werte 14
    - forward() 1110
    - modifizierbarer L-Value 182
    - Referenz 84
    - this 504
    - Typumwandlungen 1109
    - vertauschen 1111
  - R-Wert-Referenzen 207
- ## S
- sample (Stichprobe) 1277
  - sbumpc() 1191
  - scan\_is() 1236
  - scan\_not() 1236
  - Scheduler 1315
    - Aushungern 1319
  - Schere, Stein, Papier 766

- Schleifen 49, 253
  - Arrays 51
  - break 257
  - continue 258
  - do 257
  - Endlos- 256
  - for 255
  - for, bereichsbasierte 254
  - goto 258
  - return 257
  - verlassen 257
  - while 50, 256
  - Zeiger 51
- Schleifenverschmelzung 918
- Schlüssel 109
- Schlüsselwörter 1380
  - alternative Darstellungen 282
  - C++ 171
  - kontextuelle 639
  - Spezifizierer 168
  - virtual 74, 634
- Schnittstellen 55
  - ABI (Application Binary Interface) 1083
  - Deklarationen 58
  - dynamic\_cast 701
  - Garbage Collection 1082
  - Implementierungen 440
  - Komfort- 1247
  - Mehrfach- 676
  - Modularisierung 435
  - Spezialisierungen 794
  - Templates 829
- Schnittstellenvererbung 79, 649, 668
- Schrittweite 1267
- scientific 1180, 1183
- Scoped 845
- <scoped\_allocator> 930
- scoped\_allocator\_adaptor 1081
- Scope *siehe* Gültigkeitsbereiche
- search() 1009
- search\_n() 1009
- second 1062
- seed 1278
- seekp() 1172
- Selbstreferenz 503
- Select 844, 852
- Selektoren
  - conditional 845
- Semantik des exklusiven Besitzes 1320
- sentry 1168
- Sequenzen 112, 1001, 1002, 1029
  - Duplikate entfernen 1012
  - Ende 113
  - halboffene 1029
  - Heap 1024
  - merge() 1022
  - nicht gefunden 113
  - partitionieren 1015
  - Permutationen 1015
  - sortieren 1018
  - Suchen 1009, 1018
  - Summen 1273
  - teilen 1015
  - zusammenführen 1022
- sequenzielle Konsistenz 1292
- Sequenzprädikate 1006
- set()
  - bitset 1058
- <set> 930
- setbase() 1184
- set\_difference() 1023
- setf() 1180
- setfill() 1184
- setg() 1191
- set\_intersection() 1023
- setiosflags() 1184
- set\_new\_handler() 310
- setp() 1191
- setprecision() 1184
- set\_rdbuf() 1178
- setstate() 1177
  - Ausnahmen 1226
- set\_symmetric\_difference() 1024
- set\_terminate() 407, 941
- set\_union() 1023
- setw() 1184
- SFINAE (Substitution Failure Is Not An Error)
  - 748, 864
- sgetc() 1191
- sgetn() 1191
- shadowed *siehe* verdeckt
- shallow copy (flache Kopie) 552
- share() 1342
- shared\_future 1336, 1345
- shared\_ptr 122, 1069
  - -> (Operator) 598
  - move() 1072
  - Operationen 1071

- Shell Sort 364
- shift()
  - valarray 1264
- Shift 1215, 1239
- short string optimization 607
- showbase 1180, 1182
- showmanyc() 1191, 1192
- showpoint 1180, 1182
- showpos 1180, 1183
- shrink\_to\_fit() 1123
- shuffle\_order\_engine 1279
- sicher abgeleitete Zeiger 1083
- SI-Einheiten 883
- signed char 153
- sin() 140, 1257
- Sinus 1257
- size() 1059, 1123
  - valarray 1264
- sizeof 46, 135
- sizeof() 222
- size\_type 966
- skipws 1180, 1183
- sleep\_for() 1315
- sleep\_until() 1315
- slice 1260
- slice\_array 1260, 1269
- Slices 895, 1267
  - Bereichsüberprüfungen 1269
  - Iteratoren 1269
  - Matrix 905
  - Matrizen 897
  - Schrittweite (stride) 1267
  - slice\_array 1269
  - verallgemeinerte 1270
- Slicing 554, 630
  - Kopieren 551
  - unique\_ptr 1068
- Slots
  - Vektoren 959, 975
- smart pointers 598
  - siehe auch* Zeiger, intelligente
- smatch 1135
- snext() 1191
- sort() 1018
  - bereichsbasiert 135
- sort\_heap() 1025
- Sortieren 1018
  - Groß-/Kleinschreibung 1220
  - Operatoren, relationale 962
  - Shell Sort 364
  - strenge schwache Ordnung 961
  - Vergleichsfunktionen 365
- Speicher
  - Ausrichtung 1106
  - beschaffen 309
  - C-Standardbibliothek 1362
  - dynamischer 54, 303
  - explizit darstellen 414
  - Frei- 54
  - get\_temporary\_buffer() 1086
  - Heap 54
  - konvertieren 549
  - nichtinitialisierter 1085
  - return\_temporary\_buffer() 1086
  - roher 1085
  - temporärer 1086
  - transaktionaler 1294
- Speicherdauer 1317
- Speicherklassen 184
- Speichermodelle 1289
  - carries\_dependency 1297
  - volatile 1305
- Speicherordnung 1292
  - sequenzielle Konsistenz 1292
- Speicherstellen 1290
- Speicherverwaltung 305
- sperrbare Objekte 1326
- Sperren 1288
  - doppelt überprüfte 1301
  - kritische Abschnitte 1320
  - mehrere 1328
  - Spinlocks 1303
- Spezialisierungen 725, 781, 802
  - benutzerdefinierte 791, 802
  - explizite 750, 802
  - generierte 802
  - Implementierungs- 795
  - iterator\_traits 1033
  - partielle 793
  - Reihenfolge 797
  - Schnittstellen 794
  - Template-Funktionen 798
  - Überladungen 798
  - vollständige 792
- spezielle mathematische Funktionen 1258
- Spezifizierer 168
  - decltype() 181

- final 640
- override 639
- sph\_bessel() 1258
- sph\_legendre() 1258
- sph\_neumann() 1258
- Spinlocks 1303
- splice() 980, 1321
- Sprachunterstützung 934
- sprintf() 1356
- sputback() 1191
- sputc() 1191
- sputn() 1191
- sqrt() 140, 1257
- srand() 1367
- <sstream> 277, 931
- stable\_partition() 1015
- stable\_sort() 1018
- stack
  - Allokatoren 995
  - Containeradapter 994
  - Deque 995
  - Überlauf 995
  - Unterlauf 995
- <stack> 930
  - Containeradapter 994
- Stack-Abwicklung 390, 398, 932
- stack frame 183
- Stack Unwinding 390, 398
- Standards
  - ECMA 1136
  - ISO 4217 (Währungscodes) 1228
  - ISO/IEC 10646 157
  - ISO/IEC 14882 147
  - undefiniert 148
- Standard-Allokatoren 1077
- Standardargumente 354
  - Funktions-Templates 790
  - Templates 789
- Standardbeziehungen
  - Templates 826
- Standardbibliothek 31
  - Ausnahmen 404, 936
  - Ausnahmenklassen 399
  - basic\_string 723
  - Beschreibung 929
  - C 1355
  - C++11-Ergänzungen 1373
  - Fehlerbehandlung 935
  - getline() 102
  - Header 97, 464, 929
  - Hilfskomponenten 134
  - istream 101
  - Komponenten 926
  - list 107
  - map 109
  - Namespaces 97
  - ostream 100
  - Sprachunterstützung 934
  - <stdexcept> 404
  - Stream-Ein-/Ausgabe 100
  - Suchbaum 109
  - Systemfehler 942
  - Überblick 96
  - unordered\_map 110
  - Zeit 134
- Standardfacetten 1215
- Standardfehlerausgabe 1161
- Standardfunktionen
  - mathematische 1257
- Standardkonstruktoren 69, 534, 562
- Standardoperationen 559, 561
  - verwenden 562
- Standard-Streams 1161
- Standardzeichenausgabe 1161
- Standardzeicheneingabe 1161
- Startrichtlinien 1347
- starvation (Aushungern) 1295, 1319
- state() 1249, 1250
- state\_not\_recoverable 951
- static 335
  - Bindung 457, 460
- static\_assert 64, 394, 941
- static\_cast 330
  - dynamic\_cast 700
- static\_pointer\_cast() 1073
- Status
  - gemeinsamer 1337
  - Streams 1175
- <stdbool.h> 1384
- \_\_STDC\_\_ 372
- \_\_STDC\_HOSTED\_\_ 371
- std::chrono 134
- \_\_STDC\_MB\_MIGHT\_NEQ\_WC\_\_ 372
- \_\_STDCPP\_STRICT\_POINTER\_SAFETY\_\_ 372
- \_\_STDCPP\_THREADS\_\_ 372
- <stdexcept> 404, 931
- stdio
  - Zeichenfunktionen 1360

- <stdio> 1355
- <stdlib.h> 1285
- steady\_clock 1089, 1096
- Steuerelemente 693
- Steuerungsstrukturen 852
  - Auswahl 852
- Steuerzeichen 1116
- Stichprobenverteilungen 1284
- stillschweigende Unterschiede 1378
- STL (Standard Template Library) 31, 955
- stod() 1126
- stof() 1126
- stoi() 1125
- stol() 1126
- stold() 1126
- stoll() 1126
- store() 1298
- stoul() 1126
- stoull() 1126
- strcat() 1361
- strchr() 1362
- strcmp() 1361
- strcpy() 1361
- Streams
  - Ausgabe- 1192
  - Ausgabeoperationen 1171
  - Ausnahmen 1167
  - bad() 1163
  - basic\_ios 1162, 1175
  - basic\_streambuf 1189
  - Datei- 1162
  - Ein-/Ausgabe 1159
  - Eingabe- 1193
  - Eingabeoperationen 1168
  - Ereignisse 1179
  - exceptions() 1167
  - Fehlerbehandlung 1166
  - Fehlermeldungen 274
  - Formatierung 1179
  - <fstream> 1162
  - ios\_base 1175
  - istreambuf\_iterator 1195
  - istringstream 276
  - Iteratoren 116, 1187
  - Locales 1182
  - Manipulatoren 1174, 1182
  - Modi 1163
  - Nulloperationen 1166
  - ostreambuf\_iterator 1196
  - Puffer 1188
  - sentry 1168
  - showmanyc() 1192
  - Standard- 1161
  - Status 1163, 1175
  - Strings 1164
  - Überlauf 1190
  - verketteten 1169
  - virtuelle Ausgabefunktionen 1173
- stream\_iterator
  - Operatoren 1188
- stream\_timeout 951
- streambuf 1188
- <streambuf> 931, 1188
- strenge schwache Ordnung 766, 961
- strftime() 1365
  - Formatierung 1366
- String 98, 936, 1115
  - Anführungszeichen 194
  - Backslash 194
  - C- 53, 1361
  - c\_str() 1124
  - Darstellung 609
  - Funktionen, ergänzende 611
  - Hilfsfunktionen 615
  - Klasse als Übung 607
  - Konvertierungen 1124
  - leere 193
  - lexikografische Ordnung 772
  - Mehrbytezeichen 195
  - Member-Funktionen 612
  - Operationen 608
  - Optimierung für kurze 607
  - POD 1117
  - replace() 99
  - rohe Zeichen- 194
  - Streams 1164
  - substr() 99
  - Teilstrings 1132
  - Templates 722
  - vector 978
  - vergleichen 1208, 1217
- <string> 931
- Stringliterale 45, 192
  - Anführungszeichen 194
  - rohe 139, 194
- string\_stream 1126
- stringstream 1164
  - Operationen 1164



- strlen() 1361
  - strncat() 1361
  - strncmp() 1362
  - strncpy() 1361
  - strpbrk() 1362
  - strrchr() 1362
  - strstr() 1362
  - strtod() 1362
  - strtof() 1362
  - strtol() 1362
  - strtold() 1362
  - strtoll() 1362
  - strtoul() 1362
  - strtoull() 1362
  - struct 219, 492
    - Initialisierung 220
  - Strukturen 53, 219
    - array 226
    - Arrays 225
    - Gültigkeitsbereiche 1381
    - Klassen 224
    - Konstruktoren 224
    - Layout 221
    - Namen 222
    - physische 456
    - tm 1364
  - student\_t\_distribution() 1284
  - Subklassen 74, 625
  - sub\_match
    - Operationen 1143
  - substitution failure 748
  - substr() 99, 1132
  - Subtraktion
    - complex 1259
    - Zeiger 199
  - Suche 1018
    - argumentabhängige 815
    - binäre 1021
    - parallele 1349
    - regex\_search() 1149
    - Rot-Schwarz-Bäume 109
    - Sequenzen 1009
  - Suffixe 161
  - Suffixrückgabetypen 337
  - sum()
    - valarray 1264
  - Summen
    - Sequenzen 1273
  - sungetc() 1191
  - Superklassen 74, 625
  - swap() 1017, 1062, 1072, 1110, 1265, 1327, 1339
    - basic\_string 1124
    - move() 1109
    - R-Werte 1111
    - thread 1308
    - valarray 1264
  - swap\_ranges() 1017
  - switch 250
    - break 251
    - default 251
    - goto 258
    - return 251
    - RTTI 714
  - Sync\_queue 1333
  - sync\_with\_stdio() 1176
  - Synchronisierung
    - Operationen 1296
    - Threads 1296
  - Syntaxanalyse 264
  - system() 1367
  - system\_clock 1089, 1096
  - Systeme international d'unités 883
  - system\_error 943, 1322
    - Ausnahmen 946
  - <system\_error> 931, 942
  - Systemfehler 942
  - Systemprogrammierung 11
- ## T
- Tags
    - Iteratoren 1032
  - Tag-Dispatching 136, 1033
  - tagged Union 235
  - tan() 1257
  - Tangens 1257
  - Taschenrechner 263, 435
    - Fehlerbehandlung 274
    - Header 275
    - Rahmenprogramm 274
  - Tasks 125, 1287, 1307
    - asynchrone 133
    - future 1337
    - Kommunikation 130
    - packaged\_task 1339
    - promise 1337
  - Teiler
    - größter gemeinsamer 1097

- Teilstrings 1132
- tellp() 1172
- Templates 87, 719, 781
  - Ableitung 821
  - Alias 750
  - Argumente 782
  - Argumentersetzung 747
  - array 226
  - atomic 1298
  - Ausdrucks- 920
  - basic\_string 1118
  - Beziehungen 826
  - Binden 753
  - conditional 845
  - Curiously Recurring Template Pattern (CRTP) 834
  - Daten-Member 730
  - definieren 724
  - Definition, bedingte 748
  - Definitionen überprüfen 777
  - enable\_if 748
  - Ersetzungsfehler 748
  - Fakultät 856
  - Fehlererkennung 729
  - Friends 738
  - Funktionen 88
  - Gültigkeitsbereiche 738
  - Hierarchien 827
  - Instanziierung 725, 801, 802
  - Iteration 856
  - Konstruktoren 734
  - Konvertierungen 826
  - Konzepte 727, 762
  - Lisp 841
  - Matrix 894
  - Member 733
  - Member, statische 732
  - Member-Funktionen 731
  - Metaprogramme 350
  - Namen aus Basisklassen 817
  - Namespaces 814
  - numeric\_limits 1254
  - Operationen als Argumente 786
  - Parameter 782
  - Parameter als Basisklassen 829
  - Parameterpakete 875
  - Polymorphie 821
  - primäre 795
  - Quellcodeorganisation 751
  - Rekursion 856
  - Schnittstellen 829
  - Schnittstellenspezialisierungen 794
  - Spezialisierungen 725, 802
  - Spezialisierungen, explizite 750
  - Spezialisierungsreihenfolge 797
  - Standardargumente 789
  - Standardbeziehungen 826
  - String 1118
  - Strings 722
  - Templates als Argumente 788
  - Typen, benutzerdefinierte 783
  - Typen, integrierte 783
  - Typalias 731
  - Typäquivalenz 728
  - Typargumente 782
  - Typprüfung 726
  - variadische 92, 126, 350, 873
  - Vererbung 749
  - virtual 735
  - Weiterleitung 878
  - Wertargumente 784
  - Wertparameter 784
  - wstring\_convert 1248
- Template-Funktionen
  - Funktions-Templates 724
  - Instanziierung 803
  - Spezialisierungen 798
- Template-Klassen
  - Klassen-Templates 724
- Template-Literaloperator 606
- Template-Metaprogrammierung 135, 757
  - Beispiel 866
- temporäre Objekte 285
- terminate() 383, 407, 941, 1311
- terminate\_handler 407
- Tests 49
- test\_and\_set() 1303
- Test-Harnisch 779
- text\_file\_busy 951
- Textkörper 318
- The Annotated C++ Reference Manual (ARM) 30
- this 504
  - Lambda-Ausdrücke 324
- \*this 504
- this\_thread 1315
- thousands\_sep() 1221, 1228
- thread 936

- <thread> 933
- Threads 125, 1287, 1307
  - Aushungern 1319
  - Bedingungsvariablen 1330
  - call\_once() 1329
  - cancel 1316
  - condition\_variable 1330
  - Dämonen 1313
  - Data Races 125
  - Destruktoren 1311
  - detach() 1313
  - Ereignisse 129
  - future 1342
  - Gültigkeitsbereich verlassen 1315
  - Hyperthreading 1309
  - Identität 1309
  - interrupt 1316
  - join() 1312
  - joinable() 1311
  - kill 1316
  - Konstruktion 1310
  - kritische Abschnitte 1320
  - once\_flag 1329
  - packaged\_task 1339
  - Semantik des exklusiven Besitzes 1320
  - shared\_future 1345
  - Speicherdauer 1317
  - sperrbare Objekte 1326
  - Synchronisierung 1296
  - terminate() 1311
  - this\_thread 1315
  - thread\_local 1316
  - verknüpfbare 1311
  - vorzeitig beenden 1316
  - wait\_for\_all() 1344
  - wait\_for\_any() 1344
  - Wrapper 1311
  - Zeitüberschreitung 1334
  - Zerstörung 1311
- thread launcher (Thread-Starter) 1346
- thread\_local 1316
- Thread-Starter 1346
- throw 62
  - Ausdrücke, bedingte 301
- throw\_with\_nested() 939
- tie() 1064, 1065, 1178
- tiefe Kopie 552
- time 1203, 1215
- time() 1364
- \_\_TIME\_\_ 371
- time\_get 1215, 1233
- timed\_mutex 1316, 1319, 1323
- timed\_out 951
- timeout 1343
- time\_point 1089, 1093
- time\_put 1215, 1232
- time\_put\_byname 1215
- time\_t 1364
- tm 1232, 1364
- to\_bytes() 1249
- to\_char\_type() 1117
- to\_int\_type() 1117
- to\_string() 1059, 1126
- to\_time\_t() 1095
- to\_ullong() 1059
- to\_ulong() 1059
- to\_wstring() 1126
- Token
  - Regel der längsten Übereinstimmung 281
- Tokenizing 264
- Token-Splitting 1155
- Token\_stream 264
- tolower() 1116, 1236, 1248
- too\_many\_files\_open 951
- too\_many\_files\_open\_in\_system 951
- too\_many\_links 951
- too\_many\_symbolic\_link\_levels 951
- totale Ordnung 963
- toupper() 1116, 1236, 1248
- tp=now() 1095
- traceable (verfolgbar) 1084
- Traits 328, 850
  - Allokatoren 1079
  - is\_error\_code\_enum 949
  - Iteratoren 1032
  - Typ- 1099
  - Zeichen- 1117
  - Zeiger 1080
  - Zeit 1096
- Transactional Memory (TM) 1294
- transform() 1010, 1156, 1218
- Transformationen 1106, 1110
  - einfachste sinnvolle 1237
- transform\_primary() 1156
- translate() 1156
- translate\_nocase() 1156
- true 151
- truname() 1221

- trunc 1176
- try
  - Funktionen 405
- try\_lock() 1320
- try\_lock\_for() 1323, 1327
- try\_lock\_until() 1323, 1327
- Tupel 877, 1061
  - konstante 871
  - pair 1061
  - tuple 1061
- tuple 138, 879, 1051, 1061, 1063
  - tie() 1065
- Tuple
  - Ausgabefunktionen 868
  - Elementzugriff 869
- <tuple> 930, 1063
- tuple\_cat() 1064
- tuple\_element 1063
- tuple\_elements 1064
- tuple\_size 1062, 1064
- Turing-vollständig 841
- Typen 46, 150, 152, 487
  - Abstand von Iteratoren 1034
  - abstrakte 73
  - Alias 508
  - arithmetische 151
  - atomare 1295, 1298
  - atomic\_flag 1303
  - auswählen 853, 855
  - auto 179
  - benutzerdefinierte 53, 151
  - bool 152
  - boolesche 151
  - char 153
  - char16\_t 153
  - char32\_t 153
  - Closure 325
  - complex 69
  - const wchar\_t[] 195
  - event 1179
  - event\_callback 1179
  - function 1046
  - fundamentale 150
  - Ganzzahl- 158
  - Ganzzahl-literale 159
  - Geldbeträge 1226
  - gemeinsame 1107
  - generierte 824
  - geordnete 770
  - Gleitkomma- 160
  - Größe 46
  - Größen 162
  - hash 987
  - herleiten 179
  - implizite Konvertierung 292
  - integrale 151
  - integrierte 53, 151
  - Iteratoren 115
  - komplexe Zahlen 140, 580
  - konkrete 68, 73, 509, 517
  - Lambda-Ausdrücke 325
  - literale 290
  - Mehrbytezeichen 195
  - Member 732
  - Member- 508
  - Member-Typen von Containern 966
  - numerische Grenzen 144
  - parametrisierte 87
  - polymorphe 74, 636
  - Prädikate 848
  - Referenz auf Array 348
  - reguläre 765
  - Rückgabe- 337
  - signed char 153
  - sizeof 46
  - Standard- 1382
  - time\_point 1093
  - trivial kopierbare 229
  - tuple 879
  - unsigned char 153
  - Verbund- 1100
  - void 162
  - wchar\_t 153
  - Werttypen 517
  - Zeichen 153
  - Zeiger 187
  - Zeitgeber 1095
  - zugeordnete 731
  - zugrunde liegende 238
- Typalias 184
  - Templates 731
  - value\_type 136
- Typäquivalenz 728
- Typargumente
  - Templates 782
- Type Erasure 794
- typeid 71
- typeid() 705, 936

- type\_index 1112
- <typeid> 930, 1112
- type\_info 71, 1112
- <typeid> 71, 932, 934
- Typeigenschaften
  - alignment\_of 1103
  - extent 1103
  - has\_virtual\_destructor 1103
  - is\_abstract 1101
  - is\_assignable 1101
  - is\_const 1100
  - is\_constructible 1101
  - is\_copy\_assignable 1101
  - is\_copy\_constructible 1101
  - is\_default\_constructible 1101
  - is\_destructible 1101
  - is\_empty 1100
  - is\_literal\_type 1100
  - is\_move\_assignable 1101
  - is\_move\_constructible 1101
  - is\_nothrow\_assignable 1102
  - is\_nothrow\_constructible 1102
  - is\_nothrow\_copy\_assignable 1102
  - is\_nothrow\_copy\_constructible 1102
  - is\_nothrow\_default\_constructible 1102
  - is\_nothrow\_destructible 1103
  - is\_nothrow\_move\_assignable 1103
  - is\_nothrow\_move\_constructible 1102
  - is\_pod 1100
  - is\_polymorphic 1100
  - is\_signed 1101
  - is\_standard\_layout 1100
  - is\_trivial 1100
  - is\_trivially\_assignable 1102
  - is\_trivially\_constructible 1102
  - is\_trivially\_copyable 1100
  - is\_trivially\_copy\_assignable 1102
  - is\_trivially\_copy\_constructible 1102
  - is\_trivially\_default\_constructible 1102
  - is\_trivially\_destructible 1102
  - is\_trivially\_move\_assignable 1102
  - is\_trivially\_move\_constructible 1102
  - is\_unsigned 1101
  - is\_volatile 1100
  - rank 1103
- typename
  - Template-Parameter 782
- <type\_traits> 853, 930, 1099
  - enable\_if 859
- Typfelder 631
- Typfunktionen 135, 843
  - Argumente 844
  - Array\_type 844
  - Common\_type 902
  - Conditional 852
  - decltype() 852
  - declval() 1107, 1109
  - is\_polymorphic<T> 843
  - Prädikate 1099
  - Select 844, 852
  - Typen auswählen 855
  - <type\_traits> 1099
  - Value\_type 1009
- Typgeneratoren 824, 1104
- Typidentifizierung 71
- Typinformationen
  - erweiterte 713
- Typkonvertierung
  - implizite 292
- Typprädikate 137
  - Eigenschaften 1100
    - is\_arithmetic 1100
    - is\_array 1099
    - is\_class 1099
    - is\_compound 1100
    - is\_enum 1099
    - is\_function 1099
    - is\_fundamental 1100
    - is\_integral 1099
    - is\_lvalue\_reference 1099
    - is\_member\_function\_pointer 1099
    - is\_member\_object\_pointer 1099
    - is\_member\_pointer 1100
    - is\_object 1100
    - is\_pod 848
    - is\_pointer 1099
    - is\_reference 1100
    - is\_rvalue\_reference 1099
    - is\_scalar 1100
    - is\_union 1099
    - is\_void 1099
    - primäre 1099
    - zusammengesetzte 1100
- Typprüfung 726
  - printf() 1359
- Typrelationen 1103
  - is\_base\_of 1103

- is\_convertible 1103
- is\_same 1103
- Typsicherheit 841
- Typumwandlungen 588
  - auto 48
  - benannte 329
  - C- 331
  - casting 17, 330
  - explizite 326
  - funktionale Notation 331
  - move() 1109
  - R-Werte 1109

## U

### Übereinstimmungen

- Ergebnisse 1143
- faule 1140
- gierige 1140
- Überladen 356
  - Funktions-Templates 745
  - Gültigkeitsbereiche 359
  - Namespaces 446
  - new 310
  - Operatoren 516
  - Rückgabetypen 358
- Überladungen
  - Ableitungen 749
  - automatisch auflösen 356
  - manuelle Auflösung 360
  - nicht hergeleitete Parameter 749
  - Spezialisierungen 798
  - using 430

### Überlauf 995

- Iteratoren 1039
- Streams 1190
- überschreiben 75, 635

### Übersetzung

- bedingte 370
- Header-Dateien 460
- Vorübersetzung 460
- Übersetzungseinheiten 455
- Übersetzungszeit
  - Arithmetik 1097
  - Laufzeit 853

### UDLs (User-Defined Literals)

- *siehe* Literale, benutzerdefinierte
- uflow() 1191

### Uhren *siehe auch* Zeitgeber

- externe 1096
- Uhrzeit 134
- Umordnung 1291
- uncaught\_exception() 941
- Und 299
- undefiniert 148
- #undef 370
- underflow() 1191
- underlying\_type 1106
- unescaped 194
- unformatierter Eingabe 1170
- ungeordnete assoziative Container 981
- ungetc() 1361
- Ungleichheit
  - complex 1260
- Unicode
  - Codepunkte 196
  - Literale 195
- uniform\_int\_distribution 141
- uninitialized\_copy() 413, 1016
- uninitialized\_copy\_n() 1016
- uninitialized\_fill() 413, 1016
- uninitialized\_fill\_n() 413, 1016
- Unions 231
  - anonyme 234, 236, 611
  - diskriminierte 235
  - Klassen 233
  - tagged 235
- unique() 1012, 1072
- unique\_copy() 1012
- unique\_lock 1319, 1324
- unique\_ptr 122, 1066
  - -> (Operator) 598
- unitbuf 1180, 1183
- universelle Initialisierung 532
- universelle Zeichennamen 157, 196
- unlock() 1320, 1323, 1327
- unordered\_map 110, 987
- <unordered\_map> 930
- <unordered\_set> 930
- unsetf() 1181
- unshift() 1240, 1241
- unsigned char 153
- Unterlauf 995
- Untermatrizen
  - Matrix\_slice 907
- Unterschiede
  - stillschweigende 1378

Unveränderlichkeit 48  
 Upcast 694  
 uppercase 1180, 1183  
 use\_count() 1072  
 use\_facet() 1210  
 User-Defined Literals, UDLs 604  
 uses\_allocator 1340  
 using 61, 429, 642  
 – Basisklassen 642  
 – Direktiven 430  
 – Überladungen 430  
 Utilities 1109  
 <utility> 930, 1061, 1109  
 – relationale Operatoren 1111

## V

valarray 1051, 1260  
 – Indizierung 1263  
 – Konstruktoren 1261  
 – Operationen 1264  
 – Optimierung 1266  
 – Row-major Order 1267  
 – Verschieben 1266  
 – Zuweisungen 1262  
 <valarray> 932, 1260  
 – Vektoren 143  
 valid() 1340, 1343, 1346  
 value 843  
 value() 1156  
 value\_too\_large 951  
 value\_type 136, 966  
 Value\_type 1009  
 Variablen 46  
 – automatische 338  
 – deklarieren in Bedingungen 252  
 – lokale 338, 343  
 – nichtlokale initialisieren 479  
 – thread\_local 1316  
 variadische Templates 92  
 Vec 106  
 vector 104, 936, 1051  
 – Argumente 974  
 – at() 106  
 – Bereichsüberprüfung 106  
 – Implementierung 410  
 – Iteratoren 115  
 – Member 974  
 – push\_back() 419, 421

– reserve() 420  
 – resize() 419, 421  
 – string 978  
 – Typen 974  
 <vector> 930  
 vector\_base 415  
 vector<bool> 1060  
 Vektoren  
 – Arithmetik 143  
 – Slots 959, 975  
 Veränderlichkeit 500  
 – Indirektion 502  
 Verband 686  
 Verbundanweisungen 246  
 Verbundtypen 1100  
 verdeckte Namen 173  
 Vererbung 74, 627  
 – Implementierungs- 79, 649, 664  
 – Mehrfach- 698  
 – Schnittstellen- 79, 649, 668  
 verfolgbare Zeiger 1084  
 Vergleiche  
 – lexikografische 1026  
 – Operatoren, relationale 962  
 – strenge schwache Ordnung 961  
 – Wert- 1026  
 Vergleichen  
 – Strings 1208, 1217  
 – Teilstrings 1132  
 Vergleichen und Vertauschen 1299  
 Vergleichsfunktionen 365  
 Vergleichsobjekte 982  
 verkettete Listen  
 – Verknüpfungstyp 735  
 Verkettung 98  
 Verklemmungen 1322  
 verschachtelte Klassen 508  
 Verschiebeiteratoren 1040  
 Verschiebekonstruktoren 84, 522  
 – vector\_base 417  
 Verschieben 521, 548, 555  
 – Bits 1058  
 – Container 83  
 – Linksschieben 1058  
 – Objekte 81  
 – Rechtsschieben 1058  
 – valarray 1266  
 Verschiebezuweisungen 84, 522  
 – vector\_base 417

- Verschränkung 552
- Verteilungen 141
  - exponential\_distribution 141
  - Glockenkurve 141
  - Normal- 1283
  - normal\_distribution 141
  - Poisson- 1283
  - Stichproben- 1284
  - uniform\_int\_distribution 141
  - Zufallszahlen 1276, 1281
- Vertical Tab 156
- Vertikaltabulator 156, 1115
- virtual 74, 335, 634
  - Destruktoren 528
  - Templates 735
- virtuelle Funktionen 74, 76, 634
  - dynamische Bindung 705
- virtuelle Funktionstabellen 76, 636
- virtuelle Konstruktoren 646, 674
- Visitor 708
- Visitor *siehe* Besucher
- void 162, 337
- void\* 188
- volatile 335, 1304
- vollständige Ausdrücke 183, 285
- Vorbedingungen 361
- Vorlagen *siehe* Templates
- Vorwärts-Iteratoren 135
- vtbl 76, 636

## W

- Wächter 392
  - Include- 478
  - lock\_guard 1325
- Wagenrücklauf 156, 1115
- wait() 1343, 1346
- wait\_for() 1316, 1343, 1346
- wait\_for\_all() 1344
- wait\_for\_any() 1344
- wait\_until() 1316, 1343, 1346
- Warteschlangen 129
  - Prioritäts- 996
  - priority\_queue 996
  - queue 996
  - Sync\_queue 1333
- wbuffer\_convert 1250
- wcerr 1161
- wchar\_t 153

- wcin 1161
- wclog 1161
- wcout 1161
- weak\_ptr 1073
- weibull\_distribution() 1283
- weiterleitende Konstruktoren 544
- Weiterleitung
  - Argumente 1110
  - Ausnahmen 938
  - forward() 1110
- Wertargumente 784
- Wertkonzepte 775
- wertorientierte Programmierung 517
- Wertparameter 784
- Werttypen 136, 517
- Wertvergleiche 1026
- Wettlaufsituationen 129
- WG21 30
- what() 937
- while 50, 256
  - Endlosschleifen 266
- Whitespaces 270, 1115
  - Ellipse 876
  - unterdrücken 1169
- widen() 1178, 1236
  - Transformation, einfachste sinnvolle 1237
- Widgets 693
- width() 1181
- Wiederholungen 1136
- Wörterbuch 109
- Wrapper
  - Referenzen 1045, 1311
- write() 1172
- wrong\_protocol\_type 951
- wstring\_convert 1248

## X

- xalloc() 1179
- xor 282
- XOR 1058
  - Hashwerte 991
- xor\_eq 282
- xsgethn() 1192
- xsputhn() 1192



## Y

yield() 1315

## Z

## Zahlen

- Dezimaltrennzeichen 1221
- Hexadezimal- 156
- Interpunktion 1221
- komplexe 140, 580, 1259
- Konvertierungen (C-Strings) 1362
- numerische Grenzen 144
- Oktal- 156
- Zufalls- 141, 1275

## Zeichen

- Klassifizierung 1115
- universelle Namen 157
- vorzeichenbehaftete 155
- vorzeichenlose 155

## Zeichenklassen 1137

- Abkürzungen 1138
- mask 1235

## Zeichenklassifizierung 1235

- char\_traits 1117
- isalnum() 1116
- isalpha() 1115
- isblank() 1115
- iscntrl() 1116
- isdigit() 1115
- isgraph() 1116
- islower() 1116
- isprint() 1116
- ispunct() 1116
- isspace() 1115
- isupper() 1116
- isxdigit() 1115
- Locale-abhängige 1247

## Zeichenkonstanten 101

## Zeichenliterale 156

## Zeichensätze 153

- ASCII 149
- codecvt 1239
- EBCDIC 154
- grundlegender Quellzeichensatz 149, 1238
- ISO 646-1983 149
- japanische 1239
- Shift 1239

## Zeichen-Traits 1117

## Zeiger 187

- Addition 199
  - Arrays 196
  - atomare 1302
  - auf Funktion 363
  - Besitz 205, 1065
  - const 203
  - constexpr 342
  - Daten-Member 659
  - Funktions-Member 657
  - gemeinsame 1069
  - getarnte 1083
  - intelligente 86, 122, 598, 1065
  - Konvertierungen 295
  - Member 657
  - NULL 53, 190
  - nullptr 52, 189
  - Referenzen 215
  - Ressourcenverwaltung 1065
  - Schleifen 51
  - shared\_ptr 122, 1069
  - sicher abgeleitete 1083
  - Subtraktion 199
  - Traits 1080
  - unique\_ptr 122, 1066
  - verfolgbare 1084
  - void\* 188
  - weak\_ptr 1073
- Zeilenvorschub 156
- Zeit 134, 1089
- duration\_cast 1093
- Zeit dauern 1089
- konstruieren 1093
- Zeitgeber 1089, *siehe auch* Uhren
- Genauigkeit 1093
  - Schnittstellen 1095
  - Typen 1095
  - wait\_for() 1316
  - wait\_until() 1316
- Zeitpunkte 1089, 1232
- Epoche 1093
- Zeitüberschreitung 1334
- zero() 1097
- duration 1092
- Zerstörung 1311
- Zufallsgeräte 1275, 1280
- Zufallszahlen 141, 1275
- bind() 1276
  - C 1285

- Generatoren 1276, 1278
- Geräte 1280
- gleichförmige 1276
- gleichverteilte 1014
- Moduladapter 1279
- Module 1278
- <random> 141
- seed 1278
- Startwert 1278
- Stichproben 1277
- Verteilungen 1281
- Zufallszahlenmodul 1276
- Zufallszahlenmoduladapter 1276
- Zufallszahlenverteilung 1276
- zugeordnete Typen 731
- Zugriff
  - Member 505
- Zugriffskontrolle 491, 649
  - private 491
  - public 491
  - using 656
- Zugriffsspezifizierer
  - Basisklassen 654
  - private 493
  - public 493
- zugrunde liegende Typen 238
- Zuordnungseinheiten 231
- Zusammenführen
  - Sequenzen 1022
- zusammengesetzte Typprädikate 1100
- Zusicherungen 394, 941
  - siehe auch* Assertionen
- Zustandsmaschinen 1136
- Zuweisungen 417
  - basic\_string 1128
  - Container 967
  - Kopier- 82, 522
  - valarray 1262
  - Verschiebe- 84, 522
- Zwischenraum 1115