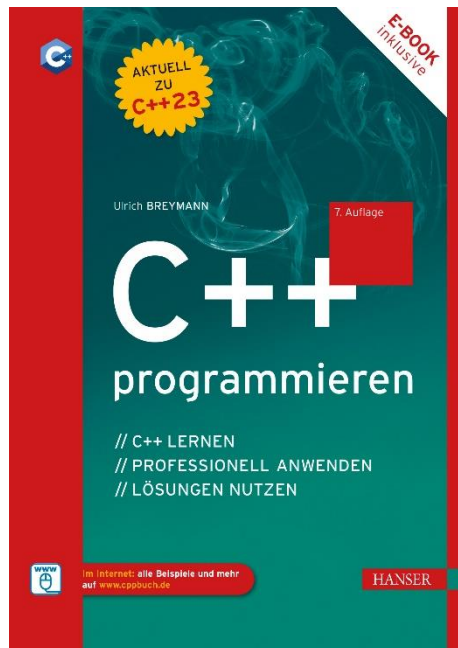


# HANSER



## Leseprobe

zu

## C++ programmieren

von Ulrich Breymann

Print-ISBN: 978-3-446-47689-9

E-Book-ISBN: 978-3-446-47846-6

E-Pub-ISBN: 978-3-446-47964-7

Weitere Informationen und Bestellungen unter

<https://www.hanser-kundencenter.de/fachbuch/artikel/9783446476899>

sowie im Buchhandel

© Carl Hanser Verlag, München

# Inhalt

<b>Vorwort .....</b>	<b>23</b>
<b>Teil I: Einführung in C++ .....</b>	<b>25</b>
<b>1 Es geht los! .....</b>	<b>27</b>
1.1 Historisches .....	27
1.2 Arten der Programmierung .....	28
1.3 Werkzeuge zum Programmieren .....	29
1.4 Das erste Programm .....	30
1.5 Integrierte Entwicklungsumgebung .....	36
1.6 Einfache Datentypen und Operatoren .....	39
1.6.1 Ausdruck .....	39
1.6.2 Regeln für Namen .....	39
1.6.3 Ganze Zahlen .....	40
1.6.4 Reelle Zahlen .....	48
1.6.5 Konstanten .....	52
1.6.6 Zeichen .....	54
1.6.7 Logischer Datentyp bool .....	57
1.6.8 Regeln zum Bilden von Ausdrücken .....	58
1.6.9 Standard-Typumwandlungen .....	59
1.7 Gültigkeitsbereich und Sichtbarkeit .....	61
1.7.1 Namespace std.....	63
1.8 Kontrollstrukturen .....	64

1.8.1	Anweisungen .....	64
1.8.2	Sequenz (Reihung) .....	65
1.8.3	Auswahl (Selektion, Verzweigung) .....	66
1.8.4	Fallunterscheidungen mit switch .....	71
1.8.5	Wiederholungen.....	74
1.8.6	Kontrolle mit break und continue .....	82
1.8.7	goto .....	84
1.9	Selbst definierte und zusammengesetzte Datentypen .....	85
1.9.1	Aufzählungstypen .....	85
1.9.2	Strukturen.....	88
1.9.3	Der C++-Standardtyp vector .....	89
1.9.4	Der C++-Standardtyp array.....	95
1.9.5	Zeichenketten: der C++-Standardtyp string .....	95
1.9.6	Container und Schleifen .....	98
1.9.7	Typermittlung mit auto .....	100
1.9.8	Deklaration einer strukturierten Bindung mit auto.....	102
1.9.9	Bitfeld und Union .....	103
1.10	Einfache Ein- und Ausgabe .....	104
1.10.1	Standardein- und -ausgabe.....	104
1.10.2	Ein- und Ausgabe mit Dateien .....	108
1.11	Guter Programmierstil.....	113
<b>2</b>	<b>Programmstrukturierung .....</b>	<b>115</b>
2.1	Funktionen.....	116
2.1.1	Aufbau und Prototypen .....	116
2.1.2	nodiscard .....	118
2.1.3	Gültigkeitsbereiche und Sichtbarkeit in Funktionen .....	119
2.1.4	Lokale static-Variable: Funktion mit Gedächtnis .....	120
2.2	Schnittstellen zum Datentransfer .....	121
2.2.1	Übergabe per Wert .....	122
2.2.2	Übergabe per Referenz .....	124
2.2.3	Gefahren bei der Rückgabe von Referenzen.....	126
2.2.4	Vorgegebene Parameterwerte und unterschiedliche Parameterzahl .....	127
2.2.5	Überladen von Funktionen .....	128
2.2.6	Funktion main() .....	130
2.2.7	Beispiel Taschenrechnersimulation .....	130
2.2.8	Spezifikation von Funktionen.....	136

2.2.9	Reihenfolge der Auswertung von Argumenten .....	136
2.3	Präprozessordirektiven .....	136
2.3.1	#include .....	137
2.3.2	#define, #if, #ifdef, #ifndef, #elif, #else, #endif, #elifdef, #elifndef.....	137
2.3.3	Vermeiden mehrfacher Inkludierung .....	138
2.3.4	__has_include .....	140
2.3.5	Textersetzung mit #define .....	140
2.3.6	Umwandlung von Parametern in Zeichenketten.....	142
2.3.7	Verifizieren logischer Annahmen zur Laufzeit .....	143
2.3.8	Verifizieren logischer Annahmen zur Compilationszeit .....	143
2.3.9	Fehler- und Warnmeldungen .....	144
2.3.10	Fehler ohne Programmabbruch lokalisieren.....	144
2.4	Modulare Programmgestaltung .....	145
2.4.1	Projekt: Mehrere cpp-Dateien bilden ein Programm .....	145
2.4.2	Projekt in der IDE anlegen.....	147
2.4.3	Übersetzungseinheit, Deklaration, Definition .....	148
2.4.4	Dateiübergreifende Gültigkeit und Sichtbarkeit.....	150
2.5	Namensräume .....	151
2.5.1	Gültigkeitsbereich auf Datei beschränken .....	154
2.6	inline-Funktionen und -Variablen.....	155
2.7	constexpr-Funktionen.....	156
2.7.1	Berechnung zur Compilationszeit mit consteval.....	158
2.8	Rückgabotyp auto.....	159
2.9	Funktions-Templates .....	161
2.9.1	Spezialisierung von Templates .....	163
2.9.2	Einbinden von Templates .....	164
2.10	C++-Header .....	167
2.11	Module .....	169
<b>3</b>	<b>Objektorientierung 1 .....</b>	<b>173</b>
3.1	Datentyp und Objekt .....	175
3.2	Abstrakter Datentyp .....	175
3.3	Klassen .....	177
3.3.1	const-Objekte und Methoden.....	180
3.3.2	inline-Elementfunktionen.....	181
3.4	Initialisierung und Konstruktoren .....	182
3.4.1	Standardkonstruktor.....	182

3.4.2	Direkte Initialisierung der Attribute .....	184
3.4.3	Allgemeine Konstruktoren .....	184
3.4.4	Kopierkonstruktor .....	187
3.4.5	Typumwandlungskonstruktor .....	190
3.4.6	Konstruktor und mehr vorgeben oder verbieten .....	191
3.4.7	Einheitliche Initialisierung und Sequenzkonstruktor .....	192
3.4.8	Delegierender Konstruktor .....	194
3.4.9	constexpr-Konstruktor und -Methoden .....	195
3.5	Beispiel Rationale Zahlen .....	199
3.5.1	Aufgabenstellung .....	199
3.5.2	Entwurf .....	200
3.5.3	Implementation .....	203
3.6	Destruktoren .....	208
3.7	Wie kommt man zu Klassen und Objekten? Ein Beispiel .....	210
3.8	Gegenseitige Abhängigkeit von Klassen .....	215
<b>4</b>	<b>Zeiger .....</b>	<b>217</b>
4.1	Zeiger und Adressen .....	218
4.2	C-Arrays .....	222
4.2.1	C-Array, std::size() und sizeof .....	223
4.2.2	Initialisierung von C-Arrays .....	224
4.2.3	Zeigerarithmetik .....	224
4.2.4	Indexoperator bei C-Arrays .....	225
4.2.5	C-Array durchlaufen .....	226
4.3	C-Zeichenketten .....	227
4.3.1	Schleifen und C-Strings .....	230
4.4	Dynamische Datenobjekte .....	233
4.4.1	Freigeben dynamischer Objekte .....	236
4.5	Zeiger und Funktionen .....	239
4.5.1	Parameterübergabe mit Zeigern .....	239
4.5.2	C-Array als Funktionsparameter .....	240
4.5.3	const und Zeiger-Parameter .....	242
4.5.4	Parameter des main-Programms .....	243
4.5.5	Gefahren bei der Rückgabe von Zeigern .....	243
4.6	this-Zeiger .....	244
4.7	Mehrdimensionale C-Arrays .....	246
4.7.1	Statische mehrdimensionale C-Arrays .....	246

4.7.2	Mehrdimensionales C-Array als Funktionsparameter .....	248
4.8	Dynamisches 2D-Array .....	251
4.9	Binäre Ein-/Ausgabe .....	257
4.10	Zeiger auf Funktionen.....	260
4.11	Typumwandlungen für Zeiger.....	264
4.12	Zeiger auf Elementfunktionen und -daten .....	265
4.13	Komplexe Deklarationen lesen .....	267
4.13.1	Lesbarkeit mit typedef und using verbessern .....	267
4.14	Alternative zu rohen Zeigern, new und delete .....	269
<b>5</b>	<b>Objektorientierung 2 .....</b>	<b>273</b>
5.1	Eine String-Klasse .....	273
5.1.1	friend-Funktionen .....	279
5.2	String-Ansicht (View) .....	280
5.3	Typbestimmung mit decltype und declval.....	283
5.4	Klassenspezifische Daten und Funktionen .....	286
5.4.1	Klassenspezifische Konstante.....	290
5.5	Klassen-Templates .....	293
5.5.1	Ein Stack-Template .....	293
5.5.2	Stack mit statisch festgelegter Größe.....	295
5.5.3	Stack auf Basis verschiedener Container .....	297
5.6	Code Bloat bei der Instanziierung von Templates vermeiden.....	298
5.6.1	extern-Template .....	299
<b>6</b>	<b>Vererbung .....</b>	<b>301</b>
6.1	Vererbung und Initialisierung.....	306
6.2	Zugriffsschutz .....	307
6.3	Typbeziehung zwischen Ober- und Unterklasse .....	310
6.4	Oberklassen-Schnittstelle verwenden .....	311
6.4.1	Konstruktor erben.....	312
6.5	Überschreiben von Funktionen in abgeleiteten Klassen .....	314
6.5.1	Virtuelle Funktionen.....	316
6.5.2	Abstrakte Klassen .....	319
6.5.3	Virtueller Destruktor.....	324
6.5.4	Vererbung verbieten .....	327
6.5.5	Private virtuelle Funktionen.....	328
6.6	Probleme der Modellierung mit Vererbung.....	330
6.7	Mehrfachvererbung.....	333

6.8	Typumwandlung bei Vererbung .....	340
6.9	Typinformationen zur Laufzeit.....	342
6.10	Private-/Protected-Vererbung .....	343
<b>7</b>	<b>Fehlerbehandlung.....</b>	<b>347</b>
7.1	Ausnahmebehandlung .....	349
7.1.1	Exception-Spezifikation in Deklarationen.....	352
7.1.2	Exception-Hierarchie .....	353
7.1.3	Besondere Fehlerbehandlungsfunktionen.....	355
7.1.4	Arithmetische Fehler/Division durch 0.....	356
7.2	Speicherbeschaffung mit new .....	358
7.3	Exception-Sicherheit .....	359
7.4	Fehlerbehandlung mit optional und expected .....	360
7.4.1	Fehlerbehandlung mit optional .....	361
7.4.2	Fehlerbehandlung mit expected.....	363
7.4.3	Monadische Operationen.....	364
<b>8</b>	<b>Überladen von Operatoren .....</b>	<b>367</b>
8.1	Rationale Zahlen – noch einmal .....	369
8.1.1	Arithmetische Operatoren.....	369
8.1.2	Ausgabeoperator << .....	371
8.1.3	Gleichheitsoperator .....	372
8.2	Eine Klasse für Vektoren .....	374
8.2.1	Indexoperator [ ] .....	377
8.2.2	Zuweisungsoperator = .....	379
8.2.3	Mathematische Vektoren .....	382
8.2.4	Multiplikationsoperator .....	383
8.3	Inkrement-Operator ++ .....	385
8.4	Typumwandlungsoperator .....	389
8.5	Smart Pointer: Operatoren -> und * .....	390
8.5.1	Smart Pointer und die C++-Standardbibliothek .....	395
8.6	Objekt als Funktion .....	396
8.7	Spaceship-Operator <=> .....	398
8.7.1	Ordnungen in C++ .....	399
8.7.2	Automatische Erzeugung der Vergleichsoperatoren .....	401
8.7.3	Klassenspezifische Sortierung .....	402
8.7.4	Freie Funktionen statt Elementfunktionen .....	403
8.8	new und delete überladen .....	404

8.8.1	Unterscheidung zwischen Heap- und Stack-Objekten .....	408
8.8.2	Empfehlungen im Umgang mit new und delete .....	410
8.9	Operatoren für Literale .....	410
8.9.1	Stringliterale .....	411
8.9.2	Benutzerdefinierte Literale .....	412
8.10	Indexoperator für Matrizen .....	414
8.10.1	Zweidimensionale Matrix als Vektor von Vektoren .....	415
8.10.2	Zweidimensionale Matrix mit zusammenhängendem Speicher .....	416
8.11	Zuweisung, Kopie und Vergleich bei Vererbung .....	419
8.11.1	Polymorpher Vergleich .....	420
8.11.2	Kopie mit clone()-Methode erzeugen .....	421
<b>9</b>	<b>Dateien und Ströme .....</b>	<b>423</b>
9.1	Eingabe .....	425
9.2	Ausgabe .....	427
9.3	Formatierung mit std::format .....	429
9.3.1	Syntax für Platzhalter .....	429
9.3.2	Formatierung eigener Datentypen .....	433
9.4	Formatierung mit Flags .....	433
9.5	Formatierung mit Manipulatoren .....	437
9.6	Fehlerbehandlung .....	443
9.7	Typumwandlung von Dateiobjekten nach bool .....	445
9.8	Arbeit mit Dateien .....	446
9.8.1	Positionierung in Dateien .....	446
9.8.2	Lesen und Schreiben in derselben Datei .....	447
9.9	Umleitung auf Strings .....	448
9.10	Formatierte Daten lesen .....	449
9.10.1	Eingabe benutzerdefinierter Typen .....	449
9.11	Blockweise lesen und schreiben .....	451
9.11.1	vector-Objekt binär lesen und schreiben .....	451
9.11.2	array-Objekt binär lesen und schreiben .....	453
9.11.3	Matrix binär lesen und schreiben .....	454
<b>10</b>	<b>Die Standard Template Library (STL) .....</b>	<b>457</b>
10.1	Container, Iteratoren, Algorithmen .....	458
10.2	Iteratoren im Detail .....	463
10.3	Beispiel verkettete Liste .....	465
10.4	Ranges und Views .....	469



**Teil II: Fortgeschrittene Themen .....473****11 Performance, Wert- und Referenzsemantik .....475**

11.1	Performanceproblem Wertsemantik .....	477
11.1.1	Auslassen der Kopie .....	477
11.1.2	Temporäre Objekte bei der Zuweisung .....	478
11.2	Referenzsemantik für R-Werte .....	479
11.2.1	Kategorien von Ausdrücken .....	479
11.2.2	Referenzen auf R- und L-Werte .....	480
11.2.3	Auswertung von Referenzen auf R-Werte .....	481
11.2.4	Referenz-Qualifizierer .....	482
11.3	Optimierung durch Referenzsemantik für R-Werte.....	484
11.3.1	Bewegungskonstruktor .....	487
11.3.2	Bewegender Zuweisungsoperator .....	488
11.4	Die move()-Funktion .....	489
11.4.1	move() und Initialisierung der Attribute .....	490
11.5	Referenzen auf R-Werte und Template-Parameter .....	491
11.5.1	Auswertung von Template-Parametern – ein Überblick.....	493
11.6	Ein effizienter Plusoperator .....	493
11.6.1	Eliminieren auch des Bewegungskonstruktors .....	494
11.6.2	Kopien temporärer Objekte eliminieren .....	495
11.7	Rule of three/five/zero .....	496

**12 Lambda-Funktionen ..... 501**

12.1	Eigenschaften.....	502
12.1.1	Äquivalenz zum Funktionszeiger .....	503
12.1.2	Lambda-Funktion und Klasse .....	504
12.2	Generische Lambda-Funktionen .....	505
12.3	Parametererfassung mit [] .....	507

**13 Metaprogrammierung mit Templates .....509**

13.1	Grundlagen .....	510
13.2	Variadic Templates: Templates mit variabler Parameterzahl .....	512
13.2.1	Ablauf der Auswertung durch den Compiler.....	513
13.2.2	Anzahl der Parameter .....	514
13.2.3	Parameterexpansion .....	515
13.3	Fold-Expressions.....	516
13.3.1	Weitere Varianten .....	518

13.3.2	Fold-Expression mit Kommaoperator .....	519
13.4	Klassen-Template mit variabler Stelligkeit .....	520
13.5	Type Traits .....	521
13.5.1	Wie funktionieren Type Traits? – ein Beispiel .....	522
13.5.2	Abfrage von Eigenschaften .....	525
13.5.3	Abfrage numerischer Eigenschaften .....	527
13.5.4	Typumwandlungen .....	527
13.5.5	Auswahl weiterer Traits .....	528
13.6	Concepts .....	530
<b>14</b>	<b>Reguläre Ausdrücke .....</b>	<b>535</b>
14.1	Elemente regulärer Ausdrücke .....	536
14.1.1	Greedy oder lazy? .....	538
14.2	Interaktive Auswertung .....	540
14.3	Auszug der regex-Schnittstelle .....	543
14.4	Verarbeitung von \n .....	544
14.5	Anwendungen .....	546
<b>15</b>	<b>Threads und Coroutinen .....</b>	<b>547</b>
15.1	Zeit und Dauer .....	548
15.2	Threads .....	549
15.2.1	Automatisch join() .....	553
15.3	Die Klasse jthread .....	554
15.3.1	Übergabe eines Funktors .....	556
15.3.2	Thread-Group .....	558
15.4	Synchronisation kritischer Abschnitte .....	559
15.4.1	Data Race erkennen .....	562
15.5	Thread-Steuerung: Pausieren, Fortsetzen, Beenden .....	562
15.6	Warten auf Ereignisse .....	566
15.7	Atomare Veränderung von Variablen .....	572
15.8	Asynchrone verteilte Bearbeitung einer Aufgabe .....	575
15.9	Thread-Sicherheit .....	578
15.10	Coroutinen .....	579
<b>16</b>	<b>Grafische Benutzungsschnittstellen .....</b>	<b>585</b>
16.1	Ereignisgesteuerte Programmierung .....	586
16.2	GUI-Programmierung mit Qt .....	587
16.2.1	Meta-Objektsystem .....	587

16.2.2	Der Programmablauf .....	588
16.2.3	Ereignis abfragen .....	589
16.3	Signale, Slots und Widgets .....	590
16.4	Dialog .....	599
16.5	Qt oder Standard-C++? .....	602
16.5.1	Threads .....	602
16.5.2	Verzeichnisbaum durchwandern .....	604

## **17 Internet-Anbindung .....607**

17.1	Protokolle .....	608
17.2	Adressen.....	608
17.3	Socket .....	611
17.3.1	Bidirektionale Kommunikation.....	614
17.3.2	UDP-Sockets .....	616
17.3.3	Atomuhr mit UDP abfragen .....	618
17.4	HTTP .....	620
17.4.1	Verbindung mit GET .....	621
17.4.2	Verbindung mit POST .....	626
17.5	Mini-Webserver .....	627
17.6	OpenAI-Schnittstellen zu ChatGPT und DALL·E 2 .....	635
17.6.1	ChatGPT .....	636
17.6.2	DALL·E 2 .....	638

## **18 Datenbankanbindung .....639**

18.1	C++-Interface.....	640
18.2	Anwendungsbeispiel.....	643

## **Teil III: Ausgewählte Methoden und Werkzeuge der Softwareentwicklung .....649**

## **19 Effiziente Programmerzeugung mit make ..... 651**

19.1	Wirkungsweise .....	652
19.2	Variablen und Muster .....	654
19.3	Universelles Makefile für einfache Projekte .....	656
19.4	Automatische Ermittlung von Abhängigkeiten .....	657
19.4.1	Makefiles für verschiedene Betriebssysteme und Compiler.....	659
19.4.2	Getrennte Verzeichnisse: src, obj, bin .....	660
19.5	Makefile für Verzeichnisbäume .....	661

19.5.1	Nur ein Makefile auf Projektebene .....	663
19.5.2	Rekursive Make-Aufrufe .....	664
19.6	Erzeugen von Bibliotheken .....	666
19.6.1	Statische Bibliotheksmodule .....	666
19.6.2	Dynamische Bibliotheksmodule .....	668
19.7	Weitere Build-Tools .....	671
<b>20</b>	<b>Unit-Test .....</b>	<b>673</b>
20.1	Werkzeuge .....	674
20.2	Boost Unit Test Framework .....	675
20.2.1	Fixture .....	677
20.2.2	Testprotokoll und Log-Level .....	677
20.2.3	Prüf-Makros .....	678
20.2.4	Kommandozeilen-Optionen .....	683
20.3	Test Driven Development .....	684

## **Teil IV: Das C++-Rezeptbuch: Tipps und Lösungen für typische Aufgaben .....685**

<b>21</b>	<b>Sichere Programmentwicklung .....</b>	<b>687</b>
21.1	Regeln zum Design von Methoden .....	688
21.2	Defensive Programmierung .....	689
21.2.1	double- und float-Werte richtig vergleichen .....	690
21.2.2	const und constexpr verwenden .....	691
21.2.3	Anweisungen nach for/if/while einklammern .....	691
21.2.4	int und unsigned/size_t nicht mischen .....	692
21.2.5	size_t oder auto statt unsigned int verwenden .....	692
21.2.6	Postfix++ mit Präfix++ implementieren .....	692
21.2.7	Ein Destruktor darf keine Exception werfen .....	693
21.2.8	explicit-Typumwandlungsoperator bevorzugen .....	693
21.2.9	explicit-Konstruktor für eine Typumwandlung bevorzugen .....	693
21.2.10	Leere Standardkonstruktoren vermeiden .....	693
21.2.11	Mit override Schreibfehler reduzieren .....	694
21.2.12	Kopieren und Zuweisung verbieten .....	694
21.2.13	Vererbung verbieten .....	695
21.2.14	Überschreiben einer virtuellen Methode verhindern .....	695
21.2.15	»Rule of zero« beachten .....	695

21.2.16	One Definition Rule.....	695
21.2.17	Defensiv Objekte löschen .....	696
21.2.18	Hängende Referenzen vermeiden .....	696
21.2.19	Speicherbeschaffung und -freigabe kapseln.....	696
21.2.20	Programmierrichtlinien einhalten .....	696
21.3	Exception-sichere Beschaffung von Ressourcen.....	696
21.3.1	Sichere Verwendung von <code>unique_ptr</code> und <code>shared_ptr</code> .....	697
21.3.2	So vermeiden Sie <code>new</code> und <code>delete</code> !.....	697
21.3.3	<code>shared_ptr</code> für C-Arrays korrekt verwenden.....	698
21.3.4	<code>unique_ptr</code> für C-Arrays korrekt verwenden .....	699
21.3.5	Exception-sichere Funktion .....	700
21.3.6	Exception-sicherer Konstruktor .....	701
21.3.7	Exception-sichere Zuweisung .....	701
21.4	Empfehlungen zur Thread-Programmierung.....	702
21.4.1	Warten auf die Freigabe von Ressourcen .....	702
21.4.2	Deadlock-Vermeidung .....	702
21.4.3	<code>notify_all</code> oder <code>notify_one</code> ?.....	703
21.4.4	Performance mit Threads verbessern?.....	704
<b>22</b>	<b>Von der UML nach C++ .....</b>	<b>705</b>
22.1	Vererbung .....	706
22.2	Interface anbieten und nutzen .....	706
22.3	Assoziation .....	708
22.3.1	Aggregation.....	712
22.3.2	Komposition .....	712
<b>23</b>	<b>Algorithmen für verschiedene Aufgaben .....</b>	<b>713</b>
23.1	Algorithmen mit Strings .....	714
23.1.1	String splitten .....	714
23.1.2	String in Zahl umwandeln .....	715
23.1.3	Zahl in String umwandeln .....	718
23.1.4	Strings sprachlich richtig sortieren .....	718
23.1.5	Umwandlung in Klein- bzw. Großschreibung.....	720
23.1.6	Strings sprachlich richtig vergleichen.....	722
23.1.7	Von der Groß-/Kleinschreibung unabhängiger Zeichenvergleich .....	722
23.1.8	Von der Groß-/Kleinschreibung unabhängige Suche .....	723
23.2	Textverarbeitung .....	724
23.2.1	Datei durchsuchen .....	724

---

23.2.2	Ersetzungen in einer Datei .....	726
23.2.3	Lines of Code (LOC) ermitteln .....	728
23.2.4	Zeilen, Wörter und Zeichen einer Datei zählen .....	729
23.2.5	CSV-Datei lesen .....	730
23.2.6	Kreuzreferenzliste .....	731
23.3	Operationen auf Folgen .....	733
23.3.1	Vereinfachungen .....	733
23.3.2	Folge mit gleichen Werten initialisieren .....	735
23.3.3	Folge mit Werten eines Generators initialisieren .....	736
23.3.4	Folge mit fortlaufenden Werten initialisieren .....	737
23.3.5	Summe und Produkt .....	737
23.3.6	Mittelwert und Standardabweichung .....	738
23.3.7	Skalarprodukt .....	739
23.3.8	Folge der Teilsummen oder -produkte .....	740
23.3.9	Folge der Differenzen .....	742
23.3.10	Kleinstes und größtes Element finden .....	743
23.3.11	Elemente rotieren .....	744
23.3.12	Elemente verwürfeln .....	746
23.3.13	Dubletten entfernen .....	746
23.3.14	Reihenfolge umdrehen .....	749
23.3.15	Stichprobe .....	749
23.3.16	Anzahl der Elemente, die einer Bedingung genügen .....	750
23.3.17	Gilt ein Prädikat für alle, kein oder wenigstens ein Element einer Folge? ..	751
23.3.18	Permutationen .....	752
23.3.19	Lexikografischer Vergleich .....	755
23.4	Sortieren und Verwandtes .....	757
23.4.1	Partitionieren .....	757
23.4.2	Sortieren .....	760
23.4.3	Stabiles Sortieren .....	761
23.4.4	Partielles Sortieren .....	762
23.4.5	Das n.-größte oder n.-kleinste Element finden .....	764
23.4.6	Verschmelzen (merge) .....	765
23.5	Suchen und Finden .....	767
23.5.1	Element finden .....	767
23.5.2	Element einer Menge in der Folge finden .....	768
23.5.3	Teilfolge finden .....	770
23.5.4	Teilfolge mit speziellem Algorithmus finden .....	771

23.5.5	Bestimmte benachbarte Elemente finden .....	772
23.5.6	Bestimmte aufeinanderfolgende Werte finden .....	773
23.5.7	Binäre Suche.....	774
23.6	Mengenoperationen auf sortierten Strukturen .....	777
23.6.1	Teilmengenrelation .....	778
23.6.2	Vereinigung .....	779
23.6.3	Schnittmenge .....	779
23.6.4	Differenz .....	780
23.6.5	Symmetrische Differenz .....	781
23.7	Heap-Algorithmen .....	782
23.8	Vergleich von Containern auch ungleichen Typs.....	786
23.8.1	Unterschiedliche Elemente finden .....	786
23.8.2	Prüfung auf gleiche Inhalte .....	788
23.9	Rechnen mit komplexen Zahlen: Der C++-Standardtyp complex.....	789
23.10	Vermischtes .....	791
23.10.1	Erkennung eines Datums.....	791
23.10.2	Erkennung einer IPv4-Adresse .....	793
23.10.3	Erzeugen von Zufallszahlen .....	794
23.10.4	for_each – auf jedem Element eine Funktion ausführen.....	799
23.10.5	Verschiedene Möglichkeiten, Container-Bereiche zu kopieren.....	800
23.10.6	Vertauschen von Elementen, Bereichen und Containern .....	803
23.10.7	Elemente transformieren .....	804
23.10.8	Ersetzen und Varianten .....	806
23.10.9	Elemente herausfiltern .....	807
23.10.10	Grenzwerte von Zahltypen .....	809
23.10.11	Minimum und Maximum .....	810
23.10.12	Wert begrenzen.....	812
23.10.13	ggT, kgV und Mitte .....	813
23.11	Parallelisierbare Algorithmen .....	814
<b>24</b>	<b>Datei- und Verzeichnisoperationen.....</b>	<b>815</b>
24.1	Übersicht .....	816
24.2	Pfadoperationen.....	817
24.3	Datei oder Verzeichnis löschen.....	818
24.4	Datei oder Verzeichnis kopieren .....	820
24.5	Verzeichnis anlegen .....	821
24.6	Datei oder Verzeichnis umbenennen .....	822

24.7	Verzeichnis anzeigen .....	823
24.8	Verzeichnisbaum anzeigen .....	824

## **Teil V: Die C++-Standardbibliothek .....825**

### **25 Aufbau und Übersicht .....827**

25.1	Auslassungen .....	830
25.2	Beispiele des Buchs und die C++-Standardbibliothek .....	831

### **26 Hilfsfunktionen und -klassen .....833**

26.1	Unterstützung der Referenzsemantik für R-Werte .....	833
26.2	Paare .....	835
26.3	Tupel .....	837
26.4	bitset .....	839
26.5	Indexfolgen .....	842
26.6	variant statt union .....	843
26.7	Funktionsobjekte .....	844
26.7.1	Arithmetische, vergleichende und logische Operationen .....	844
26.7.2	Binden von Argumentwerten .....	845
26.7.3	Funktionen in Objekte umwandeln .....	846
26.8	Templates für rationale Zahlen .....	848
26.9	Hüllklasse für Referenzen .....	850

### **27 Container ..... 851**

27.1	Gemeinsame Eigenschaften .....	853
27.1.1	Reversible Container .....	855
27.1.2	Initialisierungsliste (initializer_list) .....	856
27.1.3	Konstruktion an Ort und Stelle .....	857
27.2	Sequenzen .....	858
27.2.1	vector .....	859
27.2.2	vector<bool> .....	860
27.2.3	array .....	861
27.2.4	list .....	864
27.2.5	deque .....	867
27.3	Container-Adapter .....	868
27.3.1	stack .....	868
27.3.2	queue .....	870
27.3.3	priority_queue .....	871



27.4	Assoziative Container .....	873
27.4.1	Sortierte assoziative Container .....	875
27.4.2	Hash-Container .....	882
27.5	Sicht auf Container (span) .....	889
<b>28</b>	<b>Iteratoren .....</b>	<b>891</b>
28.1	Iterator-Kategorien .....	892
28.1.1	Anwendung von Traits .....	894
28.2	Abstand und Bewegen .....	897
28.3	Zugriff auf Anfang und Ende .....	898
28.3.1	Reverse-Iteratoren .....	899
28.4	Insert-Iteratoren .....	900
28.5	Stream-Iteratoren .....	902
<b>29</b>	<b>Algorithmen .....</b>	<b>905</b>
29.1	Algorithmen mit Prädikat .....	906
29.2	Übersicht .....	907
<b>30</b>	<b>Nationale Besonderheiten .....</b>	<b>911</b>
30.1	Sprachumgebung festlegen und ändern .....	912
30.1.1	Die locale-Funktionen .....	914
30.2	Zeichensätze und -codierung .....	915
30.3	Zeichenklassifizierung und -umwandlung .....	919
30.4	Kategorien .....	920
30.4.1	collate .....	920
30.4.2	ctype .....	921
30.4.3	numeric .....	923
30.4.4	monetary .....	924
30.4.5	messages .....	927
30.5	Konstruktion eigener Facetten .....	928
<b>31</b>	<b>String .....</b>	<b>931</b>
31.1	string_view für String-Literale .....	940
<b>32</b>	<b>Speichermanagement .....</b>	<b>943</b>
32.1	unique_ptr .....	943
32.2	shared_ptr .....	946
32.3	weak_ptr .....	948
32.4	new mit Speicherortangabe .....	949

<b>33</b>	<b>Ausgewählte C-Header .....</b>	<b>951</b>
33.1	<cassert> .....	951
33.2	<cctype> .....	952
33.3	<cmath> .....	953
33.4	<csddef> .....	954
33.5	<csdlib> .....	954
33.6	<cstring> .....	955
33.7	<ctime> .....	957
<b>A</b>	<b>Anhang .....</b>	<b>959</b>
A.1	ASCII-Tabelle .....	959
A.2	C++-Schlüsselwörter .....	961
A.3	Compilerbefehle .....	962
A.3.1	Optimierung .....	963
A.4	Rangfolge der Operatoren .....	963
A.5	C++-Attribute für den Compiler .....	965
A.6	Lösungen zu den Übungsaufgaben .....	966
A.7	Änderungen in der 7. Auflage .....	976
	<b>Glossar .....</b>	<b>977</b>
	<b>Literaturverzeichnis .....</b>	<b>987</b>
	<b>Register .....</b>	<b>991</b>

# Vorwort

Diese Auflage unterscheidet sich von der vorherigen durch eine gründliche Überarbeitung und die Umstellung auf den 2023 von der zuständigen ISO/IEC-Arbeitsgruppe verabschiedeten C++-Standard. Abschnitt A.7 bietet eine Übersicht der in diesem Buch berücksichtigten Änderungen. Das Buch ist konform zum C++23-Standard, ohne den Anspruch auf Vollständigkeit zu erheben – das Standarddokument [ISOC++] umfasst mehr als 2100 Seiten. Sie finden in diesem Buch eine verständliche und mit vielen Beispielen angereicherte Einführung in die Sprache, unabhängig vom Betriebssystem.

## ■ Für wen ist dieses Buch geschrieben?

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

## ■ Ein umfassendes Handbuch

Die ersten zehn Kapitel führen in die Sprache ein, die folgenden behandeln fortgeschrittene Themen. Die sofortige praktische Umsetzung des Gelernten anhand von leicht nachvollziehbaren Beispielen steht im Vordergrund. Klassen und Objekte, Templates und Exceptions sind Ihnen bald keine Fremdworte mehr. Es gibt 99 Übungsaufgaben – mit Musterlösungen im Anhang und zum Download. Durch das Studium dieser Kapitel werden aus Neulingen bald Fortgeschrittene – und mithilfe der weiteren Kapitel Experten.

## C++ in praktischen Anwendungen

Sie finden kurze Einführungen in die Themen Programmierung paralleler Abläufe, Netzwerk-Programmierung einschließlich eines kleinen Webservers, Datenbank-Anbindung, grafische Benutzungsoberflächen und Zugriff auf die KIs ChatGPT und DALL·E 2. Durch den Einsatz der Boost-Library und des Qt-Frameworks wird größtmögliche Portabilität erreicht.

### Softwareentwicklung ist nicht nur Programmierung

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

### ■ Moderne Programmiermethodik

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

### ■ Wie benutzen Sie dieses Buch am besten?

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

### ■ Wo finden Sie was?

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

### ■ Zu guter Letzt

Allen Menschen, die dieses Buch durch Hinweise und Anregungen verbessern halfen, sei an dieser Stelle herzlich gedankt. Insbesondere Prof.Dr .Ulrich Eisenecker danke ich für seine hilfreichen Kommentare. Frau Irene Weilhart vom Hanser Verlag und dem Lektorat danke ich für die gute Zusammenarbeit.

Bremen, im Juni 2023

Ulrich Breymann

## ■ 1.4 Das erste Programm

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

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

```
#include <iostream>

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

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

**Listing 1.2:** Programmentwurf

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

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




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

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

Kommentare werden einschließlich der Kennungen vom Compiler vollständig ignoriert. Ein Kommentar, der mit /\* beginnt, wird mit der ersten \*/-Zeichenkombination beendet,

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

### Ein Programm ist ein Text!

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

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



### Hinweis

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

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

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

int main()
{
    int summand1 {0};
    int summand2 {0};
    // Lies zwei Zahlen ein
    cout << "_Zwei_ganze_Zahlen_eingeben:";
    cin >> summand1 >> summand2;
    /* Berechne die Summe beider Zahlen
    */
    int summe = summand1 + summand2;
    // Zeige das Ergebnis auf dem Bildschirm an
    cout << "Summe=" << summe << '\n';
    return 0;
}
```

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

<code>#include&lt;iostream&gt;</code>	Einbindung der Ein-/Ausgabefunktionen. Diese Zeile muss in jedem Programm stehen, das Eingaben von der Tastatur erwartet oder Ausgaben auf den Bildschirm bringt. Sie können sich vorstellen, dass der Compiler beim Übersetzen des Programms an dieser Stelle erst alle zur Ein- und Ausgabe notwendigen Informationen liest. Details folgen in Abschnitt 2.3.
<code>using namespace std;</code>	Der Namensraum (englisch <i>namespace</i> ) <code>std</code> wird benutzt. Schreiben Sie es einfach in jedes Programm an diese Stelle und haben Sie Geduld: Erklärungen folgen auf den Seiten 63 und 151.
<code>int main()</code>	<p><code>main()</code> ist die Funktion, mit der jedes Programm beginnt (es gibt auch andere Funktionen). Der zu <code>main()</code> gehörende Programmcode wird durch die geschweiften Klammern <code>{</code> und <code>}</code> eingeschlossen.</p> <p>Ein mit <code>{</code> und <code>}</code> begrenzter Bereich heißt <i>Block</i>. Mit <code>int</code> ist gemeint, dass die <code>main()</code>-Funktion nach Beendigung eine Zahl vom Typ <code>int</code> (= ganze Zahl) an das Betriebssystem zurückgibt. Dazu dient die unten beschriebene <code>return</code>-Anweisung. Normalerweise – das heißt bei ordnungsgemäßigem Programmablauf – wird die Zahl 0 zurückgegeben. Andere Zahlen können über das Betriebssystem einen Fehler signalisieren.</p>
<code>int summand1 {0};</code> <code>int summand2 {0};</code> <code>int summe = ...</code>	<p><b>Deklaration</b> (Bekanntmachung) von Objekten: Mitteilung an den Compiler, der ab jetzt die Namen <code>summand1</code>, <code>summand2</code> und <code>summe</code> innerhalb des Blocks <code>{ }</code> kennt. Hier wird gleichzeitig Speicherplatz bereitgestellt. Es gibt verschiedene Zahlentypen in C++. Mit <code>int</code> sind ganze Zahlen gemeint: <code>summe</code>, <code>summand1</code>, <code>summand2</code> sind ganze Zahlen. Oft ist es sinnvoll, einen Anfangswert festzulegen, etwa 0, wie hier bei <code>summand1</code> und <code>summand2</code>.</p>
<code>;</code>	Ein Semikolon beendet jede Deklaration und jede Anweisung (aber keine Verbundanweisung, siehe weiter unten).
<code>cin</code>	Eingabe: Der Doppelpfeil zeigt hier in Richtung des Objekts, das ja von der Tastatur einen neuen Wert aufnehmen soll. Die Information fließt von der Eingabe <code>cin</code> zum Objekt <code>summand1</code> beziehungsweise zum Objekt <code>summand2</code> .
<code>cout</code>	Ausgabe: <code>cout</code> (Abkürzung für <i>character out</i> oder <i>console out</i> ) ist die Standardausgabe. Der Doppelpfeil deutet an, dass alles, was rechts davon steht, zur Ausgabe <code>cout</code> gesendet wird, zum Beispiel <code>cout &lt;&lt; summand1;</code> . Wenn mehrere Dinge ausgegeben werden sollen, sind sie durch <code>&lt;&lt;</code> zu trennen.

"Text"	beliebige Zeichenkette, die die Anführungszeichen selbst nicht enthalten darf, weil sie als Anfangs- beziehungsweise Endmarkierung einer Zeichenfolge dienen. Wenn die Zeichenfolge die Anführungszeichen enthalten soll, sind diese als \" zu schreiben: cout << \"C++\" ist der Nachfolger von \"C\"!"; erzeugt die Bildschirmausgabe "C++" ist der Nachfolger von "C"!
'\n'	Die Ausgabe des Zeichens \n bewirkt eine neue Zeile.
return 0;	Unser Programm läuft einwandfrei, es gibt daher 0 an das Betriebssystem zurück. Diese Anweisung darf in der main()-Funktion fehlen, dann wird automatisch 0 zurückgegeben.

<iostream> ist ein Header. Dieser aus dem Englischen stammende Begriff (head = dt. Kopf) drückt aus, dass Zeilen dieser Art am Anfang eines Programmtextes stehen. Der Begriff wird im Folgenden verwendet, weil es keine gängige deutsche Entsprechung gibt. Einen Header mit einem Dateinamen gleichzusetzen, ist meistens richtig, nach dem C++-Standard aber nicht zwingend. *Hinweis:* Tatsächlich wird ein Programm zunächst von einem vorgeschalteten *Präprozessor* bearbeitet, der das Ergebnis der Bearbeitung an den eigentlichen Compiler weiterleitet. Wenn im Folgenden also von »Compiler« die Rede ist, ist meistens auch der Präprozessor gemeint.

summand1, summand2 und summe sind veränderliche Daten und heißen Variablen. Sie sind Objekte eines vordefinierten Grunddatentyps für ganze Zahlen (int), mit denen die üblichen Ganzzahloperationen wie + und - durchgeführt werden können. Der Begriff »Variable«<sup>3</sup> wird für ein veränderliches Objekt gebraucht. Für Variablen gilt:

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

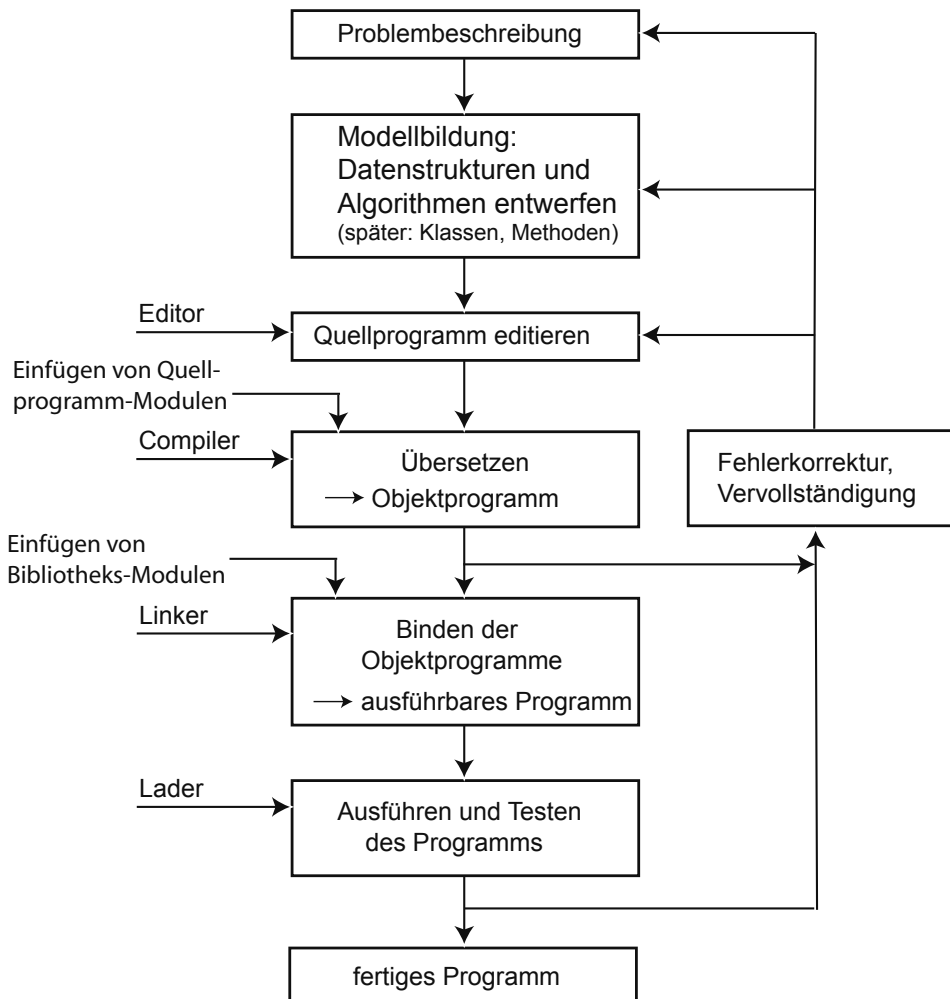
Der Speicherplatz wird vom Compiler reserviert. Man spricht dann von der *Definition* der Objekte. Definition und Deklaration werden unterschieden, weil es auch Deklarationen ohne gleichzeitige Definition gibt, doch davon später mehr. Zunächst sind die Deklarationen zugleich Definitionen. Abbildung 1.2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

<sup>3</sup> Anmerkung für Menschen mit Vorkenntnissen: Aus reiner C++-Sicht ist eine Variable eine Deklaration eines Objekts (oder einer Referenz) und sagt nichts darüber aus, ob es konstant oder veränderlich ist. Die Eigenschaft »konstant« wird durch das Schlüsselwort const bewirkt.



	Adresse	Name
	...	
	4089	10123
	2	10124
Inhalt →	100	10125
	102	10126
	1127	10127
	...	

summand1  
summand2  
summe

**Abbildung 1.1:** Speicherbereiche mit Adressen**Abbildung 1.2:** Erzeugung eines lauffähigen Programms

Der Compiler ist selbst ein Programm, das bereits in maschinenverständlicher Form vorliegt und speziell für diese Übersetzung zuständig ist. Nach Eingabe des Programmtextes mit dem Editor können Sie den Compiler starten.

Ein Programmtext wird auch »Quelltext« oder »Quellcode« (englisch *source code*) genannt. Der Compiler erzeugt aus dem Quellcode den Objektcode, der noch nicht ausführbar ist. Hinter den einfachen Anweisungen `cin >> ...` und `cout << ...` verbergen sich eine Reihe von Aktivitäten wie die Abfrage der Tastatur und die Ansteuerung des Bildschirms, die nicht speziell programmiert werden müssen, weil sie schon in vorübersetzter Form in Bibliotheksdateien vorliegen. Die Aufrufe dieser Aktivitäten im Programm müssen mit den dafür vorgesehenen Algorithmen in den Bibliotheksdateien zusammengebunden werden, eine Aufgabe, die der *Linker* übernimmt, auch *Binder* genannt. Der Linker bindet Ihren Objektcode mit dem Objektcode der Bibliotheksdateien zusammen und erzeugt daraus ein ausführbares Programm, das nun gestartet werden kann. Der Aufruf des Programms bewirkt, dass der *Lader*, eine Funktion des Betriebssystems, das Programm in den Rechnerspeicher lädt und startet. Diese Schritte werden stets ausgeführt, auch wenn sie in den Programmentwicklungsumgebungen verborgen ablaufen. Bibliotheksmodule können auch während der Programmausführung geladen werden (nicht im Bild dargestellt).

### Wie bekomme ich ein Programm zum Laufen?

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

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

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

```
g++ -c summe.cpp           compilieren (summe.o wird erzeugt)
```

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

```
g++ -o summe.exe summe.cpp beide Schritte zusammengefasst
```

Die Objektdateien können je nach System die Endung *.o* oder *.obj* tragen. Wenn `g++ -std=c++23 -o summe.exe summe.cpp` geschrieben wird, soll der Compiler den C++-Standard 2023 verwenden. Das ist bei diesem einfachen Programm nicht notwendig, weil es keine der neueren Eigenschaften nutzt.

Eine integrierte Entwicklungsumgebung lässt die genannten Schritte per Tastendruck ablaufen, wie Sie gleich sehen.

### switch/case-lokale Variablen

Wenn in der case-Anweisung eine lokale Variable angelegt wird, muss sie in einem eigenen Gültigkeitsbereich (scope) mit geschweiften Klammern {} gekapselt werden:

```
case 'D':
    zahl = 500;
    {
        auto x{zahl*10};           // Beginn des lokalen Gültigkeitsbereichs.
        cout << x << '\n';
    }                             // Die Gültigkeit von x endet hier.
    break;
```

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

## 1.8.5 Wiederholungen

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

### Schleifen mit while

Abbildung 1.8 zeigt die Syntax von while-Schleifen.

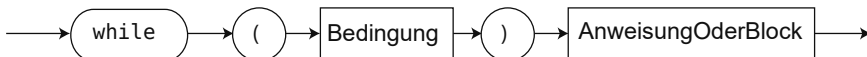
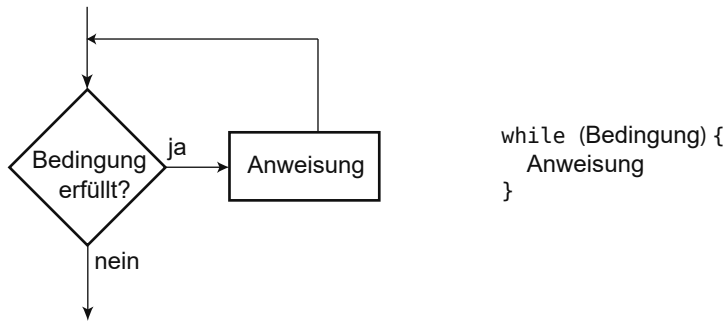


Abbildung 1.8: Syntaxdiagramm einer while-Schleife

*AnweisungOderBlock* ist wie auf Seite 66 definiert. Die Bedeutung einer while-Schleife ist: Solange die Bedingung wahr ist, die Auswertung also ein Ergebnis true oder ungleich 0 liefert, wird die Anweisung bzw. der Block ausgeführt. Die Bedingung wird auf jeden Fall zuerst geprüft. Wenn die Bedingung von vornherein unwahr ist, wird die Anweisung gar nicht erst ausgeführt (siehe Abbildung 1.9).

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

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



**Abbildung 1.9:** Flussdiagramm für eine `while`-Anweisung

### Beispiele

- Unendliche Schleife:

```
while (true)
    Anweisung
```

- Anweisung wird nie ausgeführt (unerreichbarer Programmcode):

```
while (false)
    Anweisung
```

- Summation der Zahlen 1 bis 99:

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

- Berechnung des größten gemeinsamen Teilers  $\text{ggT}(x, y)$  für zwei natürliche Zahlen  $x$  und  $y$  nach Euklid. Es gilt:

- $\text{ggT}(x, x)$ , also  $x == y$ : Das Resultat ist  $x$ .

- $\text{ggT}(x, y)$  bleibt unverändert, falls die größere der beiden Zahlen durch die Differenz ersetzt wird, also  $\text{ggT}(x, y) == \text{ggT}(x, y-x)$ , falls  $x < y$ .

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

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

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

int main()
{
    int x {0};
    int y {0};
    cout << "2_Zahlen_>_0_eingeben_:";
    cin >> x >> y;
    cout << "Der_GGT_von_" << x << "_und_" << y << "_ist_";
    while (x != y) {
```

```

    if (x > y) {
        x -= y;
    }
    else {
        y -= x;
    }
}
cout << x << '\n';
}

```

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

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

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

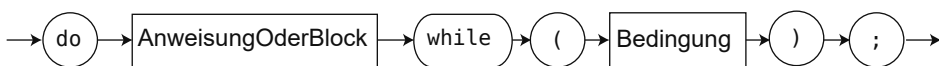


### Tipp

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

## Schleifen mit *do while*

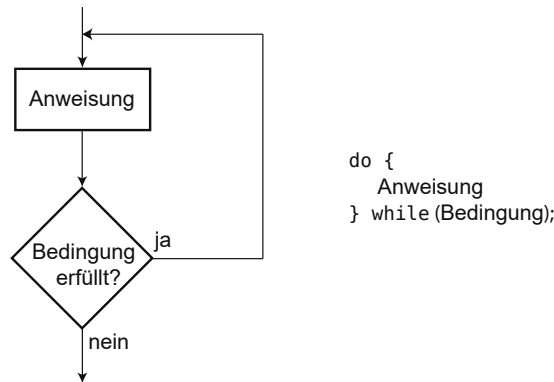
Abbildung 1.10 zeigt die Syntax einer *do while*-Schleife.



**Abbildung 1.10:** Syntaxdiagramm einer *do while*-Schleife

*AnweisungOderBlock* ist wie auf Seite 66 definiert. Die Anweisung oder der Block einer *do while*-Schleife wird ausgeführt, und *erst anschließend* wird die Bedingung geprüft. Ist sie wahr, wird die Anweisung ein weiteres Mal ausgeführt usw. Die Anweisung wird also mindestens einmal ausgeführt. Im Flussdiagramm ist die Anweisung ein Block (siehe rechts in der Abbildung 1.11).

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



**Abbildung 1.11:** Flussdiagramm für eine `do while`-Anweisung

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

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

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

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

int main()
{
    cout << "Berechnung der ersten Primzahl, die >="
          "der eingegebenen Zahl ist\n";
    // Hinweis: Mehrere, durch " getrennte Texte ergeben eine lange Zeile in der Ausgabe.
    long zahl {0L};
    // do while-Schleife zur Eingabe und Plausibilitätskontrolle
    do {
        // Abfrage, solange zahl ≤ 3 ist
        cout << "Zahl >_3 eingeben: ";
        cin >> zahl;
    } while (zahl <= 3);

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

```

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

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

```

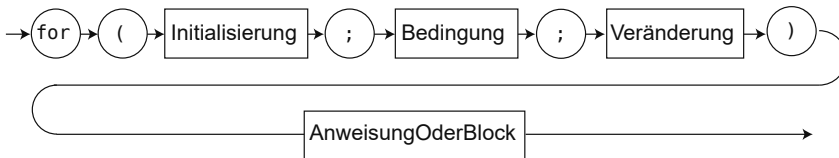


### Tipp

In do-while-Schleifen wird die Bedingung erst am Ende abgefragt, sie ist beim Lesen also nicht sofort zu finden. Deswegen verwenden Sie do-while nur, wenn Sie erreichen wollen, dass eine Schleife mindestens einmal durchlaufen werden soll.

### Schleifen mit for

Die letzte Art von Schleifen ist die for-Schleife. Sie wird häufig eingesetzt, wenn die Anzahl der Wiederholungen vorher feststeht, aber das muss durchaus nicht so sein. Abbildung 1.12 zeigt die Syntax einer for-Schleife.



**Abbildung 1.12:** Syntaxdiagramm einer for-Schleife

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

```

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

```

Bei der Abarbeitung werden die folgenden Schritte durchlaufen:

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

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

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

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

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

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

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

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

int main()
{
    cout << "Fakultät_berechnen._Zahl_>=_0?: ";
    int n {0};
    cin >> n;

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

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

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

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



### Äquivalenz von for und while

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

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

```
    Anweisung
```

ist äquivalent zu:

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

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

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

### float- oder double-Laufvariablen vermeiden!

Wegen der Rechenungenauigkeit kann es bei nicht-integralen Typen wie double oder float zu nicht vorhersagbarem Verhalten kommen. Das Beispiel:

```
for (double d = 0.4; d <= 1.2; d += 0.4) {
    cout << d << '\n';
}
```

lässt auf den ersten Blick die Ausgabe 0.4, 0.8, 1.2 erwarten, tatsächlich werden auf meinem System nur die zwei Zahlen 0.4 und 0.8 ausgegeben. Wenn ich jedoch

```
for (double d = 0.5; d <= 1.5; d += 0.5) {
    cout << d << '\n';
}
```

ausführe, werden wie erwartet die drei Zahlen 0.5, 1 und 1.5 angezeigt. Der Grund liegt darin, dass 0.5 im Binärsystem exakt darstellbar ist, 0.4 jedoch nicht. Schon ein Unterschied im letzten Bit lässt den Vergleich auf Gleichheit scheitern. Ganz ungünstig kann sich die Prüfung auf Ungleichheit mit != auswirken:

Die Ausgabe des Programms ist:

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

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

## ■ 3.7 Wie kommt man zu Klassen und Objekten? Ein Beispiel

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

Es geht hier um ein Programm, das zu einer gegebenen Personalnummer den Namen heraus sucht. Ähnlichkeiten mit der Aufgabe 1.25 von Seite 112 sind beabsichtigt. Gegeben sei eine Datei *daten.txt* mit den Namen und den Personalnummern der Mitarbeiter. Dabei folgt auf eine Zeile mit dem Namen eine Zeile mit der Personalnummer. Das `#`-Zeichen ist die Endekennung. Der Inhalt der Datei ist:

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

### Einige Analyse-Überlegungen

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

1. In der Analyse geht es zunächst einmal darum, den typischen Anwendungsfall (englisch *use case*) *in der Sprache des (späteren Programm-)Anwenders* zu beschreiben. Ein ganz konkreter Anwendungsfall, Szenario genannt, ist ein weiteres Hilfsmittel zum Verständnis dessen, was das Programm tun soll.
2. Im zweiten Schritt wird versucht, beteiligte Objekte, ihr Verhalten und ihr Zusammenwirken zu identifizieren.



### Anwendungsfall (use case)

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

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



### Szenario

Das Programm wird gestartet und gibt aus:

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

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

### Objekte und Operationen identifizieren

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

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

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

Diese drei Punkte und die Kenntnis der Datei führen zu entsprechenden Schlussfolgerungen. Dabei sind im ersten Schritt die Substantive (Hauptworte) als Kandidaten für Klassen

zu sehen und Verben (Tätigkeitsworte) als Methoden. Passivkonstruktionen sollen dabei vorher stets in Aktivkonstruktionen verwandelt werden, d.h. *ausgeben* ist besser als *die Ausgabe erfolgt*.

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

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

**Listing 3.36:** main-Programm zur Personalverwaltung (*cppbuch/k3/personalverwaltung/main.cpp*)

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

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

Die Klasse *Person* ist einfach zu entwerfen:

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

```
#ifndef PERSON_H
```

```

#define PERSON_H
#include <string>

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

    [[nodiscard]] auto getName() const { return name; }

    [[nodiscard]] auto getPersonalnummer() const { return personalnummer; }

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

```

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

**Listing 3.38:** Klasse Personalverwaltung (*cppbuch/k3/personalverwaltung/Personalverwaltung.h*)

```

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

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

    void ausgeben() const;

    void dialog() const;

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

```

Für die Implementierung der Methoden der Klasse Personalverwaltung muss man sich mehr Gedanken machen. Das überlasse ich Ihnen (siehe die nächste Aufgabe)! Die Lösung dürfte aber nicht schwer sein, wenn Sie die Aufgabe 1.25 von Seite 112 gelöst oder deren Lösung nachgesehen haben.



## Übungen

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

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

# 22

## Von der UML nach C++

Dieses Kapitel behandelt die folgenden Themen:

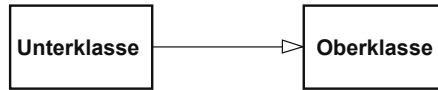
---

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

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

## ■ 22.1 Vererbung

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



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

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

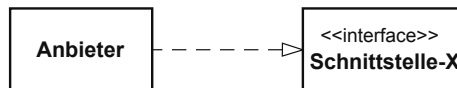
**Listing 22.1:** Syntaktische Repräsentation der Vererbung

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

## ■ 22.2 Interface anbieten und nutzen

### Interface anbieten

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



**Abbildung 22.2:** Interface-Anbieter

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

<sup>1</sup> Die UML erlaubt Bindestriche in Namen, C++ nicht.

**Listing 22.2:** Schnittstellenklasse

```

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

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

```

**Interface nutzen**

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

**Abbildung 22.3:** Interface-Nutzer

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

**Listing 22.3:** Nutzer der Schnittstelle

```

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

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

```



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

## ■ 22.3 Assoziation

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

### Einfache gerichtete Assoziation

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



**Abbildung 22.4:** Gerichtete Assoziation

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

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

```

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

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

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

### Gerichtete Assoziation mit Multiplizität

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



**Abbildung 22.5:** Gerichtete Assoziation mit Multiplizitäten

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

**Listing 22.5:** Gerichtete Assoziation mit Multiplizität: Ein Fan kennt Popstars, aber nicht umgekehrt.

```

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

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

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

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

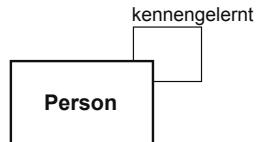
### Einfache ungerichtete Assoziation

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



**Abbildung 22.6:** Ungerichtete Assoziation

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



**Abbildung 22.7:** Rekursive Assoziation

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

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

```

#include "Person.h"

int main()
{
    Person mabuse("Dr._Mabuse");
    Person klicko("Witwe_Klicko");
    Person holle1("Frau_Holle");
    Person holle2("Frau_Holle");           // eine Namensvetterin!
    mabuse.lerntkennen(klicko);
    holle1.lerntkennen(klicko);
    holle1.lerntkennen(holle2);
    mabuse.bekannteZeigen();
    klicko.bekannteZeigen();
    holle1.bekannteZeigen();
}
  
```

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

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

```

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

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

    auto getName() const
    {
        return name;
    }

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

    void bekannteZeigen() const
    {
        std::cout << "Die_Bekannten_von_" << getName() << "_sind:\n";
        for (const auto& bekannt : bekannte) {
            std::cout << bekannt->getName() << '\n';
        }
    }

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

```

**Hinweis**

Die Methode `insert()` eines Sets gibt ein `pair`-Objekt zurück. Das ist eine Struktur mit den Elementen `first` und `second`. `second` ist ein Wahrheitswert, der angibt, ob das Einfügen stattgefunden hat. Wenn nein, war das Element schon vorhanden. `first` ist ein Iterator auf das Element, ob gerade eingefügt oder schon vorhanden gewesen.

### ■ 22.3.1 Aggregation

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

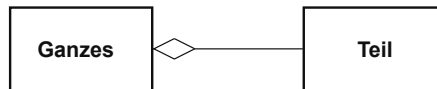


Abbildung 22.8: Aggregation

### ■ 22.3.2 Komposition

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

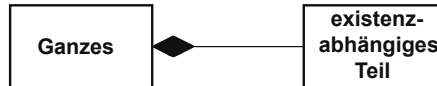


Abbildung 22.9: Komposition

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

Listing 22.8: Umsetzung der Komposition

```

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

# Register

## Symbole

\* 45, 218, 463  
\*= 45, 383  
+, +=, -, -= 45  
++ 45, 385, 459, 463  
, 59, 81, 519  
-> 221  
->\* 265  
-- 45, 388  
.\* 265  
... *siehe* Ellipse  
/, /= 45  
/\* ... \*/ 30  
// 30  
:: 62, 180, 315  
::\* 265  
; 32, 70  
<, <=, >, >= 45, 57  
<=> 45, 398  
<< 45, 107, 258, 371, 427  
<<=, >>= 45  
=, == 45, 57  
== 372  
>> 45, 105, 258, 425, 449  
?: 70  
[ ] 222, 225, 377  
[ ][ ]  
Matrixklasse 415

Zeigerdarstellung 250

# 142  
%, %= 45  
& Adress-Operator 124  
&, &= Bit-Operatoren 45  
&& logisches UND 57  
&& Shell 658  
&& R-Wert 480  
\ 31, 55, 140  
\0 227, 231–233  
\", \a, \b 55  
\f, \n, \r, \t, \v 55, 105  
\x 55  
\\ 55  
~ 45  
^ 45  
|= 45  
|| 57  
! 45, 57  
!= 57, 463  
\$<, \$^, \$@ 655  
@D, @F 665  
" 33, 227, 228

## A

abgeleitete Klasse 302, 310, 314  
und virtueller Destruktor 326  
Abhängigkeit (make) 652

- automatische Ermittlung 657
- abort() 955
- abs() 51, 789, 790, 953, 954
- abstrakte Klasse 319, 977
- abstrakter Datentyp 175, 977
- accumulate() 737
- Achterbahnzahlen 98
- acos() 790, 953
- acosh() 790
- Adapter, Iterator- 899
- Additionsoperator 369
- adjacent\_difference() 742
- adjacent\_find() 772
- adjustfield 435
- Adresse 218
  - symbolische 33
- Adressoperator 219
- advance() 897
- Aggregat 89, 224, 862
- Aggregation 712, 977
- Aktualparameter 118
- <algorithm> 713, 803, 905
- Algorithmus 28, 459
  - accumulate() 737
  - adjacent\_difference() 742
  - adjacent\_find() 772
  - all\_of, any\_of 751
  - binary\_search() 774
  - clamp() 812
  - copy(), copy\_backward() 800
  - copy\_if(), copy\_n() 802
  - count(), count\_if() 750
  - equal() 788
  - equal\_range() 776
  - exclusive\_scan() 741
  - fill(), fill\_n() 735
  - find(), find\_if(), find\_if\_not() 767
  - find\_end() 771
  - find\_first\_of() 768
  - for\_each(), for\_each\_n() 799
  - generate(), \_n() 736
  - includes() 778
  - inclusive\_scan() 740
  - inner\_product() 739
  - inplace\_merge() 766
  - iota() 737
  - is\_heap(), is\_heap\_until() 786
  - is\_partitioned() 758
  - is\_permutation() 754
  - is\_sorted(), is\_sorted\_until() 760
  - iter\_swap() 803
  - lexicographical\_compare() 755
  - lexicographical\_compare\_three\_way() 756
  - lower\_bound() 775
  - make\_heap() 784
  - make\_pair() 836
  - make\_tupel() 837
  - max(...) 810
  - max\_element() 743
  - merge() 765
  - mergesort() 766
  - min(...) 810
  - min\_element() 743
  - minmax() 811
  - minmax\_element() 743
  - mismatch() 786
  - move(), move\_backward() 834
  - next\_permutation() 753
  - none\_of 751
  - nth\_element() 764
  - partial\_sort(), \_copy 762
  - partial\_sum() 740
  - partition() 757
  - partition\_copy(), partition\_point() 758
  - pop\_heap() 783
  - prev\_permutation() 752
  - push\_heap() 784
  - remove(), \_if(), \_copy(), \_copy\_if() 807
  - replace(), \_if(), \_copy(), \_copy\_if() 806
  - reverse(), reverse\_copy() 749
  - rotate(), rotate\_copy() 744
  - sample() 749
  - search() 770, 771
  - search\_n() 773
  - set\_difference() 780
  - set\_intersection() 779
  - set\_symmetric\_difference() 781
  - set\_union() 779

- shuffle() 746
- sort() 760
- sort\_heap() 785
- split() 714
- stable\_partition() 757
- stable\_sort() 761
- swap(), swap\_ranges() 803
- transform() 804
- transform\_exclusive\_scan() 741
- transform\_inclusive\_scan() 741
- unique(), unique\_copy() 746
- upper\_bound() 775
- Alias 124, 219, 221
  - \*this 244
- alignment\_of, alignof() 527
- all\_of 751
- allgemeiner Konstruktor 184
- allocator 830
- alternative Funktions-Syntax 283
- and\_then() 364
- Anführungszeichen 33, 227, 228
- Anker (bei regulärem Ausdruck) 538
- anonymer Namespace 154
- Anweisung 64
- any() (Bitset) 842
- any\_of 751
- app 446
- append()
  - Pfad 818
  - String 934
- apply() 839
- Äquivalenz 399, 775
  - klasse 674
- arg() 789, 790
- argc 243
- Argument *siehe* Parameter
- Argument Dependent Lookup (ADL) 978
- argv[] 243
- Arithmetik
  - mit Iteratoren 462
  - mit Zeigern 224
- arithmetische Operatoren 45
- Arität (Template) 512
- Array
  - char 229
  - von C-Strings 229
  - dynamisches 234, 374
  - Freigabe 238
  - als Funktionsparameter 240, 248
  - vs. Zeiger 223
  - zweidimensionales
    - Matrixklasse 415, 416
- <array> 852, 862
- array 95, 862
- Array2d 251
- ASCII 915
  - Dateien 258
  - Tabelle 55, 959
- asctime() 958
- asin() 790, 953
- asinh() 790
- assert() 143
- assign() 858, 935
- assignable\_from 534
- Assoziation (UML) 708
- Assoziativität von Operatoren 58, 963
- async() 575
- at() 91, 96, 859, 862, 867, 878, 887, 933
- atan() 790, 953
- atan2() 953
- atanh() 790
- ate 446
- atexit() 955
- atof(), atoi(), atol() 955
- Atom-Uhr 618
- atomic 572
- atomic\_ref 574
- Attribut
  - Compilersteuerung 965
  - einer Klasse 177, 978
- Aufforderung 978
- Aufzählungstyp 85
- Ausdruck
  - Auswertung 58, 59
  - Definition 39
  - mathematischer 51, 131
- Ausgabe 104, 107, 427
  - benutzerdefinierter Typen 428
  - Datei- 108, 423
  - Formatierung
    - mit Flags 433
    - mit std::format() 429



- mit Manipulatoren 437
- Weite der 431, 433
- Ausgabeoperator 371, 427
- Ausnahme 349, 978
- Auswertungsreihenfolge 51, 59, 964
- auto 44, 97, 100, 468, 692
  - als Rückgabetyt 159, 182, 318
- auto(), auto{} 809
- auto[] 102, 837
- automatische Variable 150
  - make 655
- Autotools (GNU) 671

## B

- back() 859, 862, 864, 867, 870, 933
- back\_inserter(), -\_insert\_iterator 901
- Backslash
  - Zeichenkonstante \ 55
  - Zeilenfortsetzung 140
- Backspace 55
- bad() 445
- bad\_alloc 355
- badbit 443
- bad\_cast 340, 355
- bad\_function\_call 355
- bad\_typeid 343, 355
- bad\_weak\_ptr 355
- base() 899
- basefield 435
- basic\_string 931
- basic\_-Streamklassen 424
- Basisklasse 302
  - virtuelle B. 336
  - Konstruktor 334
  - Subobjekt 338
  - und virtueller Destruktor 326
- Bedingungsausdruck 67
- Bedingungsoperator ?: 70
- beg 447
- begin()
  - Container 465, 854
  - Namespace std 226, 855, 898
  - string 275, 932
  - vector 226
- Belegungsgrad 883
- benutzerdefinierte
  - Datentypen 85, 88
  - Klassen 371
  - Literale 412
  - Typen (Ausgabe) 428
  - Typen (Eingabe) 449
- Benutzungszählung 946
- Bereichsnotation 856, 980
- Bereichsoperator :: 62, 180, 315
  - namespace 152
- Bibliothek
  - C 167, 951
  - C++ 827
- Bibliotheksmodul 146
  - dynamisch 668
  - statisch 666
- Bidirectional-Iterator 893
- Big Three (Regel) 256
- Big Three/Five/Zero (Regel) 496
- binary 111, 446
- binary\_search() 774
- bind 845
- Binden 146
  - dynamisches *siehe* dynamisches B.
  - statisches *siehe* statisches Binden
- Binärdatei 260
- binäre Ein-/Ausgabe 257
- binärer Operator 369
  - optimiert 493
- binäres Prädikat 748, 788, 906
- binäre Zahlendarstellung 46
- Binärzahl 42
- Bit
  - feld 103
  - Operatoren 45, 46
  - pro Zahl 40
  - Verschiebung 46
- bit\_cast 104
- bitset 839
- bitweises ODER, UND, XOR 45
- Block 30, 32, 61, 65, 209
  - und dyn. Objekte 237
- bool 57
- boolalpha 57, 434, 438
- break 71, 82
- bsearch() 955
- Bstatic (Makefile) 667

Bucket 883  
 bucket() 885  
 Byte 54  
     std::byte 87, 954  
     Reihenfolge 618  
**C**  
 C++ Core Guidelines 113  
 C++-Schlüsselwörter 961  
 call wrapper 557  
 Callback-Funktion 262, 846  
 canonical() (Filesystem) 818  
 capacity() 860, 934  
 capturing group 536  
 C-Array 222  
     mehrdimensionales 246  
 case 71  
     lokale Variablen 74  
 <cassert> 143, 951  
 cast *siehe* Typumwandlung  
 catch 350  
 cbegin()  
     Container 854  
     Namespace std 855, 898  
     string 933  
 <cctype> 804, 952  
 cdecl 269  
 ceil() 953  
 cend()  
     Container 854  
     Namespace std 855, 898  
     string 933  
 cerr 105, 424  
 C-Funktionen einbinden 169  
 char 54  
 char\* 227  
 char\* const vs. const\* char 221  
 char\_traits 228, 281, 529  
 char8\_t, char16\_t, char32\_t 411, 916  
 ChatGPT 635  
 C-Header 951  
 <chrono> 385, 548  
 cin 32, 105, 424, 444  
 clamp() 812  
 class 178, 308  
 class (bei Template-Parametern) 161

clear() 445, 858, 874, 936  
 Client-Server  
     Beziehung 179  
     und callback 262  
 clock(), clock\_t 958  
 clog 105, 424  
 close() 108  
 CMake 671  
 <cmath> 51, 951, 953  
 co\_await 583  
 code bloat 298  
 Code-Formatierung 113  
 collate 920  
 combine() 914  
 <compare> 399  
 compare() 921, 938  
 Compilationsmodell 299  
 Compiler 29, 30, 33, 145, 177, 179  
     -befehle 962  
     -direktiven 137  
     und Templates 164  
     Typumwandlung 201  
 <complex> 789  
 Computerarithmetik 51  
 Concepts 530  
 conditional 529  
 configure 671  
 conj() 790  
 connect() 588  
 const 53  
     correctness 181  
     Elementfunktionen 178, 180  
     globale Konstante 151  
 const& *siehe* Referenz auf const  
 const char\* vs. char\* const 221  
 consteval 159  
 constexpr  
     Funktion 156  
     Konstante 53  
     Konstruktor/Methode 195  
 constinit 292  
 const\_iterator 853  
 const\_local\_iterator 885  
 const\_pointer 853  
 const\_reference 853  
 const\_reverse\_iterator 855, 900

- constraint, Vererben von 331
- Container 458
  - implizite Datentypen 853
  - Methoden 854
- Container-Adapter 868
- container\_type 868
- contains()
  - Container 874
  - string 938
- Contiguous-Iterator 893
- continue 82
- convertible\_to 534
- copy semantics 484
- copy() (Filesystem) 820
- copy() 800, 933
- copy\_backward() 800
- copy\_if() 802
- copy\_n() 803
- copy\_constructible 534
- copy\_options 821
- copy\_result (Ranges) 734
- copysign() 953
- co\_return 581
- Coroutinen 579
- cos(), cosh() 789, 953
- count() 842, 874
  - Algorithmus 750
- count\_if() 750
- cout 32, 105, 424
- co\_yield 580
- \_\_cplusplus 169
- crbegin()
  - Container 855
  - Namespace std 855, 898
  - string 933
- create\_directory() 821
- crend()
  - Container 855
  - Namespace std 855, 898
  - string 933
- critical section 560
- CRLF 621
- <cstdlib> 40, 220, 225, 954
- <cstdliblib> 130, 261
- c\_str() 933
- C-String 227

- Länge 230, 231
- <cstring> 227, 263, 273, 276, 955
- <ctime> 957
- ctype 721, 722, 920, 921
- cur 447
- curr\_symbol 925
- current\_path 818
- <cwctype> 952
- CXXFLAGS 654

## D

- DALL·E 2 635
- dangling
  - pointer 237
  - ranges 735
  - reference 482
- data race 560, 562
- data() 859, 862, 933
- Datagramm 616
- Datei
  - ASCII 258
  - binär 260
  - Ein-/Ausgabe 108, 423
  - kopieren 111, 820
  - löschen 818
  - Öffnungsarten 446
  - öffnen 111
  - Positionierung 447
  - schließen 109
  - umbenennen 822
  - Zugriffsrechte 815
- Daten
  - als Attributwerte 979
  - static-Element- 286
- Datenbankanbindung 639
- Datenkapselung 979
- Datentypen 33, 39, 175
  - abstrakte *siehe* abstrakter Datentyp
  - benutzerdefinierte 85
  - int und unsigned 60, 70
  - logische 57
  - parametrisierte 161, 293
  - polymorphe 323
  - strukturierte 88
  - zusammengesetzte 85

- Datum
    - Klasse 386
    - regulärer Ausdruck 791
  - days 386
  - daytime (Port 13) 611
  - Deadlock 579
  - dec 434, 438
  - decay, decay\_t 528
  - decimal\_point() 923, 924
  - decltype 283
  - decltype(auto) 285, 525, 834
  - declval 286
  - default (in switch-Anweisung) 71
  - = default 191
  - default constructor 182
  - default\_delete<X[]> 698
  - defaultfloat 434
  - Default-Parameter *siehe* vorgegebene P.
  - #define 137, 140
  - Definition 148, 979
    - von static-Elementdaten 287
    - von Objekten 185
  - Deklaration 32, 33, 148, 979
    - einer Funktion 117
    - Funktionszeiger 260
    - Lesen einer D. 267
    - in for-Schleifen 79
  - Deklarationsanweisung 64
  - Dekrementierung 45, 46
  - Dekrementoperator 388
  - Delegation 345
  - delegierender Konstruktor 194
  - delete 234, 236, 289, 324, 326
    - überladen 404
  - delete [ ] 238
  - = delete 191, 694
  - Deleter 699
  - <deque> 852, 867
  - deque 867
  - Dereferenzierung 218, 262
  - derived\_from 534
  - Destruktor 208, 335, 496
    - implizite Deklaration 208
    - und exit() 210
    - virtueller 324
  - detach() 553
  - Dezimalpunkt 48, 923
  - Dialog 599
  - diamond problem 338
  - difference\_type 853
  - Differenz (Menge) 780
  - difftime() 958
  - digits, digits10 810
  - distance() 897
  - Distribution 657, 979
  - div(), div\_t 954
  - divides 844
  - Division durch 0 356
  - DNS 609
  - do while 76
  - domain\_error 355
  - double 48, 224
    - nicht als Laufvariable 80
    - korrekter Vergleich 690
  - downcast 340
  - Drei (die großen Drei) 256, 496
  - Drei-Wege-Vergleich *siehe* Spaceship-Operator
  - drop() 471
  - Dubletten entfernen 746
  - Durchschnitt (Menge) 779
  - DYLD\_LIBRARY\_PATH 669
  - dynamic\_cast<>() 340
  - dynamic\_pointer\_cast 948
  - dynamischer Typ 342
  - dynamisches
    - Array 374
    - Binden 260, 315, 980
    - Datenobjekt 233
- ## E
- e, E 48
  - Editor 29
  - egrep 535
  - Ein- und Ausgabe 104
  - Einbinden von C-Funktionen 169
  - Eingabe 425
    - Datei- 108, 423
    - von Strings 106
    - benutzerdefinierter Typen 449
  - Einschränkung *siehe* constraint
  - Elementdaten, Zeiger auf 266
  - Elementfunktion 176

- als Funktionsobjekt 848
- spezielle 496
- Zeiger auf 265
- Ellipse
  - in catch-Klausel 351
  - template parameter pack 514
- else 66
- #else, #elif 137
- emplace() 857, 869, 870, 872
- emplace\_back() 558, 859, 865, 867
- emplace\_front() 864, 867
- empty() 854, 869–871, 933
  - Namespace std 855
- enable\_if 528
- end 447
- end()
  - Container 465, 854
  - Namespace std 226, 855, 898
  - string 275, 932
  - vector 226
- #endif 139
- endl 55, 438
  - oder '\n'? 107
- ends 438
- ends\_with() 938
- ENTER** 31, 105
- »enthält«-Beziehung 336
- enum 85
- Environment, env[] 243
- EOF 426
- eof() 351, 426, 427, 445, 460
- eofbit 443
- epsilon() 810
- equal() 788
- equal\_range() 776, 874
- equalsIgnoreCase() 722
- equal\_to 845
- erase() 858, 874, 936
- ereignisgesteuerte Programmierung 586
- errc 717
- errno 348
- #error 144
- error\_code 816
- Escape-Sequenz 55
- Exception 978
  - arithmetische Fehler 356

- und Destruktor 349
- Handling 349
- Hierarchie 353
- Speicherleck durch Exc. 696
- <exception> 355
- exception 353
- Exception-Sicherheit 359
- exchange() 804
- exclusive\_scan() 741
- ExecutionPolicy 814
- exists() (Filesystem) 819
- exit() 130, 955
  - und Destruktor 210
- Exklusiv-Oder (Menge) 781
- exp() 790, 953
- expected 361
- explicit 191
- explizite Instanziierung von
  - Templates 300
- Exponent 48–50
- export 170
- extension() (Filesystem) 818
- extent 527
- extern 149–151
- extern "C" 169
- extern template 299
- external linkage 154
- extract() 875

## F

- f, F (Suffix) 48
- fabs() 953
- Facette 920
- fail() 445
- failbit 443, 451
- fakultaet() 116
- [[fallthrough]] 73
- Fallunterscheidung 71
- false 57
- false\_type 522
- falsename() 923
- Fehlerbehandlung 347
  - Ein- und Ausgabe 443
- Fibonacci 512, 743
- \_\_FILE\_\_ 145
- filename() 818

## Filesystem

- canonical() 818
- copy() 820
- create\_directory() 821
- current\_path() 818
- exists() 819
- extension() 818
- filename() 818
- is\_directory() 819
- recursive\_directory\_iterator 824
- remove() 818
- remove\_all() 819
- stem() 818
- filesystem 108
- fill() 433, 735
- fill\_n() 735
- Filter, filter() 470
- final 327, 695
- find()
  - Algorithmus 767
  - assoziative Container 874
  - string 937
- find\_end() 771
- find\_first\_of() 768
- find...-Methoden (string) 937
- fixed 434, 436, 438
- Fixture 677
- flache Kopie 254
- flags() 434
- flip() 861
  - Bitset 841
- float 48
- float16\_t und weitere float-Typen 52
- floatfield 435
- floating\_point 534
- floor() 953
- flush 428
- flush() 107
- flush (Manipulator) 438
- fmod() 953
- fmtflags 434
- Fold-Expression 516
- for 78
  - Kurzform 99
- foreach(), for\_each\_n() 799
- Formalparameter 118

## Formatierung 429

- forward() 834
- Forward-Iterator 892
- <forward\_list> 852
- forwarding reference 492
- frac\_digits() 925
- Fragmentierung (Speicher) 238
- Framework 587
- free store 234
- free() 407
- frexp() 953
- friend 279
- from\_chars() 717
- front() 859, 862, 864, 867, 870, 933
- front\_inserter(), \_insert\_iterator 901
- <fstream> 108
- fstream 424, 447
- Füllzeichen 430, 433
- \_\_func\_\_ 145
- function 846
- <functional> 355, 844
- Funktion 116
  - frei oder global 153
  - mit Gedächtnis (static) 120
  - mit initializer\_list 856
  - klassenspezifische 287
  - mathematische 51, 953
  - vorgegebene Parameterwerte 127
  - Parameterübergabe
    - per Referenz 124
    - per Wert 122
    - per Zeiger 239
  - rein virtuelle 319
    - mit Definition 320
  - static 287
  - alternative Syntax 283
  - Überschreiben 314
  - virtuelle *siehe* virtuelle Funktionen
- Funktionsobjekte 396, 440
  - function 846
  - mem\_fn 848
- Funktions-Template 161
- Funktork *siehe* Funktionsobjekte
- <future> 355
- future 577

Fünf (die großen Fünf) 496

## G

Ganzzahlen 40

garbage collection 238

gcd() 203, 813

gegenseitige Abhängigkeit von  
Klassen 215

Genauigkeit 49

Generalisierung 302

generate(), generate\_n() 736

generische Programmierung 458

GET (http) 621

get() 105, 258, 425, 426

getenv() 955

getline() für Strings 107, 939

getline(char\*,...) 426

getloc() 925

get\_money() (Manipulator) 438

ggT 75

std::gcd() 813

schnell 203

Gleichheitsoperator 372

bei Vererbung 420

Gleichverteilung 795

Gleitkommazahl 53

Syntax 48

global 62

Funktion 153

Namensraum 167

Variable 149, 150

glvalue 479

gmtime() 958

GNU Autotools 671

good() 445

goodbit 443

goto 84

grafische Benutzungsschnittstelle 585

greater, greater\_equal 845

greedy (regex-Auswertung) 538

Grenzwerte von Zahltypen 809

grouping() 923, 924

Groß- und Kleinschreibung 31

größter gemeinsamer Teiler *siehe* ggT

guard (Threads) 561

Gültigkeitsbereich 130

Block 61

Datei 154

Funktion 119

Klassen 178

und new 237

## H

hängender Zeiger 237

hardware\_concurrency() 549

has\_facet() 914

hash() 921

hasher 885

Hash-Funktion 883, 884

has\_infinity 810

\_\_has\_include 140

has\_sort\_function 522

has\_value() (optional) 362

»hat«-Beziehung 712

Header 33, 167

Datei 145

Inhalt 149

Http 621

der Standardbibliothek 828

Heap 782

hex 434, 438

Hexadezimalzahl 42

hexfloat 434

Host Byte Order 618

hours 548

hängende Referenz 482

## I

-I Compileroption 137

iconv 919

IDE 36

identity 734

Identität von Objekten 175, 980

IEC 60559, IEEE 754 49

if 66

if consteval 158

if constexpr 142, 516

#if, #ifdef, #ifndef 137

ifstream 108, 424

ignore() 426

imag() 789, 790

imbue() 721, 912

- Implementation 145
  - sdatei, Inhalt 149
  - svererbung 343
- implizite Deklaration
  - Destruktor 208
  - Konstruktor 182
  - Zuweisungsoperator 379, 419
- import 170
- in 446
- #include 32, 137
- Include-Guard 138
- includes() 778
- inclusive\_scan() 740
- Indexoperator 90, 222, 225, 250, 377
  - mehrdimensionaler 417
- index\_sequence 842
- infinity() 810
- Initialisierung
  - array 862
  - direkte I. der Attribute 184
  - C-Array 224, 247
  - einfacher Datentypen 43
  - mit Element-Initialisierungsliste 183
  - Konstante in Objekten 183, 288
  - globaler Konstanten 149, 151
    - Reihenfolge 292
  - mit {}-Liste 192, 856, 898
  - mit konstruktor-interner Liste 288
  - und move() 490
  - von Objekten 182
  - von Referenzen 970
  - Reihenfolge der Initialisierung
    - von Attributen 183
    - von Funktionsargumenten 136
  - in for-Schleife 78
  - von static-Elementdaten 287
  - und virtuelle Basisklassen 338
  - und Vererbung 306
  - und Zuweisung 44, 187
- initializer\_list 193, 377, 811, 856
  - für zweidimensionales Array 856
  - und for-Schleifen 194
- Inklusionsmodell 299
- Inkrementierung 45, 46
- Inkrementoperator 385
- inline
  - Elementfunktion 181
  - Funktion 155
  - Konstante 151, 290
  - Variable 155, 290
- inner\_product() 739
- innere Klasse 466, 641
- inplace\_merge() 766
- Input-Iterator 892
- insert() 858, 874, 882, 935
- inserter() 902
- insert\_iterator 902
- insert\_or\_assign 878, 887
- Instanz 174, 980
- Instanziierung von Templates 295
  - explizite 300
  - ökonomische (bei vielen Dateien) 298
- int 32, 40
- int-Parameter in Templates 296
- intX\_t, int\_fastX\_t, int\_leastX\_t
  - (X = 8, 16, 32, 64), intmax\_t 47
- integer\_sequence 842
- integral 534
- integral\_constant 522
- integral promotion 60
- Interface (UML) 706
- internal 434, 438
- internal linkage 154
- Internet-Anbindung 607
- Intervall (und Notation) 980
- invalid\_argument 355
- <iomanip> 438, 439
- ios 424, 443, 445
  - failure 445
  - Flags zur Dateipositionierung 447
  - Methoden 434, 445
- <ios> 438
- ios\_base 424
  - binary 111
  - Fehlerstatusbits 443
  - Flags 434, 435
  - Manipulatoren 438
- iostate 443
- <iostream> 33, 424, 425, 428, 438
- iota, iota\_view 737



IPv4, IPv6 609  
IPv4-Adresse (regulärer Ausdruck) 793  
is() 921  
is\_abstract 527  
isalnum(), isalpha() 920, 952  
is\_arithmetic 525, 526, 531  
is\_array 526  
is\_base\_of 527  
isblank() 952  
is\_bounded 810  
is\_class 521, 526  
iscntrl() 920, 952  
is\_const 527  
is\_convertible 527  
isdigit() 135, 920, 952  
is\_directory() (Filesystem) 819  
is\_enum 526  
is\_exact 810  
is\_final 527  
is\_floating\_point 526  
is\_function 526  
is\_fundamental 527  
isgraph() 920, 952  
is\_heap(), is\_heap\_until() 786  
is\_iec559, is\_integer 810  
is\_integral 526  
islower() 920, 952  
is\_lvalue\_reference 526  
is\_modulo 810  
isnan() 400  
is\_null\_pointer 526  
ISO 10646 916  
ISO 8859-1, ISO 8859-15 915  
is\_partitioned() 758  
is\_permutation() 754  
is\_pointer 526  
is\_polymorphic 527  
isprint() 920, 952  
ispunct() 920  
is\_reference 526  
is\_rvalue\_reference 526  
is\_same 527  
is\_signed 527, 810  
is\_sorted(), is\_sorted\_until() 760  
isspace() 920, 952  
*Ist-ein*-Beziehung 302, 311, 330

istream 424, 425  
Istream-Iterator 902  
istream::seekg(), tellg() 447  
istream::ws 438  
istreamstring 424, 448  
is\_unsigned 527  
isupper() 920, 952  
is\_void 526  
isxdigit() 920, 952  
iter\_swap() 803  
Iterator 226, 275, 459, 463, 891  
    Adapter 899  
    Bidirectional 893  
    Contiguous 893  
    Forward 892  
    Input 892  
    Insert 900  
    Output 892  
    Random Access 893  
    Reverse 899  
    Stream 902  
    Tag 893  
    Zustand 464  
<iterator> 891  
iterator 853  
iterator\_category 892

## J

Jahr 957  
join() 552, 553  
jthread (Klasse) 554

## K

Kardinalität 709  
Kategorie (locale) 920  
key\_equal 885  
key\_type 873, 880  
key\_comp(), key\_compare 875  
KI 635  
Klammerregeln 58  
Klasse 174, 177, 981  
    abgeleitete *siehe* abgeleitete  
        Klasse  
    abstrakte 319  
    Basis- *siehe* Basisklasse  
    Deklaration 179

innere 466, 641  
 konkrete 320  
 Ober- *siehe* Oberklasse  
 für einen Ort 178  
 Unter- *siehe* Unterklasse  
 für rationale Zahlen 199  
 Klassenname (typeid) 343  
 klassenspezifische  
   Daten und Funktionen 286  
   Konstante 290  
 Klassen-Template 293  
 Klassifikation 302, 981  
 Kleinschreibung 31  
 kleinstes gemeinsames Vielfaches  
   *siehe* lcm()  
 Kollisionsbehandlung 883  
 Kommandointerpreter 653  
 Kommandozeilenparameter 243  
 Kommaoperator 59, 81, 519  
 Kommentar 30  
 komplexe Zahlen 789  
 Komplexität 986  
 Komposition 712  
 konkrete Klasse 320  
 Konsole  
   auf UTF-8 einstellen 915  
 Konstante 52  
   globale 149, 151  
   klassenspezifische 290  
 konstante Objekte 180  
 Konstruktor 179, 182  
   allgemeiner 184  
   implizite Deklaration 182  
   delegierender 194  
   erben 312  
   Kopier- *siehe* Kopierkonstruktor  
   vorgegebene Parameterwerte 185  
   Typumwandlungs- *siehe*  
     Typumwandlungskonstruktor  
 Kontrollstrukturen 64  
 Konvertieren von Datentypen *siehe*  
   Typumwandlung  
 Kopie, flache/tiefe 254  
 Kopieren  
   von Dateien 111  
   von Zeichenketten 232

Kopierkonstruktor 187, 377, 496  
   Auslassung durch Compiler 189  
 Kreuzreferenzliste 731  
 kritischer Bereich 560  
 Kurzform-Operatoren 45

## L

l, L, ll, LL (Suffix) 42  
 Lambda-Funktionen 501  
 LANG 912  
 late binding 260  
 Laufvariable 78, 79  
 Laufzeit 218  
   und Funktionszeiger 260  
   und new 234  
   und Polymorphie 315  
   Typinformation 342  
 lcm() 813  
 LD\_LIBRARY\_PATH 669  
 ldd 669  
 ldexp() 953  
 ldiv(), ldiv\_t 954  
 left 434, 438  
 length() (C-String, constexpr) 228, 281,  
   529  
 length() (string) 933  
 length\_error 355  
 less, less\_equal 845  
 lexicographical\_compare() 755  
 lexicographical\_compare\_three\_way() 756  
 lexikografischer Vergleich 401, 981  
 <limits> 41, 48, 809  
 \_\_LINE\_\_ 145  
 Linken 150, 167  
   dynamisches 668, 981  
   internes, externes 154  
   statisches 666, 981  
 Linker 35  
 linksassoziativ 58, 246  
 list 864  
 <list> 832, 852  
 Liste  
   Initialisierungs- 183, 288  
   Initialisierungs- (bei C-Arrays)  
     247  
 Liste (Klasse) 465

Literal 228, 981  
    benutzerdefiniert 412  
    String 411  
    Zahl- 53  
    Zeichen- 54, 916  
load\_factor() 885  
local\_iterator 885  
<locale> 911  
localtime() 958  
log(), log10() 789, 953  
logic\_error 354, 355  
logical\_and, \_not, \_or 845  
logischer Datentyp 57  
logische Negation 57  
lokal (Block) 61  
lokale Objekte 221  
long 40  
long double 48  
lower\_bound() 775, 875  
lvalue 480  
L-Wert 480  
Länge eines Vektors 739

## M

magic number 982  
main() 30, 32, 130  
MAKE 664  
make 651  
    automatische Ermittlung von  
        Abhängigkeiten 657  
    parallelisieren 665  
    rekursiv 664  
    Variable 654  
Makefile 146, 652  
make\_from\_tuple() 838  
make\_heap() 784  
make\_index\_sequence 842  
make\_integer\_sequence 842  
make\_pair() 836  
make\_shared 697, 947  
make\_tuple() 837  
make\_unique 270, 697, 945  
Makro 140  
malloc() 407  
Manipulatoren 437  
Mantisse 49  
    <map> 852, 876  
mapped\_type 878  
match\_results 542  
mathematische Funktionen 953  
mathematischer Ausdruck 51  
Matrix  
    C-Array 246  
    Klasse 251  
    operator[] 415  
max() 810  
max(initializer\_list<T>) 811  
max\_bucket\_count() 885  
max\_element() 743  
max\_exponent, max\_exponent10 810  
max\_load\_factor() 886  
max\_size() 854  
[[maybe\_unused]] 965  
mehrdimensionaler Indexoperator 417  
mehrdimensionales C-Array 246  
Mehrfachvererbung 304, 333, 335  
MeinString (Klasse) 273  
mem\_fn() 848  
member function *siehe* Element-  
    funktion  
memcpy() 376  
<memory> 355, 832  
memory leak 237, 696  
Mengenoperationen auf sortierten  
    Strukturen 777  
merge() 765, 865, 874  
mergesort() 766  
messages 920, 927  
Metaprogrammierung 509  
Methode 174, 176, 982  
    Regeln zur Konstruktion  
        von Prototypen 688  
microseconds 549  
midpoint() 813  
milliseconds 549  
MIME 982  
min() 810  
min(initializer\_list<T>) 811  
min\_element() 743  
min\_exponent, min\_exponent10 810  
minmax(...) 811  
minmax\_element() 743

minus 844  
 Minute 957  
 minutes 548  
 mischen 765  
 mismatch() 786  
 mkdir 659  
 mktime() 958  
 modf() 953  
 modulare Programmgestaltung 145  
 Module 169  
 Modulo 45  
 modulus 844  
 monadische Operationen 364  
 Monat 957  
 monetary 920, 924  
 money\_get 925  
 moneypunct 924  
 money\_put 926  
 Monitor-Konzept 570  
 move() 489  
     Initialisierung 490  
 move semantics 484  
 move(), move\_backward()  
     Container-Bereich 834  
 move\_constructible 534  
 moving constructor 487  
 mt19937 746, 794  
 multimap 880  
 multiplies 844  
 Multiplikationsoperator 383  
 Multiplizität (UML) 709  
 multiset 882  
 mutable  
     Attribut 180  
     Lambda-Funktion 506  
 mutex 560

## N

'\n' 33  
     oder endl? 107  
     und regex\_replace 544  
 Nachbedingung 136, 982  
 Nachkommastellen 48  
     precision 436  
 Name 39  
     einer Klasse (typeid) 343

name() 343, 914  
 Named Return Value Optimization (NRVO)  
     189  
 Namenskonflikte bei Mehrfach-  
     vererbung 335  
 Namespace 63, 151  
     anonym 154  
     in Header-Dateien 154  
     Verzeichnisstruktur 661  
 namespace 151  
 namespace std 63  
 NaN (not a number) 400  
 nanoseconds 548  
 narrow() 922  
 nationale Sprachumgebung 912  
 NDEBUG 143  
 negate 844  
 Negation  
     bitweise 45, 47  
     logische 57, 58  
 negative\_sign() 925  
 neg\_format() 925  
 Network Byte Order 618  
 Netzwerkprogrammierung 607  
 neue Zeile 55, 66  
 new 233, 236, 289  
     Fehlerbehandlung 358  
     Placement-Form 949  
     überladen 404  
 <new> 355, 949  
 new\_handler 358  
 next() 897  
 next\_permutation() 753  
 noboolalpha 438  
 Nodehandle 875  
 [[nodiscard]] 118, 202  
 noexcept 352  
 none() (Bitset) 842  
 none\_of 751  
 norm() 789, 790  
 Normalverteilung 797  
 noshowbase, -point, -pos 438  
 noskipws 438  
 not\_equal\_to 845  
 nothrow 359  
 notify\_one(), -\_all() 566, 703

nounitbuf, nouppercase 438  
now 386, 549  
npos 932  
nth\_element() 764  
NTP – Network Time Protocol 618  
NULL 220, 954  
nullopt 361  
nullptr 220  
Null-Zeiger und new 359  
<numbers> 196  
numeric 920, 923  
numeric\_limits 48, 809  
numerische Auslöschung 50  
numerische Umwandlung 715, 718, 940  
num\_get, num\_put 923  
NummeriertesObjekt (Klasse) 287  
numpunct 923

## O

O-Notation 986  
Oberklasse 302, 314, 982  
    Subobjekt einer 306  
    Subtyp einer 310  
    Zugriffsrechte vererben 308  
Oberklassenkonstruktor 303, 306  
object slicing 311, 319  
Objekt 28, 175, 179, 983  
    -code 35  
    dynamisches 233  
    als Funktions- 396  
    -hierarchie 336  
    Identität *siehe* Identität von  
        Objekten  
    Initialisierung 182  
    konstantes 180  
    -orientierung 173  
    Übergabe per Wert 188  
    verkettete Objekte 236  
    verwitwetes 237  
    vollständiges 338, 986  
oct 434, 438  
ODER  
    bitweises 45  
    logisches 58  
Öffnungsarten für Streams 446  
offsetof 954  
ofstream 108, 424  
Oktalzahl 42  
omanip 439  
one definition rule 149  
open() 108  
Open Source 983  
Operator  
    arithmetischer 45  
    binärer 369  
    Bit- 45  
    für char 57  
    als Funktion 368  
    Kurzform 46  
    für Literale 410  
    für logische Datentypen 57  
    Präzedenz 58, 963  
    relationale 45, 57  
    Syntax 368  
    Typumwandlungs- 389  
    unärer 369  
    für ganze Zahlen 45  
operator delete() 404  
operator new() 404  
operator()() 396  
operator\*() 384, 390, 463  
operator\*=( ) 383  
operator++() 385, 387, 463  
operator++(int) 692  
operator+=( ) 371  
operator->() 390  
operator<=>() 398  
operator<<() 371, 427  
operator=() 380  
operator==( ) 463, 842  
    bei Vererbung 420  
operator==( ) 372  
operator>>() 425  
operator[]() 416, 859, 862, 867, 893  
operator!=( ) 463  
Optimierung  
    Compileroption 963  
    Vermeiden temporärer Objekte 189  
optional 361  
or\_else() 364  
Ort (Klasse) 178  
ostream 371, 424, 427

ostream::endl, ends, flush() 438  
 Ostream-Iterator 902  
 ostream::seekp(), tellp() 447  
 ostreamstringstream 424, 448  
 osyncstream 568  
 out 446  
 out\_of\_range 355  
 Output-Iterator 892  
 overflow 43, 50  
 overflow\_error 355  
 override 317, 694

## P

packaged\_task 577  
 pair 835  
 Parameter  
   -expansion 515  
   einer Funktion 117  
   Pack 514  
   -übergabe  
     per Referenz 124  
     per Wert 122  
     per Zeiger 239  
 parametrisierte Datentypen 161, 293  
 »part-of«-Beziehung 712  
 partial\_ordering 400  
 partial\_sort(), partial\_sort\_copy 762  
 partial\_sum() 740  
 partielle Spezialisierung von  
   Templates 892  
 partition() 757  
 partition\_copy(), partition\_point() 758  
 path 108, 815  
 patsubst 657  
 peek() 427  
 perfect forwarding 834  
 Performance 475  
 Permutationen 752  
 Pfad 108  
 Pfeiloperator 221  
 PHONY 654  
 Pipe (ranges) 470  
 Placement new/delete 949  
 plus 844  
 pointer 853  
 Pointer, smarte 390  
 polar() 790  
 polymorpher Typ 323, 342  
 Polymorphismus 315, 983  
 pop() 869, 870, 872  
 pop\_back() 860, 865, 868  
 pop\_front() 865, 867  
 pop\_heap() 783  
 portabel (Zeichensatz) 918  
 pos\_format() 925  
 Positionierung innerhalb einer Datei 447  
 positive\_sign() 925  
 POSIX 717, 912  
 POST (http) 626  
 postcondition *siehe* Nachbedingung  
 Postfix-Operator 385  
 pos\_type 447  
 pow() 790  
 precision() 435, 436  
 precondition *siehe* Vorbedingung  
 prev() 897  
 prev\_permutation() 752  
 PRINT (Makro) 142  
 printf() 513  
 Priority-Queue 871  
 private 178, 307  
 private Vererbung 343  
 Programm  
   ausführbares 35  
   Strukturierung 115  
 Programmierrichtlinien 113, 696  
 Projection (Ranges) 734  
 proj() 790  
 Projekt 36, 146  
 promise 577  
 protected 307  
 protected-Vererbung 345  
 Prototyp  
   Funktions- 116  
   einer Methode 178  
   Regeln zur Konstruktion 688  
 prvalue 480  
 Prädikat  
   Algorithmus mit P. 906  
   binäres 748, 788, 906  
   unäres 750  
 Präfix-Operator 385

Präprozessor 33, 136, 657  
Präzedenz von Operatoren 58, 963  
ptrdiff\_t 40, 225, 855, 954  
public 178, 307  
Pufferung (Ein-/Ausgabe) 105, 428  
push() 869, 870, 872  
push\_back() 860, 865, 867  
    vector 94  
push\_front() 865, 867  
push\_heap() 784  
put() 108, 258, 428  
putback() 426  
put\_money() (Manipulator) 438

## Q

qsort() 261, 955  
Qt 587  
QThread 602  
Quantifizierer 538  
Quellcode 35  
Queue 870  
<queue> 832, 852, 870  
quoted() 438

## R

race condition 560, 578  
radix 810  
RAII 561, 677, 697, 983  
<random> 746  
random access 464  
Random-Access-Iterator 893  
random\_access\_iterator 534  
random\_device 795  
range based for 99  
range\_error 355  
Ranges (Bereiche), <ranges> 469  
rank 527  
<ratio> 848  
Rationale Zahl  
    Klasse 199  
    Template std::ratio 848  
rbegin()  
    Container 855, 899  
    Namespace std 855, 898  
    string 933  
rdstate() 445

read() 257  
real() 789, 790  
Rechengenauigkeit 49, 80  
rechtsassoziativ 58, 246  
recursive\_directory\_iterator 824  
reelle Zahlen 48  
ref(), reference\_wrapper 557, 850  
reference  
    bitset 840  
    Container 853  
    vector<bool> 860  
reference collapsing rules 491  
reference counting 946  
Referenz 124, 133  
    auf Basisklasse 318  
    auf const 126, 187  
    auf istream 425, 449  
    auf Oberklasse 310, 318  
    auf ostream 372, 427  
    Parameterübergabe per 124  
    auf R-Wert 484  
    Rückgabe per 378  
    -semantik 255, 475  
    weiterleitende 492  
Referenz-Qualifizierer 482  
Regel der 0/3/5 *siehe* Rule of 0/3/5  
<regex> 355, 543  
regex\_iterator 542  
regex\_match() 543  
regex\_replace() 544, 726  
regex\_search() 544, 724  
reguläre Ausdrücke 535  
Reihenfolge  
    Auswertungs- 51, 59  
    der Initialisierung  
        von Attributen 183  
        von Funktionsargumenten 136  
    umdrehen 749  
rein virtuelle Funktion 319  
    mit Definition 320  
reinterpret\_cast<>() 225, 257  
Rekursion 123  
    Template-Metaprogrammierung 510,  
        514  
rekursiver Abstieg 130, 131  
rekursiver Make-Aufruf 664

- relationale Operatoren 45, 57
- remove()
  - Algorithmus 807
  - Datei/Verzeichnis 818
  - Liste 865
- remove\_all() Dateien/Verzeichnisse 819
- remove\_const 528
- remove\_if() 807, 865
- remove\_reference 527, 834
- rename() Datei/Verzeichnis 822
- rend()
  - Container 855, 899
  - Namespace std 855, 898
  - string 933
- replace() 806, 936
- replace\_copy(), \_if(), \_copy\_if() 806
- requires 531
- reserve() 860, 934
- reset() (Bitset) 841
- resetiosflags() 438
- resize() 860, 865, 868, 934
- resume() 580
- return 33, 189
- Return Value Optimization (RVO) 189
- reverse() (list) 865
- reverse(), reverse\_copy() 749
- reverse (Ranges) 471
- reverse\_iterator 855
- Reverse-Iterator 899
- reversible Container 855
- right 434, 438
- »rohes« Stringliteral 411
- rotate(), rotate\_copy() 744
- round\_error(), round\_style() 810
- RTTI 342
- Rückgabotyp auto 159, 182, 318
- Rule of zero/three/five 496
- runtime\_error 355
- rvalue 480
- R-Wert, Referenz auf R-Wert 480
- S**
- safe\_iterator\_t 735
- safe\_subrange\_t 735
- same\_as 533
- sample() 749
- scan\_is(), scan\_not() 922
- Schleifen 74
  - und Container 98
  - do while 76
  - for 78
  - und C-Strings 230
  - Tabellensuche 93
  - terminierung 76
  - while 74
- Schlüsselwörter 961
- Schnittmenge 779
- Schnittstelle 145, 984
  - einer Funktion 121
  - Regeln zur Konstruktion 688
- scientific 434, 436, 438
- scope 61
- scoped locking 561
- scoped\_lock 561
- search() 770, 771
- search\_n() 773
- seconds 548
- seekg(), seekp() 447
- Seiteneffekt 67, 117, 136, 231, 233
  - im Makro 143
- Seitenvorschub 55
- Sekunde 957
- Selbstzuweisung 381, 498
- Selektion 66
- sentinel 93, 225
- Sequenz
  - konstruktor 192, 193, 377
  - methoden (Container) 858
- Server-Client
  - Beziehung 179
  - und callback 262
- <set> 852, 880
- set() (Bitset) 841
- setbase() 438
- set\_difference() 780
- setf() 434, 435
- setfill() 438
- set\_intersection() 779
- setiosflags() 438
- setprecision() 438
- setstate() 445
- set\_symmetric\_difference() 781



- set\_terminate() 356
- set\_union() 779
- setw() 438
- SFINAE 523, 984
- shared\_ptr 395, 697, 946
  - für C-Arrays 698
- short 40
- showbase, showpoint, showpos 434, 438
- showContainer() 524
- shrink\_to\_fit() 860, 868, 934
- shuffle() 746
- Sichtbarkeit 61
  - sbereich (namespace) 151
- sign() 953
- Signal 588, 590
- Signalton 55
- Signatur 128, 304, 316, 318, 984
- signed char 54
- signed\_integral 534
- sin(), sinh() 789, 953
- single entry/ single exit 82
- size()
  - Container 854
  - Namespace std 90, 223, 855
  - string 96
  - vector 90
- size\_t 40, 954
- size\_type 853
- sizeof 224
- sizeof 40
- sizeof... (variadische Templates) 514
- Skalarprodukt 739
- skipws 434, 438
- sleep\_for(), sleep\_until() 548, 553
- Slot 588, 590
- Small String Optimization 689, 984
- Smart Pointer 390
  - und Exceptions 696
- Socket 611
- Sommerzeit 957
- Sonderzeichen 56
- sortable 533
- sort\_heap() 785
- Sortieren
  - mit qsort() 262
  - mit sort() 760
  - stabiles 760
    - durch Verschmelzen 766
- source\_location 144
- Spaceship-Operator 398, 854
- <span> 852
- span 242, 889
- Speicher
  - klasse 150
  - leck 237, 696
  - platzfreigabe 220
- Spezialisierung
  - von Klassen 302
  - von Templates 163
- spezielle Elementfunktionen 496
- splice() 865
- split() 714
- Sprachumgebung 912
- SQL 639
- sqrt() 51, 790, 953
- ssize() 99, 855
  - Namespace std 223
- <sstream> 448
- stable\_partition() 757
- stable\_sort() 761
- Stack 61
  - Klasse 293
- <stack> 832, 852, 868
- stack unwinding 349
- Standard
  - bibliothek
    - C 167, 951
    - C++ 827
  - header 168, 831
  - klassen 831
  - Typumwandlung 59, 264
    - Zeiger 264
- Standard-Ein-/Ausgabe 105
- starts\_with() 938
- static 150
  - Attribute und Methoden 286
  - in Funktion 120
  - Initialisierungsreihenfolge 292
- static und -Bstatic (Makefile) 667
- static\_assert 143
- static\_cast<>() 56, 340
- static operator() 398

- statisches Binden 315, 984
- Statusabfrage einer Datei 445
- std 63, 153
- <stdexcept> 355
- <stdfloat> 52
- stdio 434
- Stelligkeit (Template) 512
- stem() (Filesystem) 818
- Stichprobe 749
- STL 457
- stod() 716
- stoi() 716
  - und verwandte numerische Konversionsfunktionen 940
- stop\_requested(), stop\_token() 555
- strcat(), strchr(), strcmp() 956
- strncat(), strncmp() 957
- strpbrk(), strrchr(), strstr() 956
- strcpy() 956
- strcspn() 956
- Stream
  - Iterator 902
  - Öffnungsarten 446
- stream 423
- streamsize 425
- strerror() 717, 956
- Streuspeicherung 882
- strftime() 958
- String 95
  - in Zahl umwandeln 715
  - Klasse MeinString 273
  - Literal 411, 916
- string 95, 931
  - append() 934
  - assign() 935
  - at() 96, 933
  - back() 933
  - begin() 932
  - capacity() 934
  - cbegin(), cend(), crbegin(), crend() 933
  - clear() 936
  - compare() 938
  - copy() 933
  - c\_str() 933
  - data() 933
  - empty() 933
  - end() 932
  - erase() 936
  - find() 937
  - find\_...-Methoden 937
  - front() 933
  - insert() 935
  - length() 96, 933
  - max\_size() 933
  - operator+=() 934
  - rbegin(), rend() 933
  - replace() 936
  - reserve() 934
  - resize() 934
  - shrink\_to\_fit() 934
  - size() 96, 933
  - substr() 938
  - swap() 933
- <string> 832
- stringstream 718
- string\_view 282, 940
- <string\_view> 832
- strlen() 228, 956
- strncpy() 233, 957
- strong\_ordering 400
- strtod(), strtol(), strtoul() 955
- strtok() 956
- struct 88, 308
- strukturierte Bindung 102
- Stunde 957
- Subobjekt 303, 310, 334
  - in virtuellen Basisklassen 337
  - verschiedene 336
- subrange 472
- Substitutionsprinzip 330
- substr() 938
- Subtyp 304, 310, 330, 985
- Suffix 42, 48, 97
- suspend\_always 582
- swap() 380, 854, 933
  - Algorithmus 803
- swap-Trick 256
- swap\_ranges() 803
- switch 71
- symmetrische Differenz (Menge) 781
- Synchronisation 559

Syntax 53  
Syntaxdiagramm  
  ?: Bedingungsoperator 70  
  do while-Schleife 76  
  enum-Deklaration 85  
  for-Schleife 78  
  Funktions-  
    aufruf 118  
    definition 117  
    prototyp 117  
    Template 161  
  if-Anweisung 66  
  mathematischer Ausdruck 131  
  operator-Deklaration 368  
  struct-Definition 88  
  switch-Anweisung 71  
  Typumwandlungsoperator 389  
  while-Schleife 74  
system() 955  
system\_clock 386, 548  
system\_error 355, 445  
Szenario 211

## T

Tabellensuche 93  
Tabulator 55  
Tag (des Monats) 957  
Tag Dispatching (und Alternative) 894  
tan(), tanh() 789, 953  
target (make) 652  
Taschenrechnersimulation 130  
Tastaturabfrage 105  
TCP 608  
»Teil-Ganzes«-Beziehung 712  
tellg(), tellp() 447  
Template  
  Alias 268  
  für Funktionen 161  
  Instanziierung von T. 295  
    explizite 300  
    ökonomische (bei vielen  
      Dateien) 298  
  int-Parameter 296  
  für Klassen 293  
  -Metaprogrammierung 510  
  Method (Design-Muster) 328, 420

Spezialisierung 163  
  partielle 892  
  als template-Parameter 298  
  variable Parameterzahl 512  
temporäres Objekt (Vermeidung) 189  
terminate() 355  
terminate\_handler 356  
test() (Bitset) 842  
Test Driven Development 684  
Test-Suite 675  
Textersetzung 140  
this 244  
this->, \*this bei Zugriff auf  
  Oberklassenelement 384  
this\_thread 552  
thousands\_sep() 923, 924  
Thread 547  
thread (Klasse) 550  
ThreadGroup 558  
Thread-Sicherheit 578  
throw 350  
tiefeKopie 254  
time 920  
time() 958  
time\_t 957  
tm 957  
to\_array() 863  
tolower() 721, 920, 922, 952  
top() 869, 872  
to\_string()  
  bitset 842  
  Zahlkonvertierung 940  
to\_ulong() (Bitset) 841  
toupper() 721, 917, 920, 922, 952  
trailing return type 283  
traits 891  
traits::eof() 426  
transform()  
  Algorithmus 804  
  collate 921  
  monadische Funktion 364  
  Ranges 471  
transform\_exclusive\_scan() 741  
transform\_inclusive\_scan() 741  
tree (Programm) 660, 824

Trennung von Schnittstellen und  
     Implementation 146  
 true 57  
 truenam() 923  
 true\_type 522  
 trunc 446  
 try 350  
 try\_emplace() 887  
 Tupel, <tuple> 837  
 Typ 985  
     polymorpher bzw. dynamischer  
         Typ 342  
 type cast *siehe* Typumwandlung  
 typedef 267  
 typeid() 342  
 type\_info 342  
 <typeinfo> 355  
 typename (bei Template-Parametern) 161  
 Type Traits 521  
 Typinformation 324  
     zur Laufzeit 342  
 Typumwandlung  
     cast 56, 220, 262  
     durch Compiler 201  
     dynamic\_cast<>() 340  
     mit explicit 191  
     implizite 60, 191  
     mit Informationsverlust 129  
     -skonstruktor 190, 201  
     -soperator 389  
         ios 445  
     Standard- 59  
         Zeiger 264  
     static\_cast<>() 340

## U

u, U (Suffix) 42  
 u8 916  
 u8string, u16string, u32string 411  
 UCS 916  
 UDP 608, 616  
 Überladen  
     von Funktionen 128  
     von Operatoren *siehe* operator  
 Überlauf 43, 50  
 Überschreiben

    von Funktionen 314  
 Übersetzung 146  
     -seinheit 148  
 uintX\_t, uint\_fastX\_t, uint\_leastX\_t  
     (X = 8, 16, 32, 64), uintmax\_t 47  
 Umgebungsvariable 243  
 UML 705  
 Umleitung der Ausgabe auf Strings 448  
 UND  
     bitweises 45, 47  
     logisches 58  
 #undef 139  
 undefined behaviour 238  
 underflow 51  
 underflow\_error 355  
 unsigned\_integral 534  
 unexpected 363  
 Unicode 423, 915  
 uniform\_int\_distribution 795  
 uniform\_real\_distribution 796  
 union 104  
 unique() Sequenzen 865  
 unique(), \_copy() Algorithmen 746  
 unique\_lock 561  
 unique\_ptr 270, 395, 697, 943  
     für C-Arrays 699  
 Unit-Test 673  
 unitbuf 428, 434, 438  
 unordered\_map 886  
 unordered\_multimap 888  
 unordered\_multiset 889  
 unordered\_set 888  
 unsigned 40  
 unsigned char 54  
 Unterklasse 302, 985  
 unärer Operator 369  
 unäres Prädikat 750  
 upper\_bound() 775, 875  
 uppercase 434, 438  
 URI, URL 608, 726  
 URL-Codierung 621  
 use case 211  
 use\_facet() 721, 722, 914  
 using  
     enum 87  
     Deklaration 309, 344

- Namespace
  - Deklaration 152
  - Direktive 152
  - statt typedef 267
  - Template-Alias 268, 365
- UTC 958
- UTF 411, 985
- UTF-8 55, 915
- <utility> 286, 489, 803, 834, 837

## V

- valarray 251
- valgrind 409
- value(), value\_or() (optional) 362
- value\_comp(), value\_compare 875
- value\_type 853, 873, 878
- Variable 33
  - automatische 150
  - globale 149, 150
  - make 654
  - Name 39
- variadic templates 512
- variant 843
- vector 859
  - at() 91
  - push\_back() 94
  - size() 90
- <vector> 832, 852, 859
- vector<bool> 860
- Vektor 89
  - Klasse 374
  - Länge (geom.) 739
- Verbundanweisung 65
- verdecken (Methode) 304, 309
- Vereinigung (Menge) 779
- Vererbung 301, 985
  - der abstrakt-Eigenschaft 320
  - von constraints 331
  - der Implementierung 344
  - Mehrfach- 333
  - private 343
  - protected 345
  - von Zugriffsrechten 308
  - und Zuweisungsoperator 419
- Vergleich
  - von double-Werten 690
  - bei Vererbung 420
- Verschiebung (*move*) 484
- verschmelzen (*merge*) 765
- Vertrag 331, 986
- verwitwetes Objekt 237
- Verzeichnis
  - anlegen 821
  - anzeigen 823
  - kopieren 820
  - löschen 818
  - umbenennen 822
- Verzeichnisbaum
  - anzeigen 824
  - make 661
- Verzweigung 66
- View
  - Definition 281
  - ranges 469
- virtual 316, 318, 326
- virtuelle Basisklasse 336
- virtuelle Funktionen 316, 318
  - private 328
  - rein- 319
- virtueller Destruktor 324
- void 220
  - als Funktionstyp 117
- void\* 262
  - Typumwandlung nach 220
- volatile 986
- vollständiges Objekt 338, 986
- Vorbedingung 136, 986
- vorgegebene Parameterwerte
  - in Funktionen 127
  - in Konstruktoren 185
- Vorkommastellen 48
- Vorrangregeln 58, 963
- Vorwärtsdekларation 215

## W

- Wahrheitswert 57
  - zufällig erzeugen 798
- wait() 565
  - mit Lambda-Funktion 570
- #warning 144
- Warteschlange 870
- Wartung 113

wchar\_t 54, 916, 954  
 weak\_ordering 401  
 weak\_ptr 948  
 Webserver 627  
 Weite der Ausgabe 431, 433  
 weiterleitende Referenz 492  
 Wert  
     eines Attributs 978  
     Parameterübergabe per 122  
 Wertebereich ganzer Zahlen 42  
 Wertsemantik 255, 459, 475  
     Performanceproblem 477  
 what() 354  
 while 74, 230, 232  
 whitespace *siehe* Zwischenraum-  
     zeichen  
 wide character 54  
 widen() 922  
 Widget 590  
 width() 433  
 Wiederverwendung  
     durch Delegation 345  
 wildcard 657  
 Winterzeit 957  
 Wochentag 957  
 wofstream 919  
 Wrapperklasse für Iterator 899  
 write() 257, 428  
 ws 438  
 wstring 411, 919, 931  
 Wächter (Tabellenende) 93, 225

## X

XOR, bitweises 45  
 xvalue 480

## Y

year\_month\_day 385  
 yield() 552

## Z

z, Z (Suffix) 42  
 Zahl in String umwandeln 718  
 Zahlenbereich 41, 49  
 Zeichen 54  
 Zeichenkette 33, *siehe auch* String  
     C-String 227

Kopieren einer 231  
 Zeichenklasse (Regex) 538  
 Zeichenliteral 916  
 Zeichensatz 54, 915  
 Zeiger 217, 234  
     Arithmetik 224  
     vs. Array 223  
     auf Basisklasse 318, 324  
     Darstellung  
         von [ ] 225  
         von [ ][ ] 250  
     auf Elementdaten 266  
     auf Elementfunktionen 265  
     auf Funktionen 260  
     hängender 237  
     intelligente *siehe* Smart Pointer  
     Null-Zeiger 219  
     auf Oberklasse 310, 318  
     auf Objekt (Mehrfachvererbung) 335  
     auf lokale Objekte 221  
     Parameterübergabe per Z. 239  
 Zeile  
     einlesen *siehe* getline()  
     neue 55  
 Zeit-Server 619  
 Zeitkomplexität 986  
 Ziel (make) 652  
 Ziffernzeichen 54  
 Zufallszahlen 794  
 Zugriffsspezifizierer und -rechte 307  
 zusammengesetzte Datentypen 85  
 Zusicherung 143  
 Zustand 986  
     eines Iterators 464  
 Zuweisung 65, 69, 986  
     und Initialisierung 187  
     und Vererbung 310, 419  
 Zuweisungsoperator 188, 379, 496  
     implizite Deklaration 379, 419  
     und Vererbung 419  
 zweidimensionale Matrix 414  
 Zweierkomplement 41  
 Zwischenraumzeichen 105, 228, 425