

# C++



**Lernen und professionell anwenden**



### **Hinweis des Verlages zum Urheberrecht und Digitalen Rechtemanagement (DRM)**

Der Verlag räumt Ihnen mit dem Kauf des ebooks das Recht ein, die Inhalte im Rahmen des geltenden Urheberrechts zu nutzen. Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und Einspeicherung und Verarbeitung in elektronischen Systemen.

Der Verlag schützt seine ebooks vor Missbrauch des Urheberrechts durch ein digitales Rechtemanagement. Bei Kauf im Webshop des Verlages werden die ebooks mit einem nicht sichtbaren digitalen Wasserzeichen individuell pro Nutzer signiert.

Bei Kauf in anderen ebook-Webshops erfolgt die Signatur durch die Shopbetreiber. Angaben zu diesem DRM finden Sie auf den Seiten der jeweiligen Anbieter.

Ulla Kirch  
Peter Prinz

# **C++ Lernen und professionell anwenden**



**mitp**

Bibliografische Information der Deutschen Nationalbibliothek  
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;  
detaillierte bibliografische Daten sind im Internet über <<http://dnb.d-nb.de>> abrufbar.

ISBN 978-3-95845-809-3  
8., aktualisierte Auflage 2018

[www.mitp.de](http://www.mitp.de)  
E-Mail: [mitp-verlag@sigloch.de](mailto:mitp-verlag@sigloch.de)  
Telefon: +49 7953 / 7189 - 079  
Telefax: +49 7953 / 7189 - 082

© 2018 mitp Verlags GmbH & Co. KG

Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Lektorat: Sabine Schulz  
Sprachkorrektorat: Petra Heubach-Erdmann  
Coverbild: © Bruno Passigatti / [stock.adobe.com](http://stock.adobe.com)  
Satz: Ill-satz, Husby, [www.drei-satz.de](http://www.drei-satz.de)



**Vivi und Jeany gewidmet**



# Inhaltsverzeichnis

<b>Einleitung</b> .....	<b>21</b>
<b>1 Grundlagen</b> .....	<b>23</b>
Entwicklung und Eigenschaften von C++ .....	24
Objektorientierte Programmierung .....	26
Erstellen eines C++-Programms .....	28
Ein erstes C++-Programm .....	30
Struktur einfacher C++-Programme .....	32
Übungen .....	34
Lösungen .....	36
<b>2 Elementare Datentypen, Konstanten und Variablen</b> ..	<b>37</b>
Elementare Datentypen .....	38
Konstanten .....	44
Escape-Sequenzen .....	48
Namen .....	50
Variablen .....	52
Die Schlüsselworte const und volatile .....	54
Übungen .....	56
Lösungen .....	58
<b>3 Verwenden von Funktionen und Klassen</b> .....	<b>61</b>
Deklaration von Funktionen .....	62
Aufruf von Funktionen .....	64
Der Typ void für Funktionen .....	66
Header-Dateien .....	68
Standard-Header-Dateien .....	70
Verwenden von Standardklassen .....	72
Übungen .....	74
Lösungen .....	76

<b>4</b>	<b>Ein- und Ausgaben mit Streams</b>	<b>79</b>
	Streams	80
	Formatierung und Manipulatoren	82
	Formatierte Ausgabe von Ganzzahlen	84
	Formatierte Ausgabe von Gleitpunktzahlen	86
	Ausgabe in Felder	88
	Ausgabe von Zeichen, Strings und booleschen Werten	90
	Formatierte Eingabe	92
	Formatierte Eingabe von Zahlen	94
	Unformatierte Ein-/Ausgabe	96
	Übungen	98
	Lösungen	100
<b>5</b>	<b>Operatoren für elementare Datentypen</b>	<b>103</b>
	Binäre arithmetische Operatoren	104
	Unäre arithmetische Operatoren	106
	Zuweisungen	108
	Vergleichsoperatoren	110
	Logische Operatoren	112
	Übungen	114
	Lösungen	116
<b>6</b>	<b>Kontrollstrukturen</b>	<b>117</b>
	Die while-Schleife	118
	Die for-Schleife	120
	Die do-while-Schleife	124
	Verzweigungen mit if-else	126
	else-if-Ketten	128
	Bedingte Bewertung	130
	Auswahl mit switch	132
	Sprünge mit break, continue und goto	134
	Übungen	136
	Lösungen	138

<b>7</b>	<b>Symbolische Konstanten und Makros</b>	<b>141</b>
	Makros	142
	Makros mit Parametern	144
	Arbeiten mit der #define-Direktive	146
	Bedingte Kompilierung	148
	Standardmakros zur Behandlung von Zeichen	150
	Umlenken von Standardeingabe und -ausgabe	152
	Übungen	154
	Lösungen	156
<b>8</b>	<b>Umwandlung arithmetischer Datentypen</b>	<b>161</b>
	Implizite Typumwandlungen	162
	Verfahren bei arithmetischen Typumwandlungen	164
	Implizite Typumwandlungen bei Zuweisungen	166
	Weitere Typumwandlungen	168
	Übungen	170
	Lösungen	172
<b>9</b>	<b>Die Standardklasse string</b>	<b>175</b>
	Definition und Zuweisung von Strings	176
	Verketteten von Strings	178
	Strings vergleichen	180
	Einfügen und Löschen in Strings	182
	Suchen und Ersetzen in Strings	184
	Zugriff auf Zeichen in Strings	186
	Übungen	188
	Lösungen	190
<b>10</b>	<b>Funktionen</b>	<b>193</b>
	Bedeutung von Funktionen in C++	194
	Erstellen eigener Funktionen	196
	Return-Wert von Funktionen	198
	Übergabe von Argumenten	200



inline-Funktionen . . . . .	202
Default-Argumente . . . . .	204
Überladen von Funktionen . . . . .	206
Rekursive Funktionen . . . . .	208
Platzhalter auto für Return-Typen . . . . .	210
Übungen . . . . .	212
Lösungen . . . . .	215
 <b>11 Speicherklassen und Namensbereiche . . . . .</b>	 <b>221</b>
Speicherklasse von Objekten . . . . .	222
Der Speicherklassen-Spezifizierer extern . . . . .	224
Der Speicherklassen-Spezifizierer static . . . . .	226
Speicherklassen-Spezifizierer von Funktionen . . . . .	228
Namensbereiche . . . . .	230
Das Schlüsselwort using . . . . .	232
Übungen . . . . .	234
Lösungen . . . . .	238
 <b>12 Referenzen und Zeiger . . . . .</b>	 <b>243</b>
Definition von Referenzen . . . . .	244
Referenzen als Parameter . . . . .	246
Referenzen als Return-Wert . . . . .	248
Ausdrücke mit Referenztyp . . . . .	250
Definition von Zeigern . . . . .	252
Der Verweisoperator . . . . .	254
Zeiger als Parameter . . . . .	256
Übungen . . . . .	258
Lösungen . . . . .	260
 <b>13 Definition von Klassen . . . . .</b>	 <b>265</b>
Klassen-Konzept . . . . .	266
Definition von Klassen . . . . .	268
Definition von Methoden . . . . .	270
Definition von Objekten . . . . .	272

---

Verwendung von Objekten . . . . .	274
Zeiger auf Objekte . . . . .	276
Structs . . . . .	278
Structured Bindings . . . . .	280
Unions . . . . .	282
Übungen . . . . .	284
Lösung . . . . .	286
<b>14 Methoden . . . . .</b>	<b>289</b>
Konstrukturen . . . . .	290
Aufruf von Konstruktoren . . . . .	292
Mehr Initialisierungsmöglichkeiten . . . . .	294
Destrukturen . . . . .	296
Inline-Methoden. . . . .	298
Zugriffsmethoden . . . . .	300
const-Objekte und -Methoden . . . . .	302
Standardmethoden. . . . .	304
Standardmethoden kontrollieren . . . . .	306
Der this-Zeiger . . . . .	308
Übergabe von Objekten . . . . .	310
Objekte als Return-Wert . . . . .	312
Übungen . . . . .	314
Lösungen . . . . .	318
<b>15 Teilobjekte und statische Elemente . . . . .</b>	<b>325</b>
Klassen mit Teilobjekten . . . . .	326
Elementinitialisierer . . . . .	328
Konstante Teilobjekte . . . . .	330
Statische Datenelemente . . . . .	332
Zugriff auf statische Datenelemente . . . . .	334
Aufzählungen. . . . .	336
Übungen . . . . .	338
Lösungen . . . . .	342

<b>16 Vektoren</b>	<b>349</b>
Vektoren definieren	350
Initialisierung von Vektoren	352
C-Strings	354
Die Standardklasse <code>string_view</code>	356
Klassen-Arrays	358
Mehrdimensionale Vektoren	360
Vektoren als Datenelemente	362
Übungen	364
Lösungen	368
<b>17 Zeiger und Vektoren</b>	<b>379</b>
Vektoren und Zeiger (1)	380
Vektoren und Zeiger (2)	382
Zeigerarithmetik	384
Vektoren als Argumente von Funktionen	386
Zeigerversion von Funktionen	388
Read-only-Zeiger	390
Zeiger als Return-Wert	392
Zeigervektoren	394
Argumente aus der Kommandozeile	396
Übungen	398
Lösungen	402
<b>18 Grundlagen der Dateiverarbeitung</b>	<b>409</b>
Dateien	410
File-Stream-Klassen	412
File-Streams anlegen	414
Eröffnungsmodus	416
Schließen von Dateien	418
Blockweises Schreiben und Lesen	420
Persistenz von Objekten	422
Übungen	424
Lösungen	428

<b>19 Operatoren überladen</b>	<b>443</b>
Allgemeines	444
Operatorfunktionen (1)	446
Operatorfunktionen (2)	448
Überladene Operatoren verwenden	450
Globale Operatorfunktionen	452
friend-Funktionen	454
friend-Klassen	456
Index-Operator überladen	458
Shift-Operatoren für die Ein-/Ausgabe überladen	460
Übungen	462
Lösungen	464
<b>20 Typumwandlungen für Klassen</b>	<b>473</b>
Konvertierungskonstruktoren	474
Konvertierungsfunktionen	476
Mehrdeutigkeit bei Typumwandlungen	478
Übungen	480
Lösungen	482
<b>21 Speicherreservierung zur Laufzeit</b>	<b>485</b>
Der Operator new	486
Der Operator delete	488
Dynamischer Speicher für Klassen	490
Dynamischer Speicher für Vektoren	492
Anwendung: Einfach verkettete Listen	494
Darstellung einer einfach verketteten Liste	496
Übungen	498
Lösungen	500
<b>22 Dynamische Elemente</b>	<b>509</b>
Datenfelder variabler Länge	510
Eine Klasse mit dynamischem Element	512
Auf- und Abbau eines Objekts	514

Die Implementierung der Methoden . . . . .	516
Kopierkonstruktor . . . . .	518
Zuweisung . . . . .	520
Verschieben von R-Werten . . . . .	522
Move-Konstruktor . . . . .	524
Move-Zuweisung . . . . .	526
Übungen . . . . .	528
Lösungen . . . . .	530
<b>23 Vererbung . . . . .</b>	<b>539</b>
Konzept der Vererbung . . . . .	540
Abgeleitete Klassen . . . . .	542
Elemente abgeleiteter Klassen . . . . .	544
Elementzugriff . . . . .	546
Redefinition von Elementen . . . . .	548
Auf- und Abbau abgeleiteter Klassen . . . . .	550
Objekte abgeleiteter Klassen . . . . .	552
protected-Deklarationen . . . . .	554
Weitere Zugriffseinschränkungen . . . . .	556
Übungen . . . . .	558
Lösungen . . . . .	562
<b>24 Typumwandlung in Klassenhierarchien . . . . .</b>	<b>571</b>
Konvertierung in Basisklassen . . . . .	572
Typumwandlung bei Zuweisungen . . . . .	574
Konvertierung von Referenzen und Zeigern . . . . .	576
Explizite Typumwandlungen . . . . .	578
Übungen . . . . .	580
Lösungen . . . . .	582
<b>25 Polymorphe Klassen . . . . .</b>	<b>585</b>
Polymorphie . . . . .	586
Virtuelle Methoden . . . . .	588
Abbau dynamischer Objekte . . . . .	590



---

Virtuelle Methodentabelle . . . . .	592
Dynamische Casts . . . . .	594
Übungen . . . . .	596
Lösungen . . . . .	600
<b>26 Abstrakte Klassen . . . . .</b>	<b>607</b>
Rein virtuelle Methoden . . . . .	608
Abstrakte und konkrete Klassen . . . . .	610
Zeiger und Referenzen auf abstrakte Klassen . . . . .	612
Virtuelle Zuweisungen . . . . .	614
Anwendung: Inhomogene Listen. . . . .	616
Implementierung einer inhomogenen Liste . . . . .	618
Übungen . . . . .	620
Lösungen . . . . .	622
<b>27 Mehrfachvererbung . . . . .</b>	<b>629</b>
Mehrfach abgeleitete Klassen . . . . .	630
Mehrfache indirekte Basisklassen . . . . .	632
Virtuelle Basisklassen . . . . .	634
Aufrufe von Konstruktoren . . . . .	636
Initialisierung virtueller Basisklassen . . . . .	638
Übungen . . . . .	640
Lösungen . . . . .	644
<b>28 Ausnahmebehandlung . . . . .</b>	<b>649</b>
Traditionelle Fehlerbehandlung . . . . .	650
Exception-Handling . . . . .	652
Exception-Handler . . . . .	654
Auslösen und Auffangen von Exceptions . . . . .	656
Schachteln von Ausnahmebehandlungen . . . . .	658
Definition eigener Fehlerklassen. . . . .	660
Standardfehlerklassen . . . . .	662
Übungen . . . . .	664
Lösungen . . . . .	668

<b>29 Mehr über Dateien</b>	<b>679</b>
Dateien für wahlfreien Zugriff öffnen	680
Wahlfreies Positionieren	682
Dateistatus	686
Exception-Handling für Dateien	688
Persistenz von polymorphen Objekten	690
Anwendung: Indexdateien	694
Implementierung eines Index-Dateisystems	696
Portabler Zugriff auf das Dateisystem	698
Übungen	702
Lösungen	706
<b>30 Mehr über Zeiger</b>	<b>727</b>
Zeiger auf Zeiger	728
Variable Anzahl von Argumenten	730
Zeiger auf Funktionen	734
Komplexe Deklarationen	736
Definition von Typnamen	738
Matrizen als Argumente von Funktionen	740
Anwendung: Dynamische Matrizen	742
Übungen	744
Lösungen	746
<b>31 Bitmanipulationen</b>	<b>755</b>
Logische Bitoperatoren	756
Shift-Operatoren	758
Bitmasken	760
Verwenden von Bitmasken	762
Bitfelder	764
Übungen	766
Lösungen	768
<b>32 Templates</b>	<b>773</b>
Funktions- und Klassen-Templates	774
Definition von Templates	776

---

Instantiierung von Templates . . . . .	778
Template-Parameter . . . . .	780
Template-Argumente . . . . .	782
Spezialisierungen . . . . .	784
Default-Argumente von Templates . . . . .	786
Explizite Instantiierung . . . . .	788
Variable Anzahl von Parametern . . . . .	790
Operationen mit Parameterpacks . . . . .	792
Standard-Template für Tupel . . . . .	794
Typsichere Varianten . . . . .	796
Die Standard-Klassen optional und any . . . . .	798
Standard-Templates für Bitsets . . . . .	800
Übungen . . . . .	804
Lösungen . . . . .	808
 <b>33 Container . . . . .</b>	 <b>815</b>
Arten von Containern . . . . .	816
Sequenzielle Container-Klassen . . . . .	818
Iteratoren . . . . .	820
Vereinbarung sequenzieller Container . . . . .	822
Einfügen in sequenziellen Containern . . . . .	824
Elementzugriff . . . . .	826
Größe von Containern . . . . .	828
Löschen in sequenziellen Containern . . . . .	830
Listenoperationen . . . . .	832
Assoziative Container . . . . .	834
Sets und Multisets . . . . .	836
Maps und Multimaps . . . . .	838
Initialisierungslisten und Range-for-Schleifen . . . . .	840
Hash-Tabellen . . . . .	842
Ungeordnete assoziative Container . . . . .	844
Hash-Tabellen für selbstdefinierte Typen . . . . .	846
Übungen . . . . .	848
Lösungen . . . . .	852

<b>34 Smart Pointer</b>	<b>861</b>
Grundlagen	862
Die Smart Pointer der Standardbibliothek	864
Smart Pointer vom Typ <code>unique_ptr&lt;T&gt;</code> einsetzen	866
Smart Pointer vom Typ <code>unique_ptr&lt;T&gt;</code> einsetzen (2)	868
Das Klassen-Template <code>shared_ptr</code>	870
Das Klassen-Template <code>shared_ptr</code> (2)	872
Das Klassen-Template <code>shared_ptr</code> (3)	874
Das Klassen-Template <code>weak_ptr</code>	876
Übungen	878
Lösungen	882
<b>35 Multithreading</b>	<b>893</b>
Threads	894
Erzeugen und Ausführen von Threads	896
Initialisierung und Zuweisung	898
Konkurrierende Speicherzugriffe	900
Thread-ID	902
Locks	904
Bedingungsvariablen	906
Übungen	910
Lösungen	914
<b>36 Algorithmen der Standardbibliothek</b>	<b>921</b>
Algorithmen: Grundlagen	922
Funktionsobjekte	924
Lambda-Funktionen	928
Algorithmen und Iteratoren	930
Vordefinierte Iteratoren	932
Nicht-modifizierende Algorithmen	934
Modifizierende Algorithmen	938
Mutierende Algorithmen	942
Sortieren und verwandte Operationen	944

<b>37 Numerische Bibliothek</b>	<b>949</b>
Komplexe Zahlen	950
Das Klassen-Template <code>complex&lt;T&gt;</code>	952
Methoden von <code>complex&lt;T&gt;</code>	954
Komplexe Funktionen	956
Das Klassen-Template <code>valarray&lt;T&gt;</code>	958
Zuweisung numerischer Vektoren	960
Arithmetische Operationen	962
Weitere Operationen	964
Selektive Indizierung	966
Verallgemeinerte selektive Indizierung	968
Indirekte und maskierte Indizierung	972
<b>Anhang A</b>	<b>975</b>
Binäre Zahlendarstellung	976
Präprozessor-Direktiven	979
Vordefinierte Standardmakros	986
Einbinden von C-Funktionen	987
Operatorenübersicht	989
Vorrangtabelle für Operatoren	991
ASCII-Code-Tabelle	992
Bildschirmsteuerzeichen	994
Compiler, Beispiele und Lösungen	995
<b>Anhang B</b>	<b>999</b>
Glossar	999
<b>Stichwortverzeichnis</b>	<b>1019</b>





# Einleitung

Dieses Buch wendet sich an jeden Leser, der die Programmiersprache C++ neu lernen oder vertiefen möchte, egal ob Anfänger oder fortgeschrittener C++-Programmierer.

C++ ist eine weitgehend plattformunabhängige Sprache. Die Sprachbeschreibung basiert auf der ISO-Norm 14882, die 1998 von der International Organization for Standardization verabschiedet wurde. Dieser traditionelle ISO-Standard wird von allen gängigen C++-Compilern unterstützt. Auch Sprachelemente aus den Standards von 2011, *kurz: C++11*, (wie z.B. Threads, Smart Pointer und Hash-Tabellen) und von 2014, *kurz: C++14*, (wie z.B. Ableitung von Return-Typen und generische Lambdas) werden heute von aktuellen Compilern unterstützt. Diese früheren Standards werden im Buch der Einfachheit halber mit „C++-Standard“ bezeichnet. Neue Sprachelemente aus dem Standard von 2017, *kurz: C++17*, (wie z.B. Structured Bindings, Faltung von Parameterpacks und die Filesystem-Bibliothek) werden noch nicht von allen C++-Compilern unterstützt. Sie werden im Buch deshalb mit C++17 gekennzeichnet.

Die *Kapitel* dieses Buches sind so angeordnet, dass der Leser von elementaren Sprachkonzepten bis hin zur professionellen Software-Entwicklung geführt wird. Hierbei werden alle Sprachelemente umfassend behandelt. Die Reihenfolge der Darstellung orientiert sich am Ziel, von Anfang an sinnvolle Programme schreiben zu können.

Jede *Doppelseite* im Buch ist wie folgt gegliedert: Auf der rechten Seite sind die Sprachelemente beschrieben, die durch Grafiken und C++-Programme auf der linken Seite illustriert werden. Die Beispielprogramme sind so ausgewählt, dass sie eine typische Anwendung für das jeweilige Sprachelement zeigen. Darüber hinaus machen Filterprogramme und Fallstudien den Leser mit einem breiten Anwendungsspektrum vertraut.

Um eine leistungsfähige Programmiersprache zu beherrschen, ist viel eigene Erfahrung durch selbständiges Entwickeln von Programmen erforderlich. Am Ende jedes Kapitels sind deshalb *Übungen* formuliert, zu denen auf den folgenden Seiten *Musterlösungen* angegeben sind. Der Leser kann damit seine erreichten Fähigkeiten testen und das Verständnis von C++ vertiefen.

Im *Anhang* sind nützliche Informationen wie z.B. die binäre Zahlendarstellung, Präprozessor-Direktiven und die Vorrangtabelle für Operatoren zusammengestellt. Das Buch soll damit auch den geübten C++-Programmierer begleiten, der ein gut strukturiertes und verständliches Nachschlagewerk zur Hand haben will.

Die Entwicklungsumgebung Visual Studio Community von Microsoft ist unter [www.visualstudio.com](http://www.visualstudio.com)

zum Download kostenlos verfügbar. Damit können Sie auf einem Windows-System (ab Windows 7, Service Pack 1) Ihre Programme erstellen und testen.

Hinweise zur Installation und Verwendung des Compilers finden Sie im Anhang des Buches. Es kann aber auch jeder andere C++-Compiler verwendet werden, beispielsweise der GNU-Compiler unter Linux. Die Programmbeispiele und Lösungen zu den Aufgaben stehen für Sie beim mitp-Verlag unter

[www.mitp.de/808](http://www.mitp.de/808)

zum Download bereit.

Wir bedanken uns bei allen, die an der Entstehung des Buches mitgewirkt haben, insbesondere bei Frau Schulz vom mitp-Verlag für die konstruktive Zusammenarbeit, und auch bei unseren Kindern, die uns oft genug in Ruhe lassen konnten.

Dem Leser wünschen wir **viel Spaß mit C++!**

Ulla Kirch  
[kirch@hm.edu](mailto:kirch@hm.edu)

Peter Prinz  
[prinz\\_peter@t-online.de](mailto:prinz_peter@t-online.de)

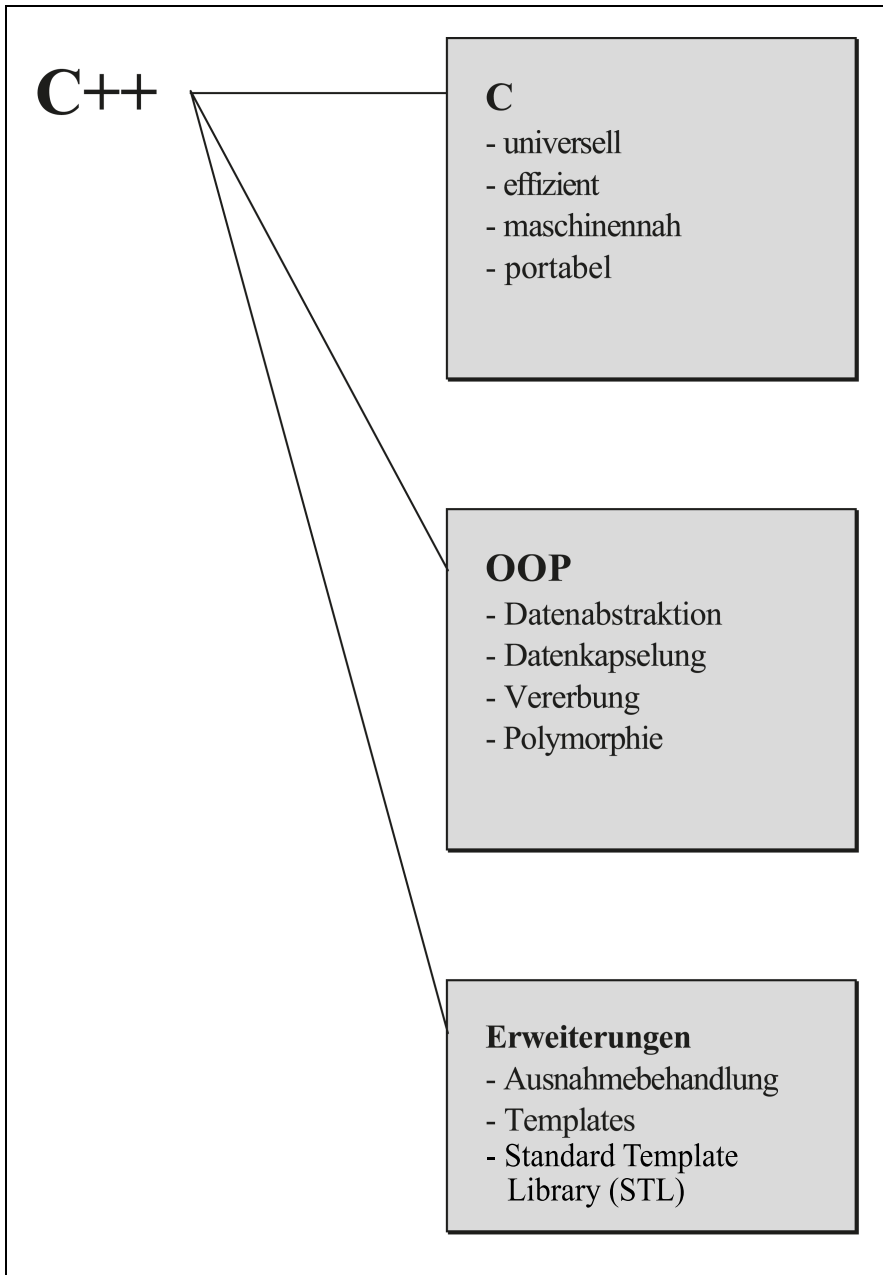
# Kapitel 1

## Grundlagen

Dieses Kapitel beschreibt die grundlegenden Eigenschaften der objektorientierten Programmiersprache C++. Außerdem werden die Schritte vorgestellt, die zur Erstellung eines lauffähigen C++-Programms erforderlich sind. Diese Schritte können Sie auf Ihrem System anhand von einführenden Beispielen nachvollziehen. Die Beispiele dienen auch dazu, die grundlegende Struktur eines C++-Programms darzustellen.

# Entwicklung und Eigenschaften von C++

## Charakteristische Eigenschaften





## Historisches

Die Programmiersprache C++ wurde von Bjarne Stroustrup und seinen Mitarbeitern in den Bell Laboratories (AT&T, USA) entwickelt, um Simulationsprojekte objektorientiert und effizient implementieren zu können. Frühe Versionen, die zunächst als „C mit Klassen“ bezeichnet wurden, gibt es seit 1980. Der Name C++ weist darauf hin, dass C++ aus der Programmiersprache C hervorgegangen ist: ++ ist der Inkrementoperator von C.

Schon 1989 wurde ein ANSI-Komitee (American National Standards Institute) gebildet, um die Programmiersprache C++ zu standardisieren. Hierbei geht es darum, dass möglichst viele Compilerbauer und Software-Entwickler sich auf eine einheitliche Sprachbeschreibung einigen, um die Bildung von Dialekten und „Sprachverwirrungen“ zu vermeiden.

Anfang 1998 wurde von der ISO (International Organization for Standardization) der Standard für C++ (14882) verabschiedet, der inzwischen mehrfach erweitert und zuletzt 2017 ergänzt wurde. Der neueste Standard von 2017 wird auch kurz mit C++17 bezeichnet.

## Eigenschaften von C++

C++ ist keine rein objektorientierte Sprache, sondern eine Hybridsprache (= „Mischsprache“): Sie enthält die Programmiersprache C als Teilmenge. Damit hat man zunächst alle Möglichkeiten, die auch C bietet:

- universell einsetzbare, modulare Programme
- effiziente, maschinennahe Programmierung
- Portabilität, d.h. Übertragbarkeit von Programmen auf verschiedene Rechner

Insbesondere kann die umfangreiche in C entwickelte Software auch in C++-Programmen verwendet werden.

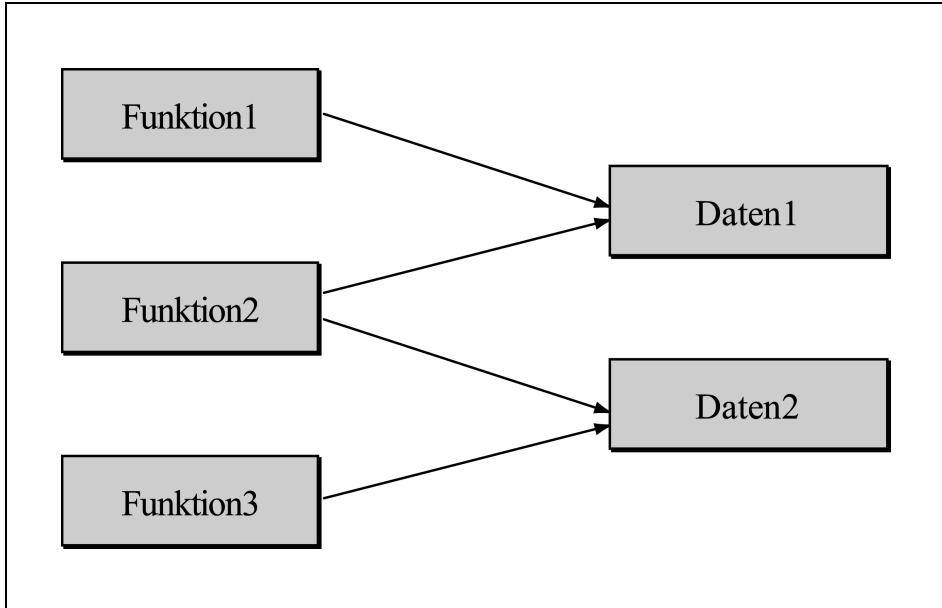
C++ unterstützt die Konzepte der objektorientierten Programmierung, nämlich:

- *Datenabstraktion*, d.h. Bildung von Klassen zur Beschreibung von Objekten
- *Datenkapselung* für den kontrollierten Zugriff auf die Daten von Objekten
- *Vererbung* durch Bildung abgeleiteter Klassen (auch mehrfach)
- *Polymorphie* (griech. „Vielgestaltigkeit“), d.h. die Implementierung von Anweisungen, die zur Laufzeit verschiedene Wirkungen haben können

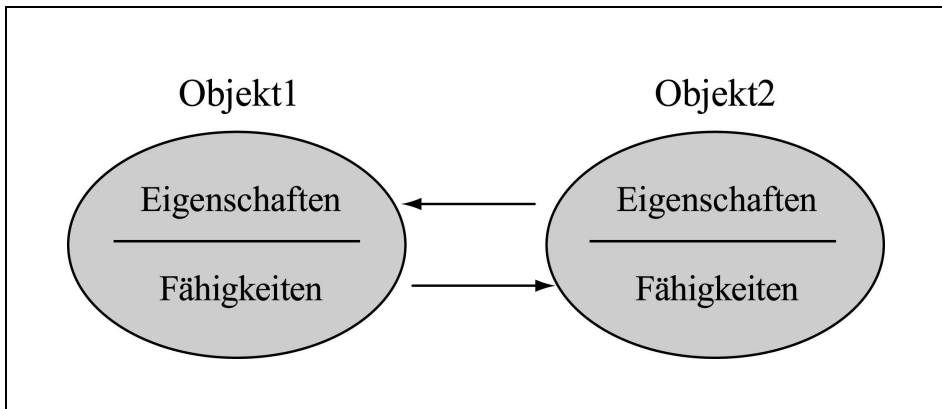
C++ wurde um zusätzliche Sprachelemente erweitert, wie z.B. Referenzen, Templates und Ausnahmebehandlung (engl. Exception Handling). Auch wenn diese Sprachelemente keinen direkten Bezug zur Objektorientierung haben, sind sie für deren effiziente Implementierung wichtig.

# Objektorientierte Programmierung

## Traditionelles Konzept



## Objektorientiertes Konzept



## Klassische, prozedurale Programmierung

Die traditionelle, prozedurale Programmierung trennt Daten und Funktionen (= Unterprogramme, Prozeduren), die diese Daten bearbeiten. Dies hat wichtige Konsequenzen für den Umgang mit den Daten in einem Programm:

- Der Programmierer muss selbst dafür sorgen, dass Daten vor ihrer Verwendung mit geeigneten Anfangswerten versehen sind und dass beim Aufruf einer Funktion korrekte Daten übergeben werden.
- Wird die Darstellung der Daten geändert, z.B. ein Datensatz erweitert, so müssen auch die zugehörigen Funktionen entsprechend angepasst werden.

Beides ist natürlich fehleranfällig und nicht besonders wartungsfreundlich.

## Objekte

Die objektorientierte Programmierung (OOP) stellt die *Objekte* in den Mittelpunkt, d.h. die Dinge, um die es bei der jeweiligen Problemstellung geht. Ein Programm zur Verwaltung von Konten beispielsweise arbeitet mit Kontoständen, Kreditlimits, Überweisungen, Zinsberechnungen usw. Ein Objekt, das ein Konto in einem Programm darstellt, besitzt dann die Eigenschaften und Fähigkeiten, die für die Kontoverwaltung wichtig sind.

Die Objekte in der OOP bilden eine Einheit aus Daten (= Eigenschaften) und Funktionen (= Fähigkeiten). Mit einer Klasse wird ein Objekt-Typ definiert, der sowohl die Eigenschaften als auch die Fähigkeiten von Objekten dieses Typs festlegt. Die Kommunikation zwischen Objekten erfolgt dann durch „Nachrichten“, die die Fähigkeiten von Objekten aktivieren.

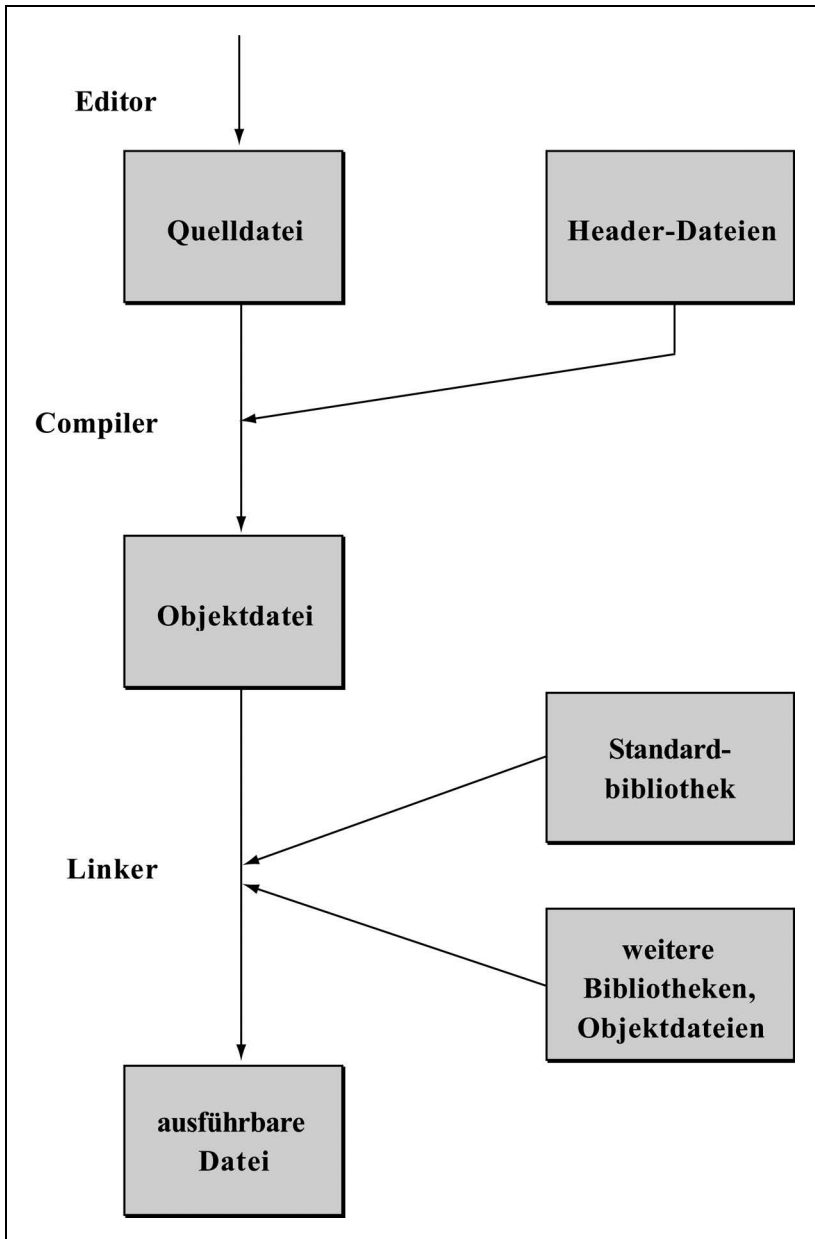
## Vorteile der OOP

Für die Software-Entwicklung hat die objektorientierte Programmierung wesentliche Vorteile:

- **geringere Fehleranfälligkeit:** Ein Objekt kontrolliert den Zugriff auf seine Daten selbst. Insbesondere kann es fehlerhafte Zugriffe abwehren.
- **gute Wiederverwendbarkeit:** Ein Objekt verwaltet sich selbst. Es kann deshalb wie ein Baustein in beliebigen Programmen eingesetzt werden.
- **geringer Wartungsaufwand:** Ein Objekt kann die interne Darstellung seiner Daten an neue Anforderungen anpassen, ohne dass ein Anwendungsprogramm etwas davon merkt.

# Erstellen eines C++-Programms

## Übersetzen eines C++-Programms



Zum Erstellen und Übersetzen eines C++-Programms sind folgende drei Schritte notwendig:

1. Das C++-Programm wird mit einem Texteditor in eine Datei eingegeben. Anders gesagt: Der *Quellcode* wird in einer *Quelldatei* abgelegt.

Bei größeren Projekten ist es üblich, *modular* zu programmieren. Dabei wird der Quellcode auf mehrere Quelldateien verteilt, die getrennt bearbeitet und übersetzt werden.

2. Die Quelldatei wird dem *Compiler* zur Übersetzung gegeben. Geht alles gut, so erhält man eine Objektdatei, die den *Maschinencode* enthält. Eine Objektdatei heißt auch *Modul*.
3. Schließlich bindet der *Linker* die Objektdatei mit anderen Modulen zu einer *ausführbaren Datei*. Die Module bestehen aus Funktionen der Standardbibliothek oder selbsterstellten, schon früher übersetzten Programmteilen.

Im *Namen* der Quelldateien muss die richtige Endung angegeben werden. Diese hängt vom jeweiligen Compiler ab. Die gebräuchlichsten Endungen sind `.cpp` und `.cc`.

Vor dem eigentlichen Kompilervorgang können *Header-Dateien*, auch Include-Dateien genannt, in die Quelldatei kopiert werden. Header-Dateien sind Textdateien mit Informationen, die in verschiedenen Quelldateien gebraucht werden, wie z.B. Typdefinitionen oder die Deklaration von Variablen und Funktionen. Die Namen von Header-Dateien enden entweder mit `.h` oder haben keine Endung.

Die C++-*Standardbibliothek* enthält fertige Funktionen und Klassen, die standardisiert sind und die jeder Compiler zur Verfügung stellt.

Moderne Compiler bieten eine *integrierte Entwicklungsumgebung*, die die obigen drei Schritte zusammenfasst. Von einer einheitlichen Benutzeroberfläche aus wird das Programm editiert, kompiliert, gelinkt und ausgeführt. Außerdem können weitere Tools, wie z.B. ein Debugger, gestartet werden.

**Wichtig:** Enthält die Quelldatei auch nur einen *Syntaxfehler*, so zeigt dies der Compiler mit einer *Fehlermeldung* an. Darüber hinaus können weitere Fehlermeldungen ausgegeben werden, die darauf beruhen, dass der Compiler versucht, trotz des Fehlers weiter zu übersetzen. Beginnen Sie also immer mit der Korrektur des ersten Fehlers in einem Programm.

Neben den eigentlichen Fehlermeldungen gibt der Compiler auch *Warnungen* aus. Eine Warnung zeigt keinen Syntaxfehler an, sondern ist lediglich ein Hinweis auf einen möglichen logischen Fehler, wie beispielsweise die Verwendung einer nicht initialisierten Variablen.

# Ein erstes C++-Programm

## Beispielprogramm

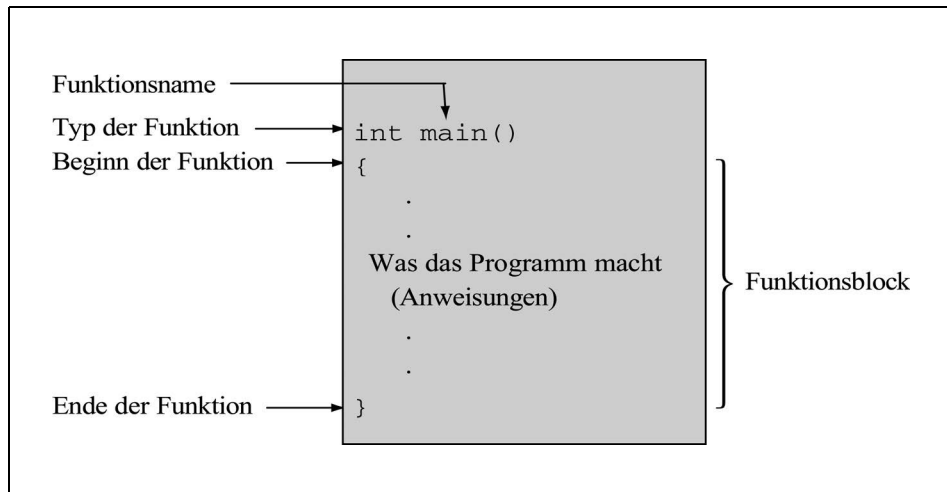
```
#include <iostream>
using namespace std;

int main()
{
    cout << "Viel Spaß mit C++!" << endl;
    return 0;
}
```

## BildschirmAusgabe

Viel Spaß mit C++!

## Die Struktur der Funktion main()



Ein C++-Programm besteht aus Objekten mit ihren *Elementfunktionen* und aus *globalen Funktionen*, die zu keiner Klasse gehören. Jede Funktion löst eine bestimmte Aufgabe und kann andere Funktionen aufrufen. Sie ist entweder selbsterstellt oder eine fertige Routine aus der Standardbibliothek. Die globale Funktion `main()` muss immer selbst geschrieben werden und hat eine besondere Rolle: Sie bildet das Hauptprogramm.

Anhand des nebenstehenden kurzen Beispielprogramms lassen sich bereits die wichtigsten Elemente eines C++-Programms erkennen. Das Programm enthält nur die Funktion `main()` und gibt eine Meldung auf dem Bildschirm aus.

Die erste Zeile beginnt mit einem Doppelkreuz `#` und ist daher für den *Präprozessor* bestimmt. Der Präprozessor ist Teil der ersten Übersetzungsphase, in der noch kein Objektcode erzeugt wird. Mit

```
#include <dateiname>
```

kopiert der Präprozessor die genannte Datei an diese Stelle in den Quellcode. Dadurch stehen dem Programm alle Informationen zur Verfügung, die in dieser Datei enthalten sind.

Die Header-Datei `iostream` enthält Vereinbarungen für die Ein-/Ausgabe mit Streams. Der Begriff *Stream* (dt. Strom) spiegelt die Situation wider, dass Zeichenfolgen wie ein Datenstrom behandelt werden.

Die in C++ vordefinierten Namen befinden sich im Namensbereich `std` (standard). Die anschließende `using`-Direktive erlaubt, die Namen aus dem Namensbereich (*Namespace*) `std` direkt zu verwenden.

Die Programmausführung beginnt mit der ersten Anweisung in der Funktion `main()`, die daher in jedem C++-Programm vorhanden sein muss. Nebenstehend ist die Struktur der Funktion dargestellt. Sie unterscheidet sich, abgesehen vom feststehenden Namen, nicht von der Struktur anderer Funktionen.

In unserem Beispiel enthält die Funktion `main()` zwei *Anweisungen*. Die erste

```
cout << "Viel Spaß mit C++!" << endl;
```

gibt den Text `Viel Spaß mit C++!` auf dem Bildschirm aus. Der Name `cout` (*console output*) bezeichnet ein Objekt, das die Ausgabe durchführt.

Die beiden Kleiner-Zeichen `<<` deuten an, dass die Zeichen in den Ausgabestrom „geschoben“ werden. Mit `endl` (end of line) wird anschließend ein Zeilenvorschub ausgelöst. Die zweite Anweisung

```
return 0;
```

beendet die Funktion `main()` und damit das Programm. Dabei wird der Wert `0` dem aufrufenden Programm als Exit-Code zurückgegeben. Es ist üblich, den Exit-Code `0` zu verwenden, wenn das Programm korrekt abgelaufen ist.

Beachten Sie, dass die Anweisungen mit einem Semikolon enden. Die kürzeste Anweisung besteht übrigens nur aus einem Semikolon und bewirkt nichts.

# Struktur einfacher C++-Programme

## Ein C++-Programm mit mehreren Funktionen

```

/*****
    Ein Programm mit mehreren Funktionen und Kommentaren
*****/

#include <iostream>
using namespace std;

void linie(), meldung();           // Prototypen

int main()
{
    cout << "Hallo! Das Programm startet in main()."
          << endl;
    linie();
    meldung();
    linie();
    cout << "Jetzt am Ende von main()." << endl;

    return 0;
}

void linie()                       // Eine Linie ausgeben.
{
    cout << "-----" << endl;
}

void meldung()                     // Eine Meldung ausgeben.
{
    cout << "In der Funktion meldung()." << endl;
}

```

## Bildschirmausgabe

```

Hallo! Das Programm startet in main().
-----
In der Funktion meldung().
-----
Jetzt am Ende von main().

```



Das nebenstehende Beispiel zeigt, wie ein C++-Programm strukturiert ist, das aus mehreren Funktionen besteht. Die Reihenfolge, in der Funktionen definiert werden, ist in C++ nicht vorgeschrieben. Zum Beispiel könnte auch zuerst die Funktion `meldung()`, dann die Funktion `linie()` und schließlich die `main`-Funktion geschrieben werden.

Üblicherweise wird die Funktion `main()` zuerst angegeben, da sie den Programmablauf steuert. Es werden also Funktionen aufgerufen, die erst später definiert werden. Dies ist möglich, da der Compiler mit dem *Prototyp* einer Funktion die notwendigen Informationen erhält.

Neu in diesem Beispiel sind die *Kommentare*. Jede Zeichenfolge, die durch `/* . . . */` eingeschlossen ist oder mit `//` beginnt, ist ein Kommentar.

**Beispiele:** `/* Ich darf mehrere Zeilen lang sein */`  
`// Ich bin ein Einzeilen-Kommentar`

Beim Einzeilen-Kommentar ignoriert der Compiler ab den Zeichen `//` alle Zeichen bis zum Zeilenende. Kommentare, die sich über mehrere Zeilen erstrecken, sind bei der Fehlersuche nützlich, um ganze Programmteile auszublenden. Jede der beiden Kommentarformen kann auch benutzt werden, um die andere auszukommentieren.

Zum *Layout* einer Quelldatei: Der Compiler bearbeitet jede Quelldatei sequenziell und zerlegt den Inhalt in „Token“ (kleinste Bestandteile), wie z.B. Funktionsnamen und Operatoren. Token können durch beliebig viele Zwischenraumzeichen getrennt sein, also durch Leer-, Tabulator- oder Newline-Zeichen. Es kommt daher nur auf die Reihenfolge des Quellcodes an, nicht auf ein bestimmtes Layout, wie etwa die Aufteilung in Zeilen und Spalten. Beispielsweise ist

```
void meldung
( ){ cout <<
    "In der Funktion meldung()." <<
endl;}
```

zwar eine schlecht lesbare, aber korrekte Definition der Funktion `meldung()`.

Eine Ausnahme hinsichtlich des Layouts bilden die Präprozessor-Direktiven, die stets eine eigene Zeile einnehmen. Dem Doppelkreuz `#` zu Beginn einer Zeile dürfen nur Leer- und Tabulatorzeichen vorangehen.

Zur besseren Lesbarkeit von C++-Programmen ist es von Vorteil, einen einheitlichen Stil in der Darstellung beizubehalten. Dabei sollten Einrückungen und Leerzeilen gemäß der Programmstruktur eingefügt werden. Außerdem ist eine großzügige Kommentierung wichtig.

# Übungen

## Programm-Listing zur 3. Aufgabe

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

void pause();          // Prototyp

int main()
{
    cout << endl << "Lieber Leser, "
         << endl << "goennen Sie"
         << " sich jetzt eine ";
    pause();
    cout << "!" << endl;

    return 0;
}

void pause()
{
    cout << "PAUSE";
}
```

### 1. Aufgabe

Erstellen Sie ein C++-Programm, das den folgenden Text auf dem Bildschirm ausgibt:

```
Moegen
taeten wir schon wollen,
aber koennen
haben wir uns nicht getraut!
```

Verwenden Sie den Manipulator endl an den entsprechenden Stellen.

### 2. Aufgabe

Das folgende Programm enthält lauter Fehler:

```
*/ Hier sollte man die Brille nicht vergessen //
#include <stream>
int main
{
    cout << "Wenn dieser Text",
    cout >> " auf Ihrem Bildschirm erscheint, ";
    cout << " endl;"
    cout << 'können Sie sich auf die Schulter '
        << "klopfen!" << endl.
    return 0;
}
```

Korrigieren Sie die Fehler, und testen Sie die Lauffähigkeit des Programms.

### 3. Aufgabe

Was gibt das nebenstehende C++-Programm auf dem Bildschirm aus?

# Lösungen

## Zur 1. Aufgabe:

```
// Auf geht's !

#include <iostream>
using namespace std;

int main()
{
    cout << "Moegen " << endl;
    cout << "taeten wir schon wollen" << endl;
    cout << "aber koennen " << endl;
    cout << "haben wir uns nicht getraut!" << endl;

    return 0;
}
```

## Zur 2. Aufgabe:

Die korrigierten Stellen sind unterstrichen.

```
/* Hier sollte man die Brille nicht vergessen */
#include <iostream>

using namespace std;
int main(())
{
    cout << "Wenn dieser Text";
    cout <= " auf Ihrem Bildschirm erscheint, ";
    cout << endl;
    cout << "können Sie sich auf die Schulter "
        << "klopfen!" << endl;
    return 0;
}
```

## Zur 3. Aufgabe:

Die Bildschirmausgabe des Programms beginnt in einer neuen Zeile:

```
Lieber Leser,
goennen Sie sich jetzt eine PAUSE!
```

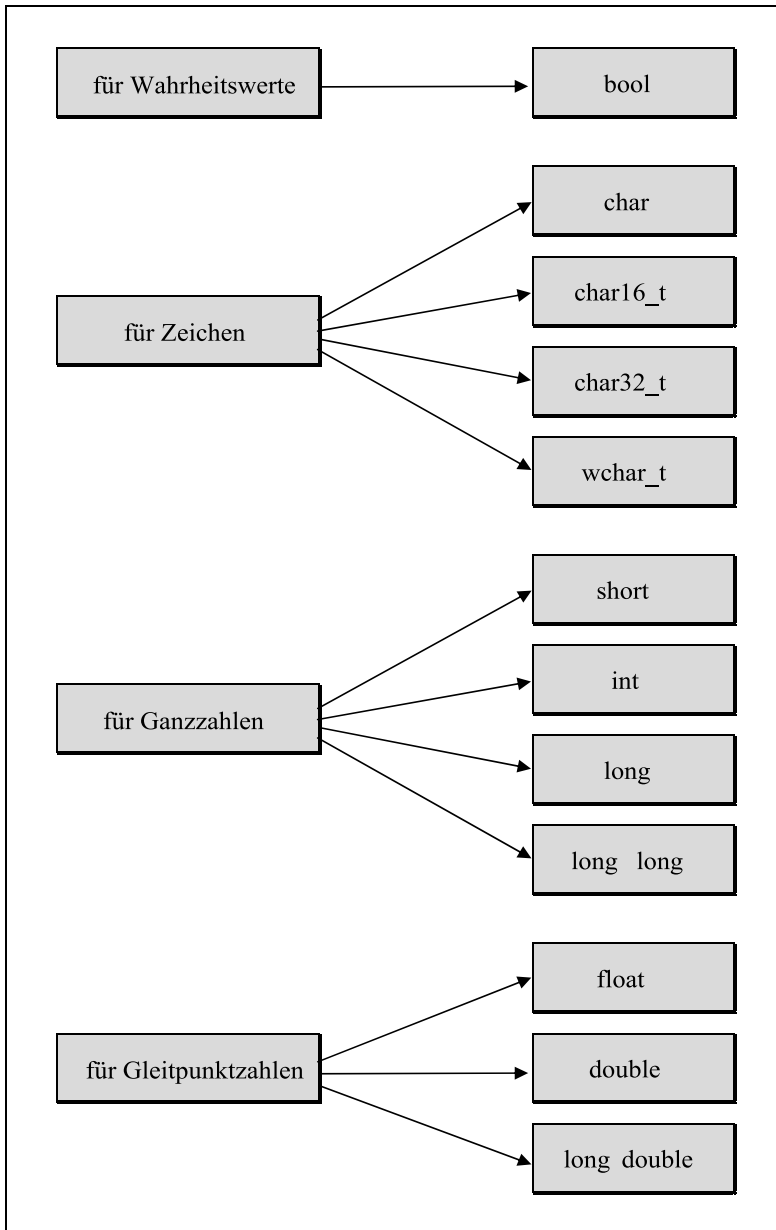
# Kapitel 2

## Elementare Datentypen, Konstanten und Variablen

In diesem Kapitel werden Sie die grundlegenden Typen und Objekte kennenlernen, mit denen ein Programm arbeitet.

# Elementare Datentypen

## Übersicht \*)



\*) Ohne den Datentyp void, der später behandelt wird.

Ein Programm benutzt zur Lösung einer Aufgabe verschiedenartige Daten, z. B. Zeichen, Ganzzahlen oder Gleitpunktzahlen. Da diese vom Computer unterschiedlich bearbeitet und gespeichert werden, muss der *Typ* der Daten bekannt sein. Ein Datentyp bestimmt:

1. die Art der internen Darstellung der Daten
2. die Größe des dazu benötigten Speicherplatzes

Beispielsweise kann eine ganze Zahl wie  $-1000$  in zwei oder vier Bytes gespeichert werden. Wenn diese Speicherstelle wieder gelesen wird, ist es wichtig, die richtige Anzahl an Bytes zu lesen. Außerdem muss der Inhalt, also das gelesene Bitmuster, richtig interpretiert werden, nämlich als ganze Zahl mit Vorzeichen.

Der C++-Compiler kennt die nebenstehend aufgeführten elementaren Datentypen, auf denen alle weiteren Datentypen (Vektoren, Zeiger, Klassen, ...) aufbauen.

## Der Datentyp `bool`

Das Ergebnis eines Vergleichs oder einer logischen Verknüpfung mit UND bzw. ODER ist ein Wahrheitswert (*boolescher Wert*), der wahr oder falsch sein kann. Zur Darstellung boolescher Werte gibt es in C++ den Typ `bool`. Ein Ausdruck vom Typ `bool` besitzt entweder den Wert `true` (wahr) oder `false` (falsch). Dabei wird `true` intern durch den numerischen Wert 1 und `false` durch 0 repräsentiert.

## Die Datentypen für Zeichen

Jedem Zeichen ist eine ganze Zahl, der so genannte *Zeichencode*, zugeordnet, beispielsweise dem Zeichen A der Code 65. Die Zuordnung ist durch den verwendeten Zeichensatz definiert.

In C++ ist nicht festgelegt, welcher Zeichensatz zu verwenden ist. Üblich ist ein Zeichensatz, der den *ASCII-Code* (American Standard Code for Information Interchange) enthält. Dieser 7-Bit-Code definiert die Codes von 33 Steuerzeichen (Codes 0-31 u. 127) und 95 druckbaren Zeichen (Codes 32-126).

Mit dem Typ `char` („character“) werden Zeichencodes in einem Byte (= 8 Bits) gespeichert. Das genügt für einen erweiterten Zeichensatz, z. B. für den ANSI-Zeichensatz, der außer den Zeichen des ASCII-Codes weitere Zeichen, wie etwa deutsche Umlaute, enthält.

Der Typ `wchar_t` („wide character type“) ist mindestens 2 Bytes (= 16 Bits) groß. Dieser Typ ist geeignet, die Codes aller Zeichen zu speichern, die bei der Ausführung des Programms vorkommen können.

Mit den neuen Typen `char16_t` und `char32_t` unterstützt C++ die *Unicode-Standards* UTF-16 und UTF-32 („*Unicode transformation formats*“). Unicode wird von allen modernen Systemen verwendet und die Codierung stimmt mit der des Standards ISO 10646 überein. In der Version 6.0 von 2011 umfasst der Unicode mehr als 100.000 Zeichen aus nahezu allen Schriften der Welt.

## Elementare Datentypen (Fortsetzung)

### Die ganzzahligen Datentypen

Typ	Speicherplatz	Wertebereich (dezimal)
char	1 Byte	-128 bis +127 bzw. 0 bis 255
unsigned char	1 Byte	0 bis 255
signed char	1 Byte	-128 bis +127
int	2 Byte bzw. 4 Byte	-32768 bis +32767 bzw. -2147483648 bis +2147483647
unsigned int	2 Byte bzw. 4 Byte	0 bis 65535 bzw. 0 bis 4294967295
short	2 Byte	-32768 bis +32767
unsigned short	2 Byte	0 bis 65535
long	4 Byte	-2147483648 bis +2147483647
unsigned long	4 Byte	0 bis 4294967295
long long	8 Byte	-9223372036854775808 bis +9223372036854775807
unsigned long long	8 Byte	0 bis 18446744073709551615

### Beispielprogramm

```

#include <iostream>
#include <climits>          // Definition von INT_MIN, ...
using namespace std;

int main()
{
    cout << "Wertebereiche der Typen int und unsigned int"
          << endl << endl;
    cout << "Typ           Minimum           Maximum"
          << endl
          << "-----"
          << endl;

    cout << "int           " << INT_MIN << "           "
          << INT_MAX << endl;

    cout << "unsigned int   " << "           0           "
          << UINT_MAX << endl;

    return 0;
}

```



## Ganzzahlige Datentypen

Für das Arbeiten mit ganzen Zahlen stehen die Datentypen `short`, `int`, `long` und `long long` zur Verfügung. Sie unterscheiden sich durch ihre Wertebereiche. In der nebenstehenden Tabelle sind die ganzzahligen Datentypen mit dem üblichen Speicherbedarf und dem entsprechenden Wertebereich zusammengestellt.

Der Datentyp `int` „integer“ ist bestmöglich an den Rechner angepasst, da seine Länge der eines Maschinenregisters entspricht. Auf 16-Bit-Rechnern ist also `int` gleichwertig mit `short`, auf 32-Bit-Rechnern ist `int` gleichwertig mit `long`.

In C++ werden Zeichencodes wie gewöhnliche Ganzzahlen behandelt. Daher kann auch mit Variablen vom Typ `char` wie mit Variablen vom Typ `int` gerechnet werden. `char` ist ein ganzzahliger Datentyp mit der Größe 1 Byte. Daraus ergibt sich ein Wertebereich von -128 bis +127 bzw. von 0 bis 255, je nachdem, ob der Compiler den Typ `char` mit oder ohne Vorzeichen interpretiert. Dies ist in C++ nicht festgelegt.

Die Zeichentypen `char16_t`, `char32_t` und `wchar_t` entsprechen jeweils einem anderen ganzzahligen Typ. So ist `wchar_t` gewöhnlich als `unsigned short` definiert.

## Die Modifizierer `signed` und `unsigned`

Die Datentypen `short`, `int` und `long` werden standardmäßig mit Vorzeichen interpretiert, wobei intern das höchstwertige Bit das Vorzeichen bestimmt. Diesen ganzzahligen Datentypen kann das Schlüsselwort `unsigned` vorangestellt werden. Die Größe des benötigten Speicherplatzes bleibt dabei erhalten. Nur der Wertebereich ändert sich, da das höchstwertige Bit nicht mehr als Vorzeichen-Bit fungiert. Das Schlüsselwort `unsigned` allein kann als Abkürzung für `unsigned int` verwendet werden.

Auch der Datentyp `char` wird normalerweise mit Vorzeichen interpretiert. Da dies aber nicht festgelegt ist, steht zusätzlich das Schlüsselwort `signed` zur Verfügung. Man unterscheidet also die drei Datentypen `char`, `signed char` und `unsigned char`.



In C++ ist die Größe der ganzzahligen Datentypen nicht festgelegt. Es gilt aber die Reihenfolge

```
char <= short <= int <= long <= long long
```

Außerdem ist der Typ `short` mindestens 2 Byte und der Typ `long` mindestens 4 Byte lang.

Die aktuellen Wertebereiche können der Header-Datei `climits` entnommen werden. In dieser Datei sind Konstanten wie `CHAR_MIN`, `CHAR_MAX`, `INT_MIN`, `INT_MAX` usw. definiert, die den kleinsten bzw. größten Wert beinhalten. Das nebenstehende Programm gibt den Wert dieser Konstanten für die Datentypen `int` und `unsigned int` aus.

## Elementare Datentypen (Fortsetzung)

### Die Datentypen für Gleitpunktzahlen

Typ	Speicherplatz	Wertebereich	kleinste positive Zahl	Genauigkeit (dezimal)
float	4 Bytes	$\pm 3.4\text{E}+38$	1.2E-38	6 Stellen
double	8 Bytes	$\pm 1.7\text{E}+308$	2.3E-308	15 Stellen
long double	10 Bytes	$\pm 1.1\text{E}+4932$	3.4E-4932	19 Stellen



Zur Darstellung der Gleitpunktzahlen wird üblicherweise das IEEE-Format (IEEE = Institute of Electrical and Electronical Engineers) verwendet. Von dieser Darstellung wird in der obigen Tabelle ausgegangen.

### Die arithmetischen Typen

#### Arithmetic types

##### Integral types

```
bool,
char, signed char, unsigned char,
wchar_t, char16_t, char32_t
short, unsigned short,
int, unsigned int,
long, unsigned long, long long, unsigned long long
```

##### Floating point types

```
float,
double
long double
```



Für die arithmetischen Typen sind die arithmetischen Operatoren definiert, das heißt, mit Variablen dieses Typs kann gerechnet werden.

## Typen für Gleitpunktzahlen

Zahlen mit gebrochenem Anteil werden in C++ mit einem Dezimalpunkt geschrieben und daher Gleitpunktzahlen genannt. Im Unterschied zu den ganzen Zahlen können Gleitpunktzahlen nur mit einer bestimmten Genauigkeit gespeichert werden. Zum Arbeiten mit Gleitpunktzahlen stehen die drei folgenden Typen zur Verfügung:

<code>float</code>	für einfache Genauigkeit
<code>double</code>	für doppelte Genauigkeit
<code>long double</code>	für sehr hohe Genauigkeit

Der Wertebereich und die Genauigkeit dieser Datentypen ergeben sich aus der Größe des Speicherplatzes und aus der internen Darstellung.

Die Genauigkeit wird in „Anzahl der Dezimalziffern“ angegeben. So bedeutet „6 Stellen genau“, dass zwei Gleitpunktzahlen, die sich innerhalb der ersten sechs Dezimalziffern unterscheiden, auch verschieden gespeichert werden können. Umgekehrt ist beispielsweise bei einer Genauigkeit von sechs Stellen nicht garantiert, dass die Zahlen 12.3456 und 12.34561 unterscheidbar sind. Dabei kommt es nur auf die Ziffernfolge an, nicht aber auf die Position des Punktes.

Sollten maschinenabhängige Größen der Gleitpunktdarstellung in einem Programm wichtig sein, so können die entsprechenden Werte der Header-Datei `cfloat` entnommen werden.

Für den interessierten Leser ist im Anhang dieses Buches ein Abschnitt enthalten, der die binäre Zahlendarstellung im Rechner beschreibt, und zwar sowohl für ganze Zahlen als auch für Gleitpunktzahlen.

## Der Operator `sizeof`

Der benötigte Speicherplatz für Objekte eines bestimmten Datentyps kann mit dem `sizeof`-Operator ermittelt werden:

```
sizeof(name)
```

gibt die Größe eines Objekts mit der Anzahl der Bytes an, wobei `name` den Typ oder das Objekt bezeichnet. Beispielsweise hat `sizeof(int)` je nach Rechner den Wert 2 oder 4. Dagegen ist `sizeof(float)` stets 4.

## Klassifizierung

Die elementaren Datentypen in C++ bestehen aus den ganzzahligen Typen (engl. *integral types*), den Gleitpunkttypen (engl. *floating point types*) und dem Typ `void`. Die Typen für Ganzzahlen und Gleitpunktzahlen werden zusammen auch als arithmetische Typen (engl. *arithmetic types*) bezeichnet, da für diese Datentypen die arithmetischen Operatoren definiert sind.

Der Typ `void` wird unter anderem für Ausdrücke benötigt, die keinen Wert darstellen. So kann z.B. ein Funktionsaufruf den Typ `void` besitzen.

## Konstanten

### Beispiele für ganzzahlige Konstanten

In jeder Zeile dieser Tabelle ist derselbe Wert verschieden dargestellt.

Dezimal	oktal	hexadezimal	Datentyp
16	020	0x10	int
255	0377	0Xff	int
32767	077777	0x7FFF	int
32768U	0100000U	0x8000U	unsigned int
100000	0303240	0x186A0	int (auf 32/64-Bit-) long (auf 16-Bit- Rechnern)
10L	012L	0xAL	long
27UL	033ul	0x1bUL	unsigned long
2147483648	020000000000	0x80000000	unsigned long



1. In C++ kann das Hochkomma ' als Trenner von Ziffern verwendet werden. Beispielsweise ist 1 Million als 1'000'000 darstellbar.
2. Mit dem Präfix 0b oder 0B kann in C++ eine Zahl auch binär (Basis 2) geschrieben werden. Beispielsweise stellt 0B101 die Zahl 5 dar.

### Beispielprogramm

```
// Hexadezimale Zahl dezimal ausgeben und umgekehrt.
//
#include <iostream>
using namespace std;

int main()
{
    // cout gibt ganze Zahlen standardmäßig dezimal aus:
    cout << "Wert von 0xFF = " << 0xFF << " dezimal"
        << endl;                // Ausgabe: 255 dezimal
    // Der Manipulator hex stellt die Ausgabe auf
    // hexadezimal um (dec wieder auf dezimal):
    cout << "Wert von 27 = " << hex << 27 << " hexadezimal"
        << endl;                // Ausgabe: 1b hexadezimal

    return 0;
}
```

Eine Konstante, auch *Literal* genannt, ist eines der booleschen Schlüsselworte `true` oder `false`, eine Zahl, ein Zeichen oder eine Zeichenkette. Zeichenketten heißen auch Strings. Entsprechend unterscheidet man:

- *boolesche Konstanten*
- *numerische Konstanten*
- *Zeichenkonstanten*
- *String-Konstanten*

Jede Konstante repräsentiert einen Wert und besitzt deshalb – wie jeder Ausdruck in C++ – einen Typ. Um welchen Typ es sich dabei handelt, ergibt sich aus der Schreibweise der Konstanten.

## Boolesche Konstanten

Für die beiden möglichen Werte eines booleschen Ausdrucks sind die Schlüsselworte `true` („wahr“) und `false` („falsch“) definiert. Beide Konstanten haben den Datentyp `bool`. Sie können z.B. benutzt werden, um Flags (= Schalter) zu setzen, die nur zwei Zustände darstellen.

## Ganzzahlige Konstanten

*Ganzzahlige numerische Konstanten* können als gewöhnliche Dezimalzahlen, Oktalzahlen oder Hexadezimalzahlen dargestellt werden:

- Eine *dezimale Konstante* (Basis 10) beginnt mit einer von 0 verschiedenen, dezimalen Ziffer, z.B. 109 oder 987650.
- Eine *oktale Konstante* (Basis 8) beginnt mit einer führenden 0, z.B. 077 oder 01234567.
- Eine *hexadezimale Konstante* (Basis 16) beginnt mit den beiden Zeichen 0x oder 0X, z.B. 0x2A0 oder 0X4b1C. Die hexadezimalen Ziffern können groß- oder kleingeschrieben werden.

Ganzzahlige Konstanten sind in der Regel vom Typ `int`. Nur wenn der Wert der Konstanten für den Typ `int` zu groß ist, bekommt sie den nächst „größeren“ Datentyp, der den Wert repräsentieren kann. Für dezimale Konstanten gilt die Abstufung:

`int`, `long`, `unsigned long`, `long long`, `unsigned long long`

Der Datentyp einer Konstanten kann auch direkt durch Anhängen der Buchstaben `L` oder `l` (für `long`), `LL` oder `ll` (für `long long`) und `U` oder `u` (für `unsigned`) festgelegt werden. So besitzen beispielsweise

`12L`      und   `12l`   den Typ `long`  
`12LL`    und   `12ll`   den Typ `long long`  
`12U`      und   `12u`   den Typ `unsigned int`  
`12UL`    und   `12ul`   den Typ `unsigned long`

## Konstanten (Fortsetzung)

### Beispiele für Gleitpunktkonstanten

In jeder Spalte der folgenden Tabelle ist derselbe Wert verschieden dargestellt.

5.19	12.	0.75	0.00004
0.519E1	12.0	.75	0.4e-4
0.0519e2	.12E+2	7.5e-1	.4E-4
519.0E-2	12e0	75E-2	4E-5

Der C++-Standard erlaubt das Trennzeichen ' auch in Gleitpunktkonstanten. So kann die Zahl 0.00004 auch als 0.000'04 oder 0.00'00'4 geschrieben werden.

### Beispiele für Zeichenkonstanten

Konstante	Zeichen	Wert der Konstanten (ASCII-Code dezimal)
'A'	Großbuchstabe A	65
'a'	Kleinbuchstabe a	97
' '	Leerzeichen	32
'.'	Punkt	46
'0'	Ziffer 0	48
'\0'	Stringende-Zeichen	0

### Interne Darstellung einer String-Konstanten

String-Konstante: "Hallo!"

Im Speicher abgelegt  
als Byte-Folge:

'H'	'a'	'l'	'l'	'o'	'!'	'\0'
-----	-----	-----	-----	-----	-----	------

## Gleitpunktkonstanten

Gleitpunktzahlen werden immer dezimal dargestellt, wobei der gebrochene Anteil durch einen Punkt vom ganzzahligen Anteil getrennt ist. Es ist auch die exponentielle Schreibweise möglich.

```
Beispiele: 27.1      1.8E-2      // Typ: double
```

Hierbei stellt 1.8E-2 den Wert  $1.8 \cdot 10^{-2}$  dar. Für E darf auch e geschrieben werden. Zur Unterscheidung von Gleitpunktzahlen und ganzzahligen Konstanten ist der Dezimalpunkt oder die Angabe von E bzw. e stets notwendig.

Jede Gleitpunktkonstante besitzt den Typ `double`. Der Typ kann aber durch Anhängen von `F` oder `f` als `float`, durch Anhängen von `L` oder `l` als `long double` festgelegt werden.

## Zeichenkonstanten

Eine Zeichenkonstante ist ein Zeichen, das in *einfachen* Anführungszeichen eingeschlossen ist. Zeichenkonstanten haben den Typ `char`.

```
Beispiel:    'A'                                // Typ: char
```

Der numerische Wert ist der Zeichencode, durch den das Zeichen repräsentiert wird. Im ASCII-Code besitzt beispielsweise die Konstante 'A' den Wert 65.

## String-Konstanten

Im Zusammenhang mit dem Stream `cout` haben Sie bereits String-Konstanten kennengelernt. String-Konstanten bestehen aus einer Folge von Zeichen, die in *doppelten* Anführungszeichen eingeschlossen sind.

**Beispiel:** "Heute ist ein schöner Tag!"

Intern wird die Zeichenfolge ohne Anführungszeichen, aber mit einem *String-ende-Zeichen* `\0` gespeichert. Dabei steht `\0` für ein Byte mit dem numerischen Wert 0, das heißt, alle Bits sind im Byte auf 0 gesetzt. Ein String benötigt deshalb ein Byte mehr Speicherplatz als die Anzahl der Zeichen, die er enthält. Der *leere String* `" "` belegt genau ein Byte.

Das Stringende-Zeichen `\0` ist von der Ziffer `0` zu unterscheiden, die einen von null verschiedenen Zeichencode besitzt. So besteht der String

**Beispiel:** " 0 "

aus zwei Bytes: Das erste Byte enthält den Code für das Zeichen 0 (im ASCII-Code: 48) und das zweite Byte den Wert 0.

Das Stringende-Zeichen `\0` ist ein Beispiel für eine Escape-Sequenz. Escape-Sequenzen werden im Folgenden besprochen.

## Escape-Sequenzen

Einzelzeichen	Bedeutung	ASCII-Wert (dezimal)
\a	alert (BEL)	7
\b	backspace (BS)	8
\t	horizontal tab (HT)	9
\n	line feed (LF)	10
\v	vertical tab (VT)	11
\f	form feed (FF)	12
\r	carriage return (CR)	13
\"	"	34
\'	'	39
\?	?	63
\\	\	92
\0	Stringende-Zeichen	0
\ooo (bis zu drei Oktalziffern)	numerischer Wert eines Zeichens	ooo (oktal!)
\xhh (Folge von Hex-Ziffern)	numerischer Wert eines Zeichens	hh (hexadezimal!)

### Beispielprogramm

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

int main()
{
    cout << "\nDies ist\t ein String\n\t\t"
          " mit \"vielen\" Escape-Sequenzen!\n";
    return 0;
}
```

### Ausgabe des Programms:

```
Dies ist      ein String
mit          "vielen" Escape-Sequenzen!
```



## Verwendung von Steuerzeichen und Sonderzeichen

Grafisch nicht darstellbare Zeichen können als *Escape-Sequenzen* angegeben werden, z.B. `\t` für das Tabulatorzeichen.

Die Wirkung der Steuerzeichen ist geräteabhängig. Zum Beispiel ist die Wirkung von `\t` abhängig von der Einstellung für die Tabulatorbreite. Diese ist gewöhnlich 8. Sie kann aber auch einen anderen Wert haben.

Eine Escape-Sequenz beginnt stets mit `\` (Backslash) und repräsentiert ein Einzelzeichen. Die standardisierten Escape-Sequenzen, ihre Bedeutung und ihr dezimaler Wert sind nebenstehend aufgelistet.

Mit den oktalen und hexadezimalen Escape-Sequenzen kann ein beliebiger Zeichencode erzeugt werden. So könnte z.B. im ASCII-Code das Zeichen A (dezimal 65) auch durch `\101` (drei Oktalziffern) oder `\x41` (zwei Hex-Ziffern) dargestellt werden. Üblicherweise werden Escape-Sequenzen nur für nicht druckbare Zeichen und Sonderzeichen benutzt. Beispielsweise beginnen die Steuerkommandos für den Bildschirmreiber und den Drucker mit dem Zeichen ESC (dezimal 27), das durch `\33` oder `\x1b` dargestellt wird.

Escape-Sequenzen werden in Zeichen- und in String-Konstanten verwendet.

**Beispiele:** `'\t'`                    `"\tHallo\n\tAnton!"`

Die Zeichen `'`, `"` und `\` verlieren ihre Sonderbedeutung durch einen vorangestellten Backslash, das heißt, sie sind durch `'\'`, `'\"'` bzw. `\\` darstellbar.

In Strings sollten bei oktalen Escape-Sequenzen stets drei Oktalziffern angegeben werden, z.B. `\033` statt `\33`. Dadurch wird vermieden, dass eine eventuell nachfolgende Ziffer zur Escape-Sequenz gerechnet wird. Bei einer hexadezimalen Escape-Sequenz gibt es keine maximale Anzahl von Ziffern. Die Folge von Hex-Ziffern endet mit dem ersten Zeichen, das keine Hex-Ziffer ist.

Das nebenstehende Beispielprogramm demonstriert die Verwendung von Escape-Sequenzen in einem String. Neu ist außerdem, dass der String sich über zwei Zeilen erstreckt: String-Konstanten, die nur durch Zwischenraumzeichen getrennt sind, werden zu *einem* String zusammengezogen.

Eine andere Möglichkeit, einen String in der nächsten Zeile fortzusetzen, besteht darin, als letztes Zeichen `\` (Backslash) in der Zeile einzugeben, die Return-Taste zu drücken und in der nächsten Zeile den String weiterzuschreiben.

**Beispiel:** `"Ich bin ein soooooo\  
langer String"`

Hierbei ist zu beachten, dass in der zweiten Zeile die führenden Leerzeichen auch zum String gehören. Daher ist im Allgemeinen die erste Methode vorzuziehen, nämlich den String mit `"` zu schließen und mit `"` wieder zu öffnen.

# Namen

## Schlüsselworte in C++

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

## Beispiele für Namen

### gültig:

```
a          DM          dm          VOID
_var       SetTextColor
B12        top_of_window
ein_sehr_langer_name123467890
```

### ungültig:

```
goto       586_cpu    Hans-Otto
zähler     true       US$
```

## Zulässige Namen

In einem Programm werden z.B. Variablen und Funktionen über *Namen* angesprochen. Für die Bildung von Namen, auch Bezeichner (engl. identifier) genannt, gelten folgende Regeln:

- Ein Name besteht aus einer Folge von Buchstaben, Ziffern oder Unterstrichen ( \_ ). Deutsche Umlaute und der Buchstabe ß sind nicht zulässig. Groß- und Kleinbuchstaben werden unterschieden.
- Das erste Zeichen muss ein Buchstabe oder Unterstrich sein.
- Ein Name kann beliebig lang sein. Alle Zeichen sind signifikant.
- Jedes Schlüsselwort in C++ ist reserviert und darf nicht als Name benutzt werden.

Die C++-Schlüsselworte sowie Beispiele für gültige und ungültige Namen sind nebenstehend aufgelistet.

Der C++-Compiler benutzt systemintern Namen, die mit zwei Unterstrichen oder mit einem Unterstrich und einem nachfolgenden Großbuchstaben beginnen. Um Verwechslungen mit diesen Namen zu vermeiden, sollte der Unterstrich nicht als erstes Zeichen verwendet werden.

Der Linker beachtet in der Regel nur eine begrenzte Anzahl von Zeichen, beispielsweise nur die ersten acht Zeichen eines Namens. Aus diesem Grund sollten Namen für globale Objekte, wie z.B. Funktionsnamen, so gewählt werden, dass die ersten acht Zeichen signifikant sind.

## Konventionen

In C++ ist es üblich, für die Namen von Variablen und Funktionen kleine Buchstaben zu benutzen. Mit einigen Variablennamen wird eine bestimmte Verwendung assoziiert.

<b>Beispiele:</b>	c, ch	für Zeichen
	i, j, k, l, m, n	für ganze Zahlen, besonders Indizes
	x, y, z	für Gleitpunktzahlen

Zur besseren Lesbarkeit von Programmen sollten im Allgemeinen aber längere „sprechende“ Namen gewählt werden, wie beispielsweise `start_index` oder `startIndex` für den ersten Index aus einem Bereich.

Bei Software-Projekten sind gewöhnlich bestimmte Namenskonventionen einzuhalten. Beispielsweise werden für Namen von Variablen bestimmte Präfixe vorgegeben, aus denen der Typ der Variablen hervorgeht.

# Variablen

## Beispielprogramm

```
// Definition und Verwendung von Variablen
#include <iostream>
using namespace std;

int gVar1;                // globale Variablen,
int gVar2 = 2;            // explizite Initialisierung

int main()
{
    char ch('A'); // lokale Variable mit Initialisierung
                  // oder: char ch = 'A';

    cout << "Wert von gVar1:  " << gVar1 << endl;
    cout << "Wert von gVar2:  " << gVar2 << endl;
    cout << "Zeichen in ch :  " << ch << endl;

    int summe, zahl = 3;    // lokale Variablen mit
                           // und ohne Initialisierung

    summe = zahl + 5;
    cout << "Wert von summe:  " << summe << endl;

    return 0;
}
```

## BildschirmAusgabe

```
Wert von gVar1:  0
Wert von gVar2:  2
Zeichen in ch :  A
Wert von summe:  8
```



Über `cout` können außer Strings alle Werte mit einem elementaren Datentyp ausgegeben werden. Die Ausgabe von ganzen Zahlen erfolgt standardmäßig dezimal.



C++ ermöglicht die „automatische“ Typableitung: Wird eine Variable in der Definition initialisiert, so kann der Compiler den Typ der Variablen automatisch aus dem Typ des Initialisierungswertes ableiten. Statt des Typs wird dann das Schlüsselwort `auto` angegeben. Ein Beispiel:

```
auto z = 1.0; // double-Variable, da 1.0 double.
```

Daten wie Zahlen, Zeichen oder ganze Datensätze werden in *Variablen* gespeichert, damit sie in einem Programm verarbeitet werden können. Variablen heißen auch *Objekte*, insbesondere dann, wenn sie vom Typ einer Klasse sind.

## Definition von Variablen

Bevor eine Variable in einem Programm benutzt werden kann, muss sie definiert werden. Bei der *Definition*, auch *Vereinbarung* genannt, wird der Datentyp der Variablen festgelegt und der entsprechende Speicherplatz reserviert. Mit dem Namen der Variablen wird dieser Speicherplatz angesprochen. Eine einfache Definition hat folgende

**Syntax:**     `typ name1 [, name2 ... ];`

Dadurch werden die in der Liste `name1 [, name2 ...]` angegebenen Namen als Variablen mit dem Datentyp `typ` vereinbart. In einer Syntaxbeschreibung bedeuten die Klammern `[ ... ]`, dass dieser Teil optional ist, d.h. fehlen kann. In einer Definition können also eine oder mehrere Variablen angegeben werden.

**Beispiele:**   `char c;`  
                   `int i, zaehler;`  
                   `double x, y, size;`

Variablen dürfen in einem Programm außerhalb oder innerhalb von Funktionen definiert werden. Der Unterschied ist folgender:

- Eine außerhalb jeder Funktion definierte Variable ist *global*, d.h. in allen Funktionen verwendbar.
- Eine innerhalb einer Funktion definierte Variable ist *lokal*, d.h. nur in derselben Funktion verwendbar.

Lokale Variablen werden normalerweise unmittelbar hinter einer sich öffnenden geschweiften Klammer – etwa am Beginn einer Funktion – definiert. Sie können aber überall da stehen, wo eine Anweisung erlaubt ist. So lassen sich Variablen an der Stelle vereinbaren, an der sie direkt verwendet werden.

## Initialisierung

Bei der Definition kann eine Variable initialisiert werden, d.h. einen Anfangswert erhalten. Die Initialisierung erfolgt, indem hinter dem Variablennamen

- ein Gleichheitszeichen (`=`) und ein Anfangswert angegeben werden oder
- der Anfangswert in runde oder geschweifte Klammern eingeschlossen wird.

**Beispiele:**   `char c = 'a';`  
                   `double x(1.75), y{-2.5};`

Nicht explizit initialisierte *globale* Variablen werden mit 0 vorbelegt. Dagegen haben nicht initialisierte *lokale* Variablen einen undefinierten Anfangswert.

# Die Schlüsselworte const und volatile

## Beispielprogramm

```
// Umfang und Fläche eines Kreises mit Radius 1.5

#include <iostream>
using namespace std;

const double pi = 3.141593;

int main()
{
    double flaeche, umfang, radius = 1.5;

    flaeche = pi * radius * radius;
    umfang = 2 * pi * radius;

    cout << "\nKreisberechnung\n" << endl;

    cout << "Radius:      " << radius    << endl
         << "Umfang:      " << umfang    << endl
         << "Flaeche:     " << flaeche   << endl;

    return 0;
}
```



Eine Gleitpunktzahl wird über cout standardmäßig mit maximal sechs Ziffern ohne abschließende 0-Ziffern ausgegeben.

## BildschirmAusgabe

Kreisberechnung

```
Radius:      1.5
Umfang:      9.42478
Flaeche:     7.06858
```

Ein Datentyp kann mit den Schlüsselworten `const` und `volatile` modifiziert werden.

## Konstante Objekte

Das Schlüsselwort `const` wird verwendet, um ein „Read-only“-Objekt anzulegen. Da ein solches Objekt konstant ist, also nachträglich nicht mehr geändert werden kann, muss es bei der Definition initialisiert werden.

**Beispiel:** `const double pi = 3.1415947;`

Damit kann der Wert von `pi` durch das Programm nicht mehr verändert werden. Auch versehentlich gesetzte Anweisungen wie

```
pi = pi + 2.0;                                // unzulässig
```

erzeugen eine Fehlermeldung.

## Volatile-Objekte

Mit dem selten benutzten Schlüsselwort `volatile` werden Variablen definiert, die nicht nur durch das Programm selbst, sondern auch durch andere Programme und Ereignisse von außerhalb veränderbar sind. Solche Ereignisse können etwa durch Interrupts einer Hardware-Uhr hervorgerufen werden.

**Beispiel:** `volatile unsigned long clock_ticks;`

Auch wenn das Programm selbst die Variable nicht verändert, muss der Compiler davon ausgehen, dass sich der Wert der Variablen seit dem letzten Zugriff verändert haben kann. Der Compiler erzeugt daher einen Maschinencode, der bei jedem lesenden Zugriff auf die Variable den Wert erneut einliest (und nicht mit dem zuvor gelesenen Wert weiterarbeitet.)

Es ist auch möglich, in einer Variablendefinition die Schlüsselworte `const` und `volatile` zu kombinieren.

**Beispiel:** `volatile const unsigned time_to_live;`

Nach dieser Definition kann die Variable `time_to_live` zwar nicht durch das Programm, jedoch durch ein Ereignis von außerhalb verändert werden.

# Übungen

## Bildschirmausgabe zur 2. Aufgabe

```
ICH
    "SAUSE"
        \HIN\
    UND
/HER/
```

## Zur 3. Aufgabe

Definition und Initialisierung von Variablen:

<code>int a(2.5);</code>	<code>const long breite;</code>
<code>int b = '?';</code>	<code>char c('\');</code>
<code>char z(500);</code>	<code>unsigned char ch = '\201';</code>
<code>int big = 40000;</code>	<code>unsigned size(40000);</code>
<code>double fläche(1.2E+5);</code>	<code>float val = 12345.12345;</code>



### *1. Aufgabe*

Mit dem `sizeof`-Operator kann die Anzahl der Bytes ermittelt werden, die Variablen eines bestimmten Datentyps im Hauptspeicher belegen. Beispielsweise hat `sizeof(short)` den Wert 2.

Schreiben Sie ein C++-Programm, das für jeden elementaren Datentyp die Größe des benötigten Speicherplatzes auf dem Bildschirm ausgibt.

### *2. Aufgabe*

Schreiben Sie ein C++-Programm, das die nebenstehende Bildschirmausgabe erzeugt.

### *3. Aufgabe*

Welche der nebenstehenden Variablendefinitionen ist nicht zulässig oder nicht sinnvoll?

### *4. Aufgabe*

Erstellen Sie ein C++-Programm, in dem zwei Variablen für Gleitpunktzahlen definiert und mit den Werten

123.456 und 76.543

initialisiert werden. Anschließend ist die Summe und die Differenz der beiden Zahlen am Bildschirm auszugeben.

# Lösungen

## Zur 1. Aufgabe:

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

int main()
{
    cout << "\nGroesse der elementaren Datentypen\n"
          << "    Typ                Anzahl Bytes\n"
          << "-----" << endl;
    cout << "    char:                " << sizeof(char) << endl;
    cout << "    short:               " << sizeof(short) << endl;
    cout << "    int:                 " << sizeof(int) << endl;
    cout << "    long:                " << sizeof(long) << endl;
    cout << "    long long:           " << sizeof(long long) << endl;
    cout << "    float:               " << sizeof(float) << endl;
    cout << "    double:              " << sizeof(double) << endl;
    cout << "    long double:         " << sizeof(long double)
          << endl;

    return 0;
}
```

## Zur 2. Aufgabe:

```
// Benutzung von Escape-Sequenzen

#include <iostream>
using namespace std;

int main()
{
    cout << "\n\n\t ICH"                // Statt der Tabs
          << "\n\n\t\t\t \"SAUSE\""          // können auch
          << "\n\n\t\t\t\t\t \\HIN\\"          // entsprechend
          << "\n\n\t\t\t UND"                // viele Blanks
          << "\n\n\t /HER/" << endl;        // ausgegeben
                                          // werden.

    return 0;
}
```

## Lösungen (Fortsetzung)

### Zur 3. Aufgabe:

Unzulässig oder nicht sinnvoll sind:

```
int a(2.5);           // 2.5 ist keine Ganzzahl.
const long breite;    // Initialisierung fehlt.
char z = 500;         // Der Wert 500 ist zu groß
                      // für ein Byte.
int big(40000);       // Vorsicht! Auf 16-Bit-Systemen
                      // sind int-Werte <= 32767.
double fläche(1.2E+5); // Das Zeichen ä ist in
                      // Namen nicht erlaubt.
float val = 12345.12345; // Die Genauigkeit von float
                      // beträgt nur sechs Stellen.
```

### Zur 4. Aufgabe:

```
// Definition und Initialisierung von Variablen

#include <iostream>
using namespace std;

int main()
{
    float x = 123.456F,           // oder double
          y = 76.543F,
          summe;

    summe = x + y;

    cout << "Summe:      "
          << x << " + " << y << " = " << summe << endl;

    cout << "Differenz:  "
          << x << " - " << y << " = " << (x - y) << endl;

    return 0;
}
```



# Kapitel 3

## Verwenden von Funktionen und Klassen

Dieses Kapitel beschreibt, wie

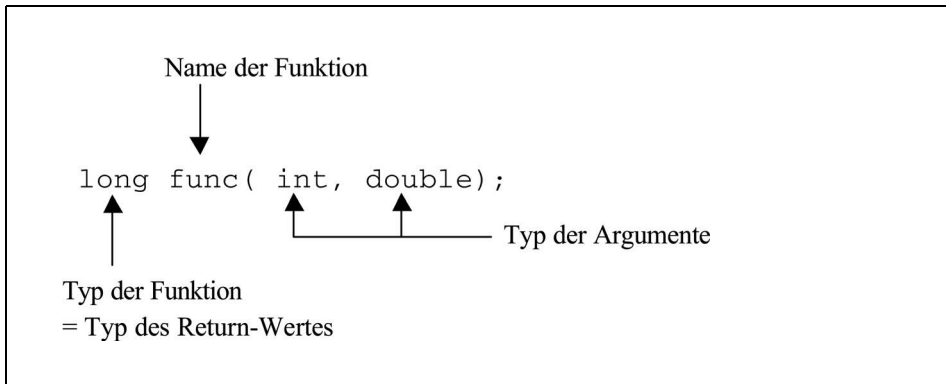
- Standardfunktionen deklariert und aufgerufen sowie
- Standardklassen eingesetzt

werden können. Hierbei wird auch die Verwendung der Standard-Header-Dateien beschrieben. Außerdem wird zum erstenmal mit String-Variablen gearbeitet, d.h. mit Objekten der Standardklasse `string`.

Die Definition eigener Funktionen und Klassen wird in einem späteren Kapitel vorgestellt.

# Deklaration von Funktionen

## Beispiel für den Prototyp einer Funktion



Der vorstehende Prototyp liefert dem Compiler folgende Informationen:

- `func` ist der Name einer Funktion.
- Die Funktion wird mit zwei Argumenten aufgerufen:  
Das erste Argument besitzt den Typ `int`, das zweite den Typ `double`.
- Der Return-Wert der Funktion ist vom Typ `long`.

## Prototypen mathematischer Standardfunktionen

```
double sin( double);           // Sinus
double cos( double);          // Cosinus
double tan( double);          // Tangens
double atan( double);         // Arcustangens
double atan2( double, double); // Arcustangens
double sqrt( double);         // Wurzel
double pow( double, double);   // Potenzieren
double exp( double);          // Exponentialfunktion
double ldexp( double, double); // Exponentialfunktion
double log( double);          // Logarithmus
double log10( double);        // Logarithmus
```

## Deklarationen

Alle Namen (= Bezeichner, engl. *identifier*), die in einem Programm vorkommen, müssen dem Compiler bekannt sein. Andernfalls erzeugt er eine Fehlermeldung. Das bedeutet, dass alle Namen, die keine Schlüsselworte sind, vor ihrer Verwendung *deklariert*, d.h. dem Compiler bekannt gemacht werden müssen.

Jede Definition einer Variablen oder einer Funktion stellt auch eine Deklaration dar. Umgekehrt ist aber nicht jede Deklaration auch eine Definition. Will man beispielsweise eine schon vorhandene Funktion aus einer Bibliothek verwenden, so muss die Funktion zwar deklariert, aber nicht neu definiert werden.

## Deklaration von Funktionen

Eine Funktion besitzt wie eine Variable einen Namen und einen Datentyp. Der Typ der Funktion ist der Typ des *Return-Wertes*, also des Wertes, den die Funktion zurückgibt. Außerdem ist wichtig, welchen Typ die Argumente der Funktion haben müssen. In einer *Funktionsdeklaration* wird daher dem Compiler

- der Name und Typ der Funktion und
- der Datentyp jedes Arguments

bekannt gemacht. Man nennt dies auch den *Prototyp* einer Funktion.

**Beispiele:** `int toupper(int);`  
`double pow(double, double);`

Damit erhält der Compiler die Information, dass die Funktion `toupper()` vom Typ `int` ist, also einen Return-Wert vom Typ `int` liefert, und ein Argument vom Typ `int` erwartet. Die zweite Funktion `pow()` ist vom Typ `double` und wird mit zwei Argumenten vom Typ `double` aufgerufen. Den Datentypen der Argumente dürfen Namen folgen, die jedoch nur die Bedeutung eines Kommentars haben.

**Beispiele:** `int toupper(int zeichen);`  
`double pow(double base, double exponent);`

Für den Compiler sind diese Prototypen gleichbedeutend mit denen im vorangegangenen Beispiel. Beide Funktionen sind Standardfunktionen.

Die Prototypen der Standardfunktionen müssen nicht – und sollten auch nicht – selbst angegeben werden. Sie sind bereits in Standard-Header-Dateien enthalten. Wird die entsprechende Header-Datei mit der `#include`-Direktive in eine Quelldatei kopiert, so kann die Funktion sofort verwendet werden.

**Beispiel:** `#include <cmath>`

Anschließend stehen in der Quelldatei alle mathematischen Standardfunktionen wie `sin()`, `cos()`, `pow()` usw. zur Verfügung. Weitere Details zu Header-Dateien finden Sie später in diesem Kapitel.

# Aufruf von Funktionen

## Beispielprogramm

```
// Berechnung von Potenzwerten
// mit der Standardfunktion pow()

#include <iostream>          // Deklaration von cout
#include <cmath>             // Prototyp von pow(), also
                           // double pow( double, double);

using namespace std;

int main()
{
    double x = 2.5, y;

    // Anhand des Prototyps erzeugt der Compiler den
    // richtigen Aufruf bzw. eine Fehlermeldung!

    // Berechnung von x hoch 3:
    y = pow("x", 3.0);      // Fehler! String ist keine Zahl.
    y = pow(x + 3.0);       // Fehler! Nur ein Argument
    y = pow(x, 3.0);        // ok! Return-Wert an y zuweisen.
    y = pow(x, 3);          // Auch ok! Compiler wandelt den
                           // int-Wert 3 in double um.

    cout << "2.5 hoch 3 ergibt:      "
         << y << endl;

    // Mit der Funktion pow() kann gerechnet werden:
    cout << "2 + (5 hoch 2.5) ergibt: "
         << 2.0 + pow(5.0, x) << endl;

    return 0;
}
```

## BildschirmAusgabe

```
2.5 hoch 3 ergibt:      15.625
2 + (5 hoch 2.5) ergibt: 57.9017
```



## Funktionsaufruf

Ein *Funktionsaufruf* ist ein Ausdruck vom Typ der Funktion, dessen Wert der Return-Wert ist. Häufig wird der Return-Wert nur an eine passende Variable zugewiesen.

**Beispiel:** `y = pow( x, 3.0 );`

Hier wird zunächst die Funktion `pow()` mit den Argumenten `x` und `3.0` aufgerufen und das Ergebnis, also die Potenz  $x^3$ , an `y` zugewiesen.

Da der Funktionsaufruf einen Wert darstellt, sind auch andere Operationen möglich. So kann z.B. mit dem Funktionsaufruf von `pow()` wie mit einem `double`-Wert gerechnet werden.

**Beispiel:** `cout << 2.0 + pow( 5.0, x );`

In dieser Anweisung wird zuerst die Zahl `2.0` zum Return-Wert von `pow(5.0, x)` hinzuaddiert und dann das Ergebnis mit `cout` ausgegeben.

Als *Argument* darf einer Funktion ein beliebiger Ausdruck übergeben werden, beispielsweise auch eine Konstante oder ein arithmetischer Ausdruck. Wichtig ist, dass die Datentypen der Argumente mit denen übereinstimmen, die die Funktion erwartet.

Anhand des Prototyps überprüft der Compiler den korrekten Aufruf der Funktion. Stimmt der Typ eines Arguments nicht exakt mit dem entsprechenden Typ im Prototyp überein, so nimmt der Compiler eine Typkonvertierung vor, sofern diese möglich ist.

**Beispiel:** `y = pow( x, 3 );` // auch ok

Als zweites Argument wird der Wert 3 vom Typ `int` übergeben. Da die Funktion aber einen `double`-Wert erwartet, nimmt der Compiler die Konvertierung von `int` nach `double` vor.

Wird eine Funktion mit einer falschen Anzahl von Argumenten aufgerufen oder ist die Konvertierung eines Arguments nicht möglich, so erzeugt der Compiler eine Fehlermeldung. Dadurch können Fehler beim Funktionsaufruf schon während der Erstellung erkannt und beseitigt werden und führen nicht zu Laufzeitfehlern.

**Beispiel:** `float x = pow(3.0 + 4.7);` // Fehler!

Der Compiler wird hier die falsche Anzahl Argumente melden. Außerdem gibt der Compiler eine Warnung aus, da ein `double`, nämlich der Return-Wert von `pow()`, an eine `float`-Variable zugewiesen wird.

# Der Typ void für Funktionen

## Beispielprogramm

```
// Es werden drei Zufallszahlen ausgegeben

#include <iostream> // Deklaration von cin und cout
#include <cstdlib>  // Prototypen von srand(), rand():
                  // void srand( unsigned int seed );
                  // int rand( void );

using namespace std;
int main()
{
    unsigned int keim;
    int z1, z2, z3;

    cout << "    --- Zufallszahlen --- \n" << endl;
    cout << "Initialisierung des Zufallszahlengenerators\n"
         << "Geben Sie eine ganze Zahl ein: ";
    cin  >> keim;          // Eine ganze Zahl einlesen

    srand( keim);          // und damit den Zufallszahlen-
                          // generator initialisieren.

    z1 = rand();           // Drei Zufallszahlen erzeugen.
    z2 = rand();
    z3 = rand();

    cout << "\nDrei Zufallszahlen: "
         << z1 << "    " << z2 << "    " << z3 << endl;

    return 0;
}
```



Mit der Anweisung `cin >> keim;` wird von der Tastatur eine ganze Zahl eingelesen, da `keim` vom Typ `unsigned int` ist.

## Beispiel für eine Bildschirmausgabe

```
--- Zufallszahlen ---

Initialisierung des Zufallszahlengenerators
Geben Sie eine ganze Zahl ein: 7777

Drei Zufallszahlen: 25435    6908    14579
```

## Funktionen ohne Return-Wert

Es können auch Funktionen geschrieben werden, die eine bestimmte Aktion ausführen, aber keinen Wert an die aufrufende Funktion zurückgeben. Für derartige Funktionen, die in anderen Programmiersprachen auch Prozeduren heißen, gibt es den Typ `void`.

**Beispiel:** `void srand( unsigned int seed );`

Die Standardfunktion `srand()` initialisiert einen Algorithmus zur Erzeugung von „Zufallszahlen“. Da die Funktion keinen Wert zurückgibt, hat sie den Typ `void`. Die Funktion erhält als Argument einen `unsigned`-Wert, den „Keim“ (engl. *seed*) des Zufallszahlengenerators. Dieser wird als Parameter benutzt, um eine neue Folge von Zufallszahlen zu generieren.

## Funktionen ohne Argumente

Erwartet eine Funktion kein Argument, so wird dies im Prototyp mit `void` deklariert oder die Klammern bleiben leer.

**Beispiel:** `int rand( void );` // oder: `int rand();`

Die Standardfunktion `rand()` wird ohne Argument aufgerufen und liefert eine Zufallszahl zwischen 0 und `RAND_MAX`. Die ganzzahlige Konstante `RAND_MAX` hat mindestens den Wert 32767. Durch wiederholten Aufruf der Funktion erhält man eine Folge von Zufallszahlen.

## Zur Verwendung von `srand()` und `rand()`

Die Prototypen der Funktionen `srand()` und `rand()` befinden sich sowohl in der Header-Datei `cstdlib`, als auch in `stdlib.h`.

Aufrufe der Funktion `rand()` ohne einen vorhergehenden Aufruf von `srand()` erzeugen dieselbe Zahlenfolge, als ob zuvor die Anweisung

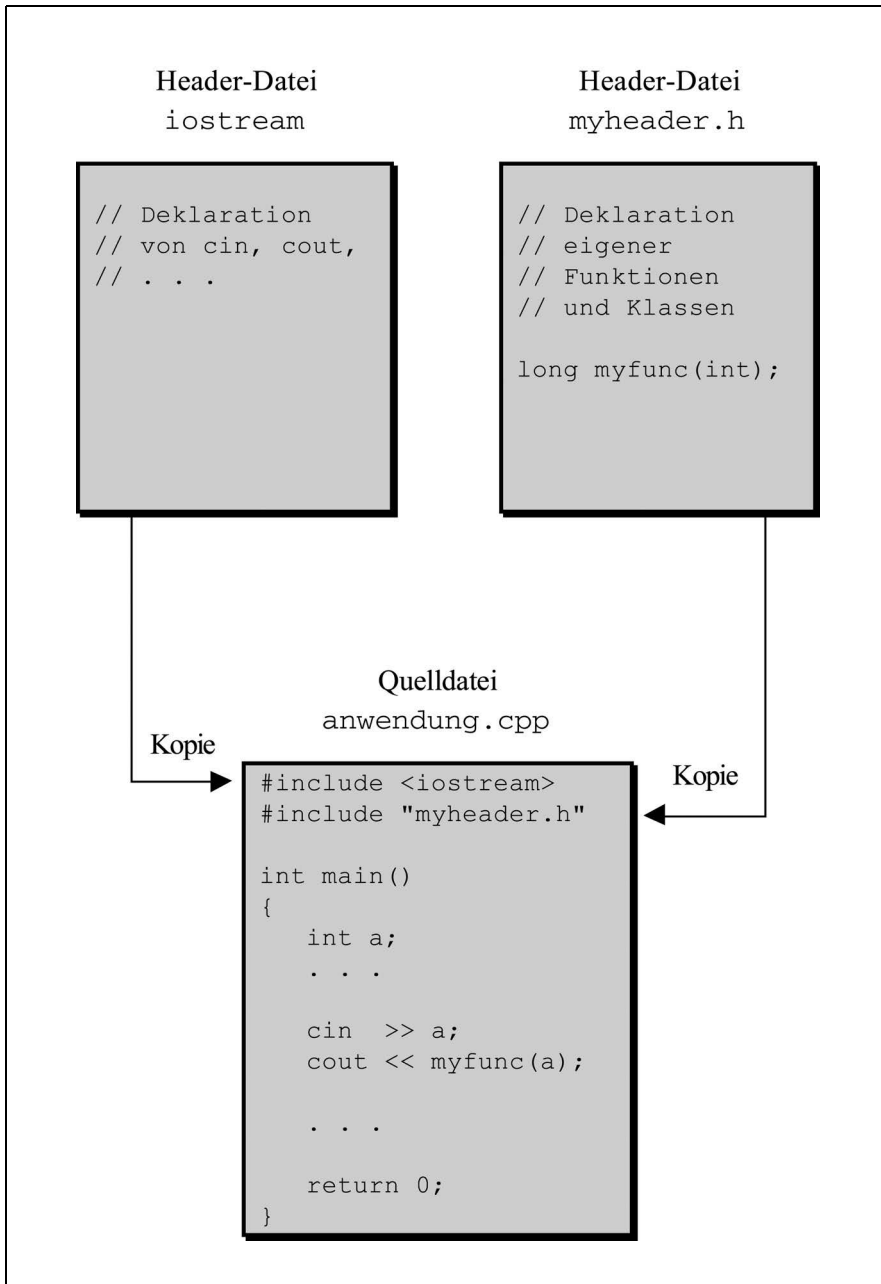
```
srand(1);
```

ausgeführt worden wäre. Soll nicht bei jedem Programmstart dieselbe Folge von „Zufallszahlen“ erzeugt werden, muss die oben beschriebene Funktion `srand()` bei jedem Programmstart mit einem anderen Argumentwert aufgerufen werden.

Üblicherweise wird die aktuelle Zeit zur Initialisierung des Zufallszahlengenerators verwendet. Ein entsprechendes Beispiel finden Sie bei den Übungen des 6. Kapitels.

# Header-Dateien

## Verwendung von Header-Dateien



## Arbeiten mit Header-Dateien

Header-Dateien sind Textdateien, die Deklarationen und Makros enthalten. Nach einer entsprechenden `#include`-Direktive stehen diese dann in jeder anderen Quelldatei, auch in einer anderen Header-Datei, zur Verfügung.

Bei der Verwendung von Header-Dateien ist Folgendes zu beachten:

- Header-Dateien sollten generell am Anfang eines Programms inkludiert werden, noch vor jeder anderen Deklaration.
- In einer `#include`-Direktive darf nur *eine* Header-Datei angegeben werden.
- Der Dateiname muss in spitzen Klammern `< ... >` oder in doppelten Hochkommas `" ... "` eingeschlossen sein.

## Suche nach Header-Dateien

Die mit der Compilersoftware ausgelieferten Header-Dateien sind gewöhnlich in einem eigenen Verzeichnis, meist mit dem Namen `include`, abgelegt. Ist der Name einer Header-Datei innerhalb von spitzen Klammern `< ... >` angegeben, so wird üblicherweise nur dieses Verzeichnis nach der Header-Datei durchsucht, aber nicht das aktuelle Verzeichnis. Das beschleunigt die Suche nach den Header-Dateien.

Bei der Entwicklung eines C++-Programms werden gewöhnlich auch eigene Header-Dateien erstellt, und zwar im aktuellen Projekt-Verzeichnis. Damit der Compiler auch diese Dateien findet, muss in der `#include`-Direktive der Name der Header-Datei in doppelte Anführungszeichen eingeschlossen werden.

**Beispiel:** `#include "project.h"`

Der Compiler durchsucht dann zusätzlich das aktuelle Directory. Für selbst erstellte Header-Dateien wird in der Regel die Endung `.h` verwendet.

## Definition der Standardklassen

Wie die Prototypen der Standardfunktionen sind auch die Definitionen der Standardklassen in Header-Dateien enthalten. Wird eine Header-Datei inkludiert, stehen die darin definierten Klassen und die evtl. deklarierten Objekte im Programm zur Verfügung.

**Beispiel:** `#include <iostream>`  
`using namespace std;`

Nach dieser Anweisung kann mit den Klassen `istream` und `ostream` sowie mit den schon definierten Streams `cin` und `cout` gearbeitet werden. Dabei ist `cin` ein Objekt der Klasse `istream`, `cout` ein Objekt der Klasse `ostream`.

## Standard-Header-Dateien

### Die Header-Dateien der C++-Standardbibliothek

algorithm	future	numeric	sstream
any(*)	initializer_list	optional(*)	system_error
array	iomanip	ostream	thread
atomic	ios	queue	tuple
bitset	iosfwd	random	typeindex
chrono	iostream	ratio	typeinfo
codecvt	istream	regex	type_traits
complex	iterator	scoped_allocator(*)	unordered_map
condition_variable	limits	set	unordered_set
dequeue	list	shared_mutex(*)	utility
exception	locale	sstream	valarray
execution(*)	map	stack	variant(*)
filesystem(*)	memory	stdexcept	vector
forward_list	memory_resource(*)	streambuf	
fstream	mutex	string	
functional	new	string_view(*)	

*Hinweis:* Die mit (\*) gekennzeichneten Header-Dateien wurden mit C++17 eingeführt und werden noch nicht von allen Compilern unterstützt.

### Die Header-Dateien der C-Standardbibliothek

assert.h	iso646.h	stdarg.h	tgmath.h
complex.h	limits.h	stdbool.h	time.h
ctype.h	locale.h	stddef.h	uchar.h
errno.h	math.h	stdint.h	wchar.h
fenv.h	setjmp.h	stdio.h	wctype.h
float.h	signal.h	stdlib.h	
inttypes.h	stdalign.h	string.h	

Die Header-Dateien der C++-Standardbibliothek sind nebenstehend zusammengestellt. Sie besitzen keine Kennung `.h` und nehmen sämtliche Deklarationen im eigenen Namensbereich `std` vor. Namensbereiche werden Sie in einem späteren Kapitel kennenlernen. Hier genügt es zu wissen, dass Bezeichner aus einem anderen Namensbereich nicht direkt angesprochen werden können. Nach der Direktive

**Beispiel:**    `#include <iostream>`

würde der Compiler die Streams `cin` und `cout` also nicht kennen. Damit die Bezeichner aus dem Namensbereich `std` global verfügbar sind, muss eine *using*-Direktive angegeben werden.

**Beispiel:**    `#include <iostream>`  
              `#include <string>`  
              `using namespace std;`

Anschließend können `cin` und `cout` ohne weitere Angaben wie gewohnt verwendet werden. Außerdem wurde noch die Header-Datei `string` inkludiert. Dadurch steht die Klasse `string` zur Verfügung, die in C++ zum komfortablen Umgang mit Strings definiert ist. Weitere Informationen dazu folgen auf den nächsten Seiten.

## Die Header-Dateien der Programmiersprache C

Auch die Header-Dateien der Programmiersprache C sind in den C++-Standard übernommen worden. Damit stehen einem C++-Programm alle Funktionen der C-Standardbibliothek zur Verfügung.

**Beispiel:**    `#include <math.h>`

Anschließend können die mathematischen Funktionen aufgerufen werden.

Alle in einer C-Header-Datei deklarierten Bezeichner sind global verfügbar. Das kann bei großen Programmen zu Namenskonflikten führen. Daher gibt es in C++ zu jeder C-Header-Datei `name.h` eine entsprechende Header-Datei `cname`, die dieselben Bezeichner in Namensbereich `std` deklariert. So ist beispielsweise das inkludieren der Datei `math.h` äquivalent mit:

**Beispiel:**    `#include <cmath>`  
              `using namespace std;`

Werden in einem Programm die elementaren Funktionen zur Manipulation von C-Strings aufgerufen, so muss die Datei `string.h` oder `cstring` inkludiert werden. Diese Header-Dateien stellen die Funktionalitäten der C-String-Bibliothek bereit. Sie ist unbedingt von der Header-Datei `string` zu unterscheiden, in der die Klasse `string` definiert ist.

Jeder Compiler bietet darüber hinaus weitere Header-Dateien für plattformabhängige Funktionalitäten. Hierzu gehören beispielsweise Grafikbibliotheken und Schnittstellen zur Datenbankanbindung.

# Verwenden von Standardklassen

## Beispielprogramm mit der Klasse string

```
// Mit Strings arbeiten.

#include <iostream>      // Deklaration von cin, cout
#include <string>        // Deklaration der Klasse string
using namespace std;

int main()
{
    // Vier Strings definieren:
    string prompt("Wie lautet Ihr Name: "),
           name,                // leer
           linie( 40, '-' ),    // String mit 40 '-'
           gesamt = "Hallo ";   // auch möglich!

    cout << prompt;           // Eingabe-Aufforderung
    getline( cin, name );      // Name (eine Zeile) einlesen

    gesamt = gesamt + name;     // Strings verketteten
                                // und zuweisen.

    cout << linie << endl      // Linie u. Name ausgeben
         << gesamt << endl;
    cout << "Ihr Name ist "    // Länge ausgeben
         << name.length() << " Zeichen lang!" << endl;
    cout << linie << endl;
    return 0;
}
```



Für Objekte der Klasse string sind neben den Operatoren +, += für die Verkettung auch die Vergleichsoperatoren <, <=, >, >=, ==, != definiert. Strings können mit cout und dem Operator << ausgegeben werden.

Die Klasse string wird in einem späteren Kapitel noch im Detail behandelt.

## Beispiel für eine Bildschirmausgabe

```
Wie lautet Ihr Name:  Peter Lustig
-----
Hallo Peter Lustig
Ihr Name ist 12 Zeichen lang!
-----
```



Die C++-Standardbibliothek definiert zahlreiche Klassen. Hierzu gehören beispielsweise die Stream-Klassen für die Ein-/Ausgabe, aber auch Klassen zur Darstellung von Strings oder zur Behandlung von Fehlersituationen.

Jede Klasse ist ein Datentyp mit bestimmten Eigenschaften und Fähigkeiten. Wie bereits erwähnt, werden die Eigenschaften der Klasse durch ihre *Datenelemente*, die Fähigkeiten durch *Methoden* dargestellt. Methoden sind Funktionen, die zu einer Klasse gehören und mit den Datenelementen bestimmte Operationen ausführen. Methoden werden auch *Elementfunktionen* genannt.

## Anlegen von Objekten

Ein *Objekt* ist eine Variable vom Typ einer Klasse. Man spricht auch von einer *Instanz* der Klasse. Beim Anlegen eines Objekts wird der Speicher für die Datenelemente bereitgestellt und mit geeigneten Anfangswerten versehen.

**Beispiel:** `string s("Ich bin ein String");`

Hier wird ein Objekt `s` vom Typ der Standardklasse `string` – kurz: ein *String* – definiert und mit der angegebenen String-Konstanten initialisiert. Ein Objekt der Klasse `string` verwaltet den erforderlichen Speicherplatz für die Zeichenfolge selbständig.

Es gibt im Allgemeinen verschiedene Möglichkeiten, ein Objekt einer Klasse zu initialisieren. So kann ein String auch mit einer bestimmten Anzahl eines Zeichens initialisiert werden. Dies zeigt auch das nebenstehende Beispiel.

## Aufruf von Methoden

Für ein Objekt können alle Methoden aufgerufen werden, die in der entsprechenden Klasse „öffentlich“ (engl. *public*) deklariert sind. Im Gegensatz zum Aufruf einer globalen Funktion erfolgt der Aufruf einer Methode immer für *ein bestimmtes Objekt*. Der Name des Objekts ist vor dem Namen der Methode, durch einen Punkt getrennt, anzugeben.

**Beispiel:** `s.length();`                      `// objekt.methode();`

Die Methode `length()` liefert die Länge eines Strings, d.h. die Anzahl der Zeichen in einem String. Für den oben definierten String `s` ist das der Wert 18.

## Klassen und globale Funktionen

Für einige Standardklassen gibt es auch *global definierte Funktionen*, die bestimmte Operationen für ein *übergebenes* Objekt ausführen. So liest die globale Funktion `getline()` eine Textzeile von der Tastatur in einen String.

**Beispiel:** `getline(cin, s);`

Die Textzeile wird mit der Return-Taste abgeschlossen, das entsprechende Newline-Zeichen `'\n'` gelesen, aber nicht im String gespeichert.

# Übungen

## Bildschirmausgabe zur 1. Aufgabe

ZAHL	WURZEL
4	2
12.25	3.5
0.0121	0.11

## Listing zur 2. Aufgabe

```
// Ein Programm mit Fehlern!

# include <iostream>, <string>
# include <stdlib>
# void srand( seed);

int main()
{
    string meldung "\nAus Fehlern wird man klug!";
    cout << meldung << endl;

    int len = length( meldung);
    cout << "Die Laenge des Strings: " << len << endl;

    // Und noch eine Zufallszahl:
    int a, b;
    a = srand(12.5);
    b = rand( a );
    cout << "\nZufallszahl: " << b << endl;

    return 0;
}
```

### 1. Aufgabe

Erstellen Sie ein Programm, das die Wurzeln der Zahlen

4,     12.25.     0.0121

berechnet und wie nebenstehend gezeigt ausgibt. Lassen Sie dann noch eine Zahl von der Tastatur einlesen und geben Sie ebenfalls die Wurzel dieser Zahl aus.

Zur Berechnung der Wurzel einer Zahl verwenden Sie die Funktion `sqrt()`, die in der Header-Datei `math.h` (oder auch `cmath`) mit folgendem Prototyp deklariert ist:

```
double sqrt( double x);
```

Der Return-Wert der Funktion `sqrt()` ist die Wurzel von `x`.

### 2. Aufgabe

Das nebenstehende Programm enthält mehrere **Fehler**! Korrigieren Sie die Fehler, und testen Sie die Lauffähigkeit des Programms.

### 3. Aufgabe

Erstellen Sie ein C++-Programm, in dem ein String mit der Zeichenkette

```
Schon wieder was dazugelernt!
```

definiert und die Länge des Strings auf dem Bildschirm ausgegeben wird.

Lesen Sie dann zwei Textzeilen von der Tastatur ein. Fügen Sie die beiden Strings so zu einem String zusammen, dass die beiden Teilstrings durch " \* " getrennt sind. Geben Sie den neuen String am Bildschirm aus.

# Lösungen

## Zur 1. Aufgabe:

```
// Wurzeln berechnen

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

int main()
{
    double x1 = 4.0, x2 = 12.25, x3 = 0.0121;

    cout << "\n    ZAHL    \t WURZEL" << endl;
    cout << "\n        " << x1 << "    \t " << sqrt(x1)
        << "\n        " << x2 << "    \t " << sqrt(x2)
        << "\n        " << x3 << "    \t " << sqrt(x3) << endl;

    cout << "\nVon welcher Zahl soll die Wurzel "
        << "berechnet werden? ";

    cin  >> x1;

    cout << "\n    ZAHL    \t WURZEL" << endl;
    cout << "\n        " << x1 << "    \t " << sqrt(x1) << endl;

    return 0;
}
```

## Zur 2. Aufgabe:

```
// Das korrigierte Programm:

#include <iostream>      // Jeweils nur eine Header-Datei
#include <string>

#include <cstdlib>       // Prototypen der Funktionen
                        // void srand( unsigned int seed);
                        // int rand(void);

// oder:
// #include <stdlib.h>

using namespace std;    // Namen aus dem Namensbereich std
                        // global verfügbar machen.

int main()
{
```

```
string meldung = "\nAus Fehlern wird man klug!";    // =
cout << meldung << endl;

int len = meldung.length(); // statt: length(meldung);
cout << "Die Laenge des Strings: " << len << endl;

// Und noch eine Zufallszahl:
int b; // a wird nicht gebraucht.
srand(12); // statt: a = srand(12.5);
b = rand(); // statt: b = rand(a);

cout << "\nZufallszahl: " << b << endl;

return 0;
}
```

### ***Zur 3. Aufgabe:***

```
#include <iostream>    // Deklaration von cin, cout
#include <string>       // Deklaration der Klasse string
using namespace std;

int main()
{
    string meldung("Schon wieder was dazugelernt!\n"),
           prompt("Bitte zwei Zeilen Text eingeben:");
    str1, str2, summe;

    cout << meldung << endl;    // Meldung ausgeben

    cout << "Die Laenge der Strings: " << meldung.length()
         << endl;

    cout << prompt << endl;    // Eingabe-Aufforderung

    getline( cin, str1);       // 1. Zeile einlesen
    getline( cin, str2);       // 2. Zeile einlesen

    summe = str1 + " * " + str2; // Strings verketten,
                                // zuweisen
    cout << summe << endl;    // und ausgeben.

    return 0;
}
```



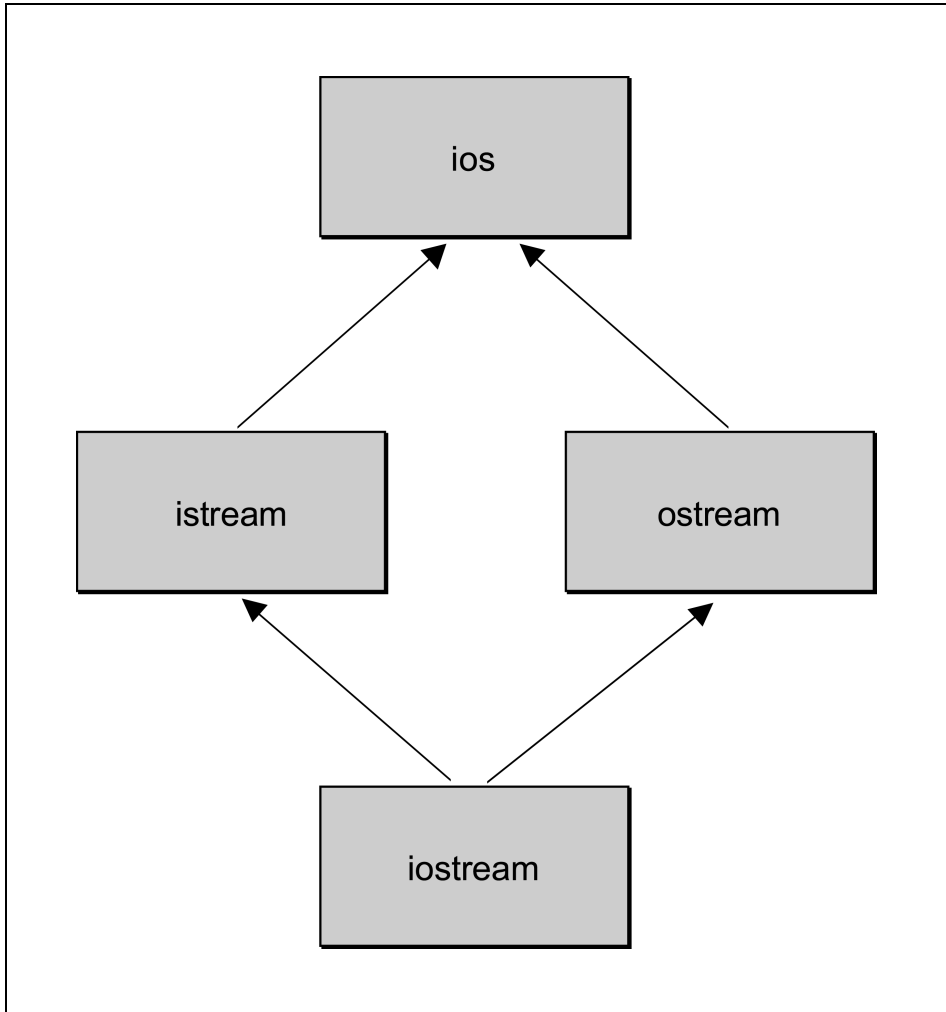
# Kapitel 4

## Ein- und Ausgaben mit Streams

Dieses Kapitel beschreibt den Einsatz der Streams für die Ein- und Ausgabe. Der Schwerpunkt liegt dabei auf den Möglichkeiten zur Formatierung.

# Streams

## Stream-Klassen für die Ein- und Ausgabe



### Die vier Standard-Streams

- **`cin`**     Objekt der Klasse `istream` für die Standardeingabe
- **`cout`**    Objekt der Klasse `ostream` für die Standardausgabe
- **`cerr`**    Objekt der Klasse `ostream` zur ungepufferten Fehlerausgabe
- **`clog`**    Objekt der Klasse `ostream` zur gepufferten Fehlerausgabe



## I/O-Stream-Klassen

Mit der Entwicklung von C++ wurde ein neues Ein-/Ausgabe-System auf der Basis von Klassen implementiert. Es entstanden die *I/O-Stream-Klassen*, die in einer eigenen Bibliothek, der sog. *iostream-Bibliothek*, zur Verfügung stehen.

Das nebenstehende Diagramm zeigt eine sog. Klassenhierarchie, die durch Vererbung entsteht. Die Klasse `ios` ist die Basisklasse aller Stream-Klassen: Sie stellt die Eigenschaften und Fähigkeiten dar, die allen Streams gemeinsam sind. Dazu gehören im Wesentlichen die folgenden Aufgaben:

- Die Klasse `ios` verwaltet die Verbindung zum eigentlichen Datenstrom, der z.B. die Daten Ihres Programms in eine Datei oder auf dem Bildschirm ausgibt.
- Die Klasse `ios` stellt die grundlegende Funktionalität für die Formatierung der Daten zur Verfügung. Dazu werden eine Reihe von Flags definiert, die festlegen, wie z.B. Zeichen bei der Eingabe interpretiert werden.

Eine komfortable Schnittstelle für das Arbeiten mit Streams bieten die von `ios` abgeleiteten Klassen `istream` und `ostream`. Die Klasse `istream` ist für das Lesen und `ostream` ist für das Schreiben in Streams konzipiert. Hier sind beispielsweise die Operatoren `>>` und `<<` definiert.

Die Klasse `iostream` entsteht durch Mehrfachvererbung aus `istream` und `ostream`. Sie stellt damit die Funktionalität beider Klassen zur Verfügung.

Von den genannten Klassen sind weitere Stream-Klassen abgeleitet, beispielsweise für die Dateiverarbeitung. Das bedeutet, dass die hier beschriebenen Techniken auch beim Lesen und Schreiben in Dateien eingesetzt werden können. Diese Klassen, die z.B. auch Methoden für das Öffnen und Schließen von Dateien enthalten, werden in einem späteren Kapitel vorgestellt.

## Standard-Streams

Die bereits bekannten Streams `cin` und `cout` sind Objekte vom Typ der Klasse `istream` bzw. `ostream`. Beim Programmstart werden diese beiden Objekte automatisch für das Lesen von der *Standardeingabe* bzw. für das Schreiben auf die *Standardausgabe* angelegt.

Die Standardeingabe ist normalerweise die Tastatur, die Standardausgabe der Bildschirm. Standardein- und -ausgabe können aber auch in Dateien umgelenkt werden. Dann werden Daten nicht von der Tastatur, sondern aus einer Datei eingelesen bzw. Daten nicht am Bildschirm angezeigt, sondern in eine Datei geschrieben.

Die zwei weiteren Standard-Streams `cerr` und `clog` werden dazu benutzt, um beim Auftreten eines Fehlers eine Meldung anzuzeigen. Die Anzeige erfolgt auch dann auf dem Bildschirm, wenn die Standardausgabe in eine Datei umgelenkt worden ist.

## Formatierung und Manipulatoren

### Beispiel für einen Manipulator-Aufruf

Hier wird der Manipulator `showpos` aufgerufen.



```
cout << showpos << 123;    // Ausgabe:  +123
```

Die vorstehende Anweisung ist äquivalent zu:

```
cout.setf( ios::showpos );  
cout << 123;
```

Alle weiteren Ausgaben positiver Zahlen ebenfalls mit Vorzeichen:

```
cout << 22;                // Ausgabe:  +22
```

Die Anzeige des positiven Vorzeichens kann durch den Manipulator `noshowpos` wieder rückgängig gemacht werden:

```
cout << noshowpos << 123;  // Ausgabe:  123
```

Die letzte Anweisung ist äquivalent zu:

```
cout.unsetf( ios::showpos );  
cout << 123;
```



Hinweise:

- Die Operatoren `>>` und `<<` formatieren ihre Ein- bzw. Ausgabe gemäß den Einstellungen der Flags in der Basisklasse `ios`.
- Der Manipulator `showpos` ist eine Funktion, die ihrerseits die Methode `cout.setf( ios::showpos );` aufruft. Dabei ist `ios::showpos` das Flag `showpos` in der Klasse `ios`.
- Der Einsatz von Manipulatoren ist deutlich einfacher als die direkte Veränderung der Flags. Daher werden im folgenden die Manipulatoren beschrieben und nur in Ausnahmefällen die Methoden `setf()` und `unsetf()` verwendet.
- Für ältere Compiler, welche die Manipulatoren noch nicht unterstützen, müssen die Methoden `setf()` und `unsetf()` trotzdem verwendet werden.

## Formatierungen

Beim Einlesen von der Tastatur muss festgelegt sein, wie die Eingabe zu interpretieren ist und welche Formate zulässig sind. Umgekehrt erfolgt die Ausgabe auf dem Bildschirm nach bestimmten Regeln, die die Darstellung z.B. von Gleitpunktzahl bestimmen.

Für derartige Formatierungen stellen die Stream-Klassen `istream` und `ostream` verschiedene Möglichkeiten zur Verfügung. So können beispielsweise Zahlen in einfacher Weise tabellarisch angezeigt werden.

In den vorhergehenden Kapiteln haben Sie die Streams `cin` und `cout` bereits in Anweisungen wie

```
cout << "Bitte eine Zahl eingeben: ";
cin  >> x;
```

kennengelernt. Im folgenden werden die wichtigsten Fähigkeiten der Streams systematisch beschrieben. Hierzu gehören:

- Die Operatoren `>>` und `<<` für die formatierte Ein- und Ausgabe.  
Diese Operatoren sind für Ausdrücke mit einem elementaren Datentyp also für Zeichen, boolesche Werte und Zahlen sowie für Strings definiert.
- Manipulatoren, die in den Ein- bzw. Ausgabestrom „eingefügt“ werden.  
Mit Manipulatoren können z.B. auf einfache Weise neue Formatierungen für nachfolgende Ein-/Ausgaben festgelegt werden. Ein bereits bekannter Manipulator ist `endl`, der bei der Ausgabe einen Zeilenvorschub auslöst.
- Weitere Methoden, z.B. um den Status des Streams abzufragen oder zu ändern sowie für die unformatierte Ein- und Ausgabe.

## Flags und Manipulatoren

In welcher Form Zeichen ein- bzw. ausgegeben werden, wird auch durch so genannte *Flags* (dt. Schalter) in der Basisklasse `ios` gesteuert. Flags werden durch einzelne Bits innerhalb einer ganzzahligen Variablen repräsentiert, die eine spezielle Bedeutung haben. Ein gesetztes oder nicht gesetztes Bit legt z.B. fest, ob positive Zahlen mit oder ohne ein führendes Plus-Zeichen ausgegeben werden.

Jedes Flag besitzt eine *Standardeinstellung*. Damit ist z.B. von vornherein festgelegt, dass ganze Zahlen dezimal ausgegeben und positive Zahlen ohne Plus-Zeichen dargestellt werden.

Es ist möglich, einzelne Format-Flags zu verändern. Dafür gibt es z.B. die Methoden `setf()` und `unsetf()`. Auf einfachere Weise kann dies jedoch mit Hilfe von sog. *Manipulatoren* geschehen, die für alle wichtigen Flags definiert sind. Manipulatoren sind Funktionen, die in den Ein- bzw. Ausgabestrom „eingefügt“ und dadurch aufgerufen werden.

# Formatierte Ausgabe von Ganzzahlen

## Manipulatoren zur Formatierung von Ganzzahlen

Manipulator	Wirkung
oct	Oktale Darstellung
hex	Hexadezimale Darstellung
dec	Dezimale Darstellung (Standard)
showpos	Positive Zahlen werden mit Vorzeichen ausgegeben.
noshowpos	Positive Zahlen werden ohne Vorzeichen ausgegeben (Standard).
uppercase	Bei der Ausgabe von Hexadezimalzahlen werden Großbuchstaben verwendet.
nouppercase	Bei der Ausgabe von Hexadezimalzahlen werden Kleinbuchstaben verwendet (Standard).

## Beispielprogramm

```
// Ganze Zahl dezimal einlesen und
// oktal, dezimal und hexadezimal ausgeben.

#include <iostream>      // Deklaration von cin, cout und
using namespace std;    // den Manipulatoren oct, ...

int main()
{
    int zahl;
    cout << "Eine ganze Zahl eingeben: ";
    cin >> zahl;

    cout << uppercase           // für Hex-Ziffern
         << " oktal  \t dezimal  \t hexadezimal\n "
         << oct << zahl << "      \t "
         << dec << zahl << "      \t "
         << hex << zahl << endl;

    return 0;
}
```

## Formatierungsmöglichkeiten

Der Operator `<<` kann ganze Zahlen vom Typ `short`, `int`, `long` oder einem entsprechenden `unsigned`-Typ ausgeben. Dabei sind u.a. folgende Formatierungen möglich:

- Festlegung des Zahlensystems, in dem die Zahl dargestellt wird: dezimal, oktal oder hexadezimal
- Die Verwendung von Groß- statt Kleinbuchstaben für hexadezimale Ziffern
- Anzeige des Vorzeichens bei positiven dezimalen Zahlen

Außerdem kann eine Feldbreite festgelegt werden. Die Angabe einer Feldbreite ist auch bei der Ausgabe von Zeichen, Strings und Gleitpunktzahlen möglich und wird anschließend behandelt.

## Zahlensystem

Standardmäßig werden ganze Zahlen dezimal dargestellt. Für den Wechsel in die oktale oder hexadezimale bzw. zurück in die dezimale Darstellung gibt es die Manipulatoren `oct`, `hex` und `dec`.

**Beispiel:** `cout << hex << 11; // Ausgabe: b`

Die hexadezimalen Ziffern werden standardmäßig mit Kleinbuchstaben angezeigt, also mit `a`, `b`, ..., `f`. Sollen Großbuchstaben verwendet werden, so ermöglicht dies der Manipulator `uppercase`.

**Beispiel:** `cout << hex << uppercase << 11; // Ausgabe: B`

Der Manipulator `nouppercase` stellt die Ausgabe wieder auf Kleinbuchstaben um.

## Negative Zahlen

Bei der Ausgabe von negativen Zahlen in *dezimaler* Darstellung wird deren Vorzeichen immer ausgegeben. Um auch positive Zahlen mit Vorzeichen zu versehen, kann der Manipulator `showpos` verwendet werden.

**Beispiel:** `cout << dec << showpos << 11; // Ausgabe: +11`

Mit `noshowpos` kann die ursprüngliche Einstellung wiederhergestellt werden.

Bei *oktaler* oder *hexadezimaler* Ausgabe wird das Bitmuster der angegebenen Zahl stets *ohne Vorzeichen* interpretiert! Es wird also das Bitmuster der Zahl in oktaler bzw. hexadezimaler Form angezeigt.

**Beispiel:** `cout << dec << -1 << " " << hex << -1;`

Auf einem 32-Bit-System erzeugt diese Anweisung die Ausgabe:

```
-1    ffffffff
```

# Formatierte Ausgabe von Gleitpunktzahlen

## Manipulatoren zur Formatierung von Gleitpunktzahlen

Manipulator	Wirkung
showpoint	Der Dezimalpunkt wird stets angezeigt. Es werden so viele Ziffern angezeigt, wie es der eingestellten Genauigkeit entspricht.
noshowpoint	Abschließende Nullen hinter dem Dezimalpunkt werden nicht angezeigt. Folgt dem Dezimalpunkt keine Ziffer, wird der Dezimalpunkt nicht angezeigt. (Standardeinstellung)
fixed	Darstellung als Festpunktzahl
scientific	Darstellung in exponentieller Notation
setprecision(int n)	Setzt die Genauigkeit auf n

## Methoden für die Genauigkeit

Methode	Wirkung
int precision(int n);	Setzt die Genauigkeit auf n
int precision() const;	Liefert den aktuellen Wert für die Genauigkeit zurück.



Das Schlüsselwort `const` im Prototyp von `precision()` bedeutet, dass die Methode nur lesende Operationen ausführt.

## Beispielprogramm

```
#include <iostream>
using namespace std;
int main()
{
    double x = 12.0;
    cout.precision(2); // Genauigkeit 2
    cout << " Standard:  " << x << endl;
    cout << " showpoint: " << showpoint << x << endl;
    cout << " fixed:     " << fixed << x << endl;
    cout << " scientific: " << scientific << x << endl;
    return 0;
}
```

## Standardeinstellungen

Gleitpunktzahlen werden standardmäßig mit einer Genauigkeit von sechs Dezimalziffern ausgegeben. Hierbei wird ein Dezimalpunkt, also kein Komma, verwendet. Abschließende Nullen nach dem Dezimalpunkt werden nicht angezeigt. Folgt dem Dezimalpunkt keine Ziffer, so wird auch kein Dezimalpunkt angezeigt.

**Beispiele:**

```
cout << 1.0;           // Ausgabe: 1
cout << 1.234;         // Ausgabe: 1.234
cout << 1.234567;      // Ausgabe: 1.23457
```

Die letzte Anweisung zeigt, dass die siebte Ziffer nicht abgeschnitten wird, sondern dass gerundet wird. Bei sehr großen und sehr kleinen Zahlen wird die *exponentielle Darstellung* verwendet.

**Beispiel:**

```
cout << 1234567.8;     // Ausgabe: 1.23457e+06
```

## Formatierungsmöglichkeiten

Die Standardeinstellungen können weitgehend angepasst werden. So kann

- der Wert für die Genauigkeit geändert,
- die Ausgabe des Dezimalpunkts mit abschließenden Nullen erzwungen und
- die Art der Darstellung (Festpunkt oder exponentiell) festgelegt werden.

Sowohl der Manipulator `setprecision()` als auch die Methode `precision()` setzen den Wert der Genauigkeit neu.

**Beispiel:**

```
cout << setprecision(3); // Genauigkeit: 3
// oder: cout.precision(3);
cout << 12.34;           // Ausgabe: 12.3
```

Dabei ist zu beachten, dass für den Manipulator `setprecision()` die Header-Datei `iomanip` zu inkludieren ist. Das gilt für alle Standardmanipulatoren, die mit mindestens einem Argument aufgerufen werden.

Der Manipulator `showpoint` bewirkt, dass stets der Dezimalpunkt und abschließende Nullen angezeigt werden. Die Gesamtzahl der ausgegebenen Ziffern entspricht dann der aktuell eingestellten Genauigkeit (z.B. 6).

**Beispiel:**

```
cout << showpoint << 1.0; // Ausgabe: 1.00000
```

Oft ist jedoch die *Festpunktdarstellung* mit einer festgelegten Anzahl von Nachpunktstellen zweckmäßiger. Diese Darstellung wird mit dem Manipulator `fixed` erreicht. Dabei bestimmt die Genauigkeit die Anzahl der Nachpunktstellen. Im folgenden Beispiel wird der Standardwert 6 angenommen.

**Beispiel:**

```
cout << fixed << 66.0;   // Ausgabe: 66.000000
```

Der Manipulator `scientific` hingegen bestimmt, dass jede Gleitpunktzahl in exponentieller Schreibweise ausgegeben wird. Falls der Manipulator `fixed` noch nicht zur Verfügung steht, kann die Methode `setf()` wie folgt aufgerufen werden:

```
cout.setf(ios::fixed);
```

# Ausgabe in Felder

## Elementfunktionen zur Feldausgabe

Methode	Wirkung
<code>int width() const;</code>	Liefert die aktuelle Feldbreite
<code>int width(int n);</code>	Setzt die Feldbreite auf n
<code>int fill() const;</code>	Liefert das aktuell gesetzte Füllzeichen
<code>int fill( int ch);</code>	Setzt das Füllzeichen auf ch

## Manipulatoren zur Feldausgabe

Manipulator	Wirkung
<code>setw(int n)</code>	Setzt die Feldbreite auf n
<code>setfill(int ch)</code>	Setzt das Füllzeichen auf ch
<code>left</code>	Linksbündige Ausgabe im Feld
<code>right</code>	Rechtsbündige Ausgabe im Feld
<code>internal</code>	Vorzeichen linksbündig, Wert rechtsbündig im Feld



Die Manipulatoren `setw()` und `setfill()` sind in der Header-Datei `iomanip` deklariert.

## Beispiele

```
#include <iostream>           // die notwendigen
#include <iomanip>             // Deklarationen
using namespace std;
```

**1. Beispiel:** `cout << '|' << setw(6) << 'X' << '|';`

**Ausgabe:**                   |           X|           // X im Feld der Breite 6

**2. Beispiel:** `cout << fixed << setprecision(2)
<< setw(10) << 123.4 << endl
<< "1234567890" << endl;`

**Ausgabe:**           123.40                               // Feldbreite 10  
                  1234567890



Der Operator `<<` kann seine Ausgabe in *Ausgabefeldern* positionieren. Hierbei ist es möglich

- eine *Feldbreite* vorzugeben
- die *Ausrichtung* im Feld, z.B. rechtsbündig oder linksbündig, zu bestimmen
- ein *Füllzeichen* anzugeben, mit dem das Feld aufgefüllt wird

## Feldbreite

Die Feldbreite ist die Anzahl der Zeichen, die ein Feld aufnehmen kann. Ist die auszugebende Zeichenfolge größer als die Feldbreite, wird die Ausgabe nicht abgeschnitten, sondern das Feld vergrößert. Es werden also mindestens so viele Zeichen ausgegeben, wie die Feldbreite vorgibt.

Die Feldbreite kann entweder mit der Methode `width()` oder mit dem Manipulator `setw()` gesetzt werden.

**Beispiel:** `cout.width(6);` // oder: `cout << setw(6);`

Eine Besonderheit der Feldbreite ist, dass sie nicht permanent ist: Die angegebene Feldbreite gilt stets nur für die nächste Ausgabe! Dies zeigen auch die nebenstehenden Beispiele. Im 1. Beispiel wird nur das Zeichen 'x' in einem Feld der Breite 6 ausgegeben, nicht aber das Zeichen '| '.

Der Standardwert für die Feldbreite ist 0. Die aktuell gesetzte Feldbreite kann mit der Methode `width()` auch abgefragt werden. In diesem Fall wird `width()` ohne Argument aufgerufen.

**Beispiel:** `int feldbreite = cout.width();`

## Füllzeichen und Ausrichtung

Ist das Feld größer als die auszugebende Zeichenfolge, so wird standardmäßig mit Blanks aufgefüllt. Ein neues Füllzeichen kann entweder mit der Methode `fill()` oder mit dem Manipulator `setfill()` festgelegt werden.

**Beispiel:** `cout << setfill('*') << setw(5) << 12;`  
// Ausgabe: \*\*\*12

Das Füllzeichen bleibt so lange gültig, bis ein anderes Zeichen neu gesetzt wird.

Wie das vorhergehende Beispiel schon zeigt, erfolgt die Ausgabe in einem Feld standardmäßig „rechtsbündig“. Weitere Möglichkeiten sind „linksbündig“ und „intern“. Hierfür gibt es die Manipulatoren `left` und `internal`. Der Manipulator `internal` setzt das Vorzeichen einer Zahl linksbündig und die Zahl rechtsbündig in ein Feld.

**Beispiel:** `cout.width(6); cout.fill('0');`  
`cout << internal << -123; // Ausgabe: -00123`

# Ausgabe von Zeichen, Strings und booleschen Werten

## Beispielprogramm

```
// Ein Zeichen einlesen und den Zeichencode
// okt, dezimal und hexadezimal ausgeben.

#include <iostream>      // Deklaration von cin, cout
#include <iomanip>        // Für Manipulatoren, die mit
                        // Argumenten aufgerufen werden.
#include <string>
using namespace std;

int main()
{
    int zahl = ' ';

    cout << "Das Leerzeichen hat den Zeichencode: "
          << zahl << endl;

    char ch;
    string prompt =
        "\nGeben Sie ein Zeichen und <Return> ein: ";

    cout << prompt;

    cin >> ch;                // Zeichen einlesen
    zahl = ch;

    cout << "Das Zeichen " << ch
          << " hat den Zeichencode" << endl;

    cout << uppercase          // für Hex-Ziffern
          << "      okt    dezimal  hexadezimal\n "
          << oct << setw(8) << zahl
          << dec << setw(8) << zahl
          << hex << setw(8) << zahl << endl;

    return 0;
}
```

## Ausgabe von Zeichen und Zeichencodes

Der Operator `<<` interpretiert eine Zahl mit dem Datentyp `char` als Zeichencode eines Zeichens und gibt das entsprechende Zeichen aus:

**Beispiel:**

```
char ch = '0';
cout << ch << ' ' << 'A';
// Ausgabe der drei Zeichen: 0 A
```

Es ist auch möglich, den Zeichencode eines Zeichens auszugeben. Dazu wird der Zeichencode in einer `int`-Variablen gespeichert und diese ausgegeben.

**Beispiel:**

```
int code = '0';
cout << code;           // Ausgabe: 48
```

Das Zeichen `'0'` wird durch den ASCII-Code 48 repräsentiert. Weitere Beispiele enthält das nebenstehende Programm.

## Ausgabe von Strings

Mit dem Operator `<<` können sowohl String-Literale wie `"Hallo"` als auch String-Variablen ausgegeben werden. Dies wurde auch schon in früheren Beispielen verwendet. Wie bei den anderen Datentypen können auch Strings in Ausgabefelder positioniert werden.

**Beispiel:**

```
string s("Donau so blau");
cout << left                // linksbündig
    << setfill('?')        // Füllzeichen ?
    << setw(20) << s ;     // Feldbreite 20
```

Hier wird also die Zeichenfolge `"Donau so blau???????"` ausgegeben. Mit dem Manipulator `right` kann wieder die rechtsbündige Ausrichtung im Feld eingestellt werden.

## Ausgabe von booleschen Werten

Boolesche Werte gibt der Operator `<<` standardmäßig ganzzahlig aus. Dabei steht der Wert 0 für `false` und der Wert 1 für `true`. Sollen statt dessen die Zeichenfolgen `false` bzw. `true` angezeigt werden, so muss das Flag `ios::boolalpha` gesetzt werden. Dies kann direkt mit der Methode `setf()` geschehen oder mit dem Manipulator `boolalpha`.

**Beispiel:**

```
bool ok = true;
cout << ok << endl          // 1
    << boolalpha << ok << endl; // true
```

Diese Einstellung kann mit dem Manipulator `noboolalpha` wieder rückgängig gemacht werden.

# Formatierte Eingabe

## Beispielprogramm

```
// Eine Artikelbezeichnung und einen Preis einlesen

#include <iostream>      // Deklaration von cin, cout,...
#include <iomanip>        // Für den Manipulator setw()
#include <string>
using namespace std;

int main()
{
    string bezeichnung;
    double preis;

    cout << "\nGeben Sie die Artikelbezeichnung ein: ";

    // höchstens 16 Zeichen für die Bezeichnung einlesen:
    cin >> setw(16);      // oder:  cin.width(16);
    cin >> bezeichnung;
    cin.sync();          // Puffer leeren
    cin.clear();          // evtl. gesetzte Fehlerflags löschen

    cout << "\nGeben Sie den Artikelpreis ein: ";
    cin >> preis;         // Preis einlesen

    // Kontrollausgabe:
    cout << fixed << setprecision(2)
         << "\nArtikel:"
         << "\n  Bezeichnung:  " << bezeichnung
         << "\n  Preis:       " << preis << endl;

    // ... und weiter im Programm


    return 0;
}
```



Mit dem Aufruf der Methoden `sync()` und `clear()` wird der Eingabepuffer gelöscht, und evtl. gesetzte Fehlerflags werden gelöscht. Dadurch ist sichergestellt, dass beim Einlesen des Preises auch dann auf eine neue Eingabe gewartet wird, wenn für die Artikelbezeichnung mehr als 16 Zeichen eingegeben wurden.

Der Operator `>>` der Klasse `istream` berücksichtigt bei der Eingabe die aktuell gesetzten Flags für die Zahlenbasis und die Feldbreite.

- Die Basis des Zahlensystems bestimmt, ob eine Ganzzahl dezimal, oktal oder hexadezimal eingelesen wird.
- Die Feldbreite legt für Strings die Anzahl der Zeichen fest, die maximal eingelesen werden.

Das Lesen von der Standardeingabe `cin` ist *zeilenweise gepuffert*. Die Eingabe von der Tastatur wird also erst durch das abschließende Return  „abgeschickt“. Das hat zur Folge, dass Fehler bei der Eingabe noch mit der Backspace-Taste korrigiert werden können, solange nicht die Return-Taste gedrückt wurde. Die Eingabe wird auf dem Bildschirm angezeigt.

## Eingabefelder

Generell liest der Operator `>>` das nächste *Eingabefeld*, konvertiert die Eingabe anhand des Typs der angegebenen Variablen und legt das Ergebnis in der Variablen ab. Dabei werden führende Zwischenraumzeichen (Leer-, Tabulator- und Newline-Zeichen) überlesen.

**Beispiel:**     `char ch;`  
                  `cin >> ch;`             `// Ein Zeichen einlesen`


Bei Eingabe von

`<Return> <Tab> <Leertaste> <X> <Return>`

wird das Zeichen 'X' an die Variable `ch` zugewiesen.

Ein Eingabefeld endet mit dem ersten Zwischenraumzeichen bzw. mit dem ersten Zeichen, das nicht mehr verarbeitet werden kann.

**Beispiel:**     `int i;`  
                  `cin >> i;`

Bei der Eingabe von `123 Euro`  wird der dezimale Wert 123 an `i` zugewiesen. Die übrigen Zeichen, `Euro` und das Newline-Zeichen, verbleiben im Eingabepuffer. Sie werden bei der nächsten Leseoperation als erstes gelesen.

Beim Einlesen von Strings wird jeweils nur ein Wort gelesen, da ja mit einem Zwischenraumzeichen das nächste Eingabefeld beginnt.

**Beispiel:**     `string ort;`  
                  `cin >> ort;`             `// Liest nur ein Wort!`

Bei der Eingabe `Bad Tölz` wird also nur `Bad` in den String `ort` eingelesen. Durch die Angabe einer Feldbreite kann zusätzlich die Anzahl der einzulesenden Zeichen begrenzt werden. Bei einer Feldbreite `n` werden höchstens `n` Zeichen gelesen. Dabei zählen führende Zwischenraumzeichen nicht mit. Ein Beispiel enthält das nebenstehende Programm. Es zeigt auch, wie der Eingabepuffer gelöscht werden kann.

# Formatierte Eingabe von Zahlen

## Beispielprogramm

```
// Hex-Zahl und Gleitpunktzahlen einlesen
//
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int zahl = 0;

    cout << "\nGeben Sie eine Zahl hexadezimal ein: "
          << endl;
    cin >> hex >> zahl;           // Hex-Zahl einlesen

    cout << "Ihre Eingabe dezimal: " << zahl << endl;
    // Falls eine fehlerhafte Eingabe gemacht wurde:
    cin.sync();                   // Puffer leeren
    cin.clear();                   // Fehlerflags löschen

    double x1 = 0.0, x2 = 0.0;

    cout << "\nGeben Sie nun zwei Gleitpunktzahlen ein: "
          << endl;
    cout << "1. Zahl: ";
    cin >> x1;                     // 1. Zahl einlesen
    cout << "2. Zahl: ";
    cin >> x2;                     // 2. Zahl einlesen

    cout << fixed << setprecision(2)
          << "\nDie Summe der beiden Zahlen:   "
          << setw(10) << x1 + x2 << endl;

    cout << "\nDas Produkt der beiden Zahlen: "
          << setw(10) << x1 * x2 << endl;

    return 0;
}
```

## Einlesen von Ganzzahlen

Mit den Manipulatoren `hex`, `oct` und `dec` kann festgelegt werden, dass eine eingegebene Zeichenfolge als Hexadezimal-, Oktal- oder Dezimalzahl interpretiert wird.

**Beispiel:**     `int n;`  
                  `cin >> oct >> n;`

Bei der Eingabe von 10 wird die Zahl oktal interpretiert, was dem dezimalen Wert 8 entspricht. Mit

**Beispiel:**     `cin >> hex >> n;`

wird die Eingabe als Hexadezimalzahl interpretiert. Eingaben wie `f0a` oder `-F7` sind dann möglich.

## Einlesen von Gleitpunktzahlen

Der Operator `>>` interpretiert die Eingabe als dezimale Gleitpunktzahl, wenn der Typ der Variablen ein Gleitpunkttyp ist, also `float`, `double` oder `long double`. Die Gleitpunktzahl kann als Festpunktzahl oder in exponentieller Form eingegeben werden.

**Beispiel:**     `double x;`  
                  `cin >> x;`

Hier wird die eingegebene Zeichenfolge in einen `double`-Wert konvertiert. Eingaben wie `123` oder `-22.0` oder `3e10` sind zulässig.

## Fehlerhafte Eingaben

Was geschieht nun, wenn die Eingabe nicht zum Typ der entsprechenden Variablen passt?

**Beispiel:**     `int i, j;     cin >> i >> j;`

Bei Eingabe von `1A5` beispielsweise wird 1 an die Variable `i` zugewiesen. Mit `A` beginnt das nächste Eingabefeld. Da aber wieder dezimale Ziffern erwartet werden, bleibt die Zeichenfolge ab `A` unverarbeitet. Findet, wie in diesem Fall, keine Konvertierung statt, so wird der entsprechenden Variablen – hier `j` – kein Wert zugewiesen und intern ein Fehlerflag gesetzt.

Im Allgemeinen ist es sinnvoller, Zahlenwerte einzeln einzulesen und nach jeder Eingabe den Eingabepuffer und eventuell gesetzte Fehlerflags zu löschen.

Wie ein Programm auf einen Fehler bei der Eingabe reagieren kann, wird in Kapitel 6, *Kontrollstrukturen*, und später in Kapitel 28, *Ausnahmebehandlung*, gezeigt.

# Unformatierte Ein-/Ausgabe

## Beispielprogramm

```
// Text mit dem Operator >> und
// mit der Funktion getline() einlesen.

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

string header =
"    --- Beispielprogramm für unformatierte Eingabe ---";

int main()
{
    string wort, rest;

    cout << header
         << "\n\nWeiter mit <Return>" << endl;
    cin.get();                // Newline einlesen,
                             // aber nicht speichern.

    cout << "\nBitte Text mit mehreren Worten eingeben!"
         << "\nMit <!> und <Return> abschließen."
         << endl;

    cin >> wort;              // 1. Wort einlesen und
    getline( cin, rest, '!'); // den Rest der Eingabe
                             // bis zum Zeichen !

    cout << "\nDas erste Wort:   " << wort
         << "\nRest der Eingabe: " << rest << endl;

    return 0;
}
```



1. Es kann auch ein mehrzeiliger Text eingegeben werden.
2. Das Beispielprogramm geht davon aus, dass wenigstens ein Wort (mit nachfolgendem Zwischenraumzeichen) eingegeben wird.



Die unformatierte Ein- und Ausgabe benutzt keine Felder und intern gesetzte Formatierungsflags bleiben unberücksichtigt. Die aus einem Stream gelesenen „Bytes“ werden unverändert an das Programm übergeben. Insbesondere werden führende Zwischenraumzeichen nicht überlesen.

## Zeichen lesen und schreiben

Einzelne Zeichen können mit den Methoden `get()` und `put()` gelesen bzw. geschrieben werden. Die Methode `get()` liest das nächste Zeichen aus dem Stream und überträgt es in die angegebene `char`-Variable.

**Beispiel:**    `char ch;`  
              `cin.get(ch);`

Ist das gelesene Zeichen ein Zwischenraumzeichen, z.B. ein Newline-Zeichen, so wird dieses in die Variable `ch` übertragen. Dagegen würde mit der Anweisung

```
cin >> ch;
```

das erste Zeichen gelesen, das kein Zwischenraumzeichen ist.

Die Methode `get()` kann auch ohne ein Argument aufgerufen werden. In diesem Fall liefert `get()` den Zeichencode als Return-Wert vom Typ `int`.

**Beispiel:**    `int c = cin.get();`

Für die unformatierte Ausgabe eines Zeichens steht die Methode `put()` zur Verfügung. Als Argument erhält `put()` das auszugebende Zeichen.

**Beispiel:**    `cout.put('A');`

Diese Anweisung ist äquivalent zu `cout << 'A';`, falls keine Feldbreite angegeben oder die Feldbreite auf 1 gesetzt ist.

## Eine Zeile einlesen

Mit dem Operator `>>` kann immer nur ein Wort in einen String eingelesen werden. Soll eine ganze Textzeile eingelesen werden, so kann die schon vorgestellte globale Funktion `getline()` aufgerufen werden.

**Beispiel:**    `getline(cin, text);`

Hier werden so lange Zeichen von `cin` gelesen und in der Stringvariablen `text` abgespeichert, bis das Newline-Zeichen auftritt. Als Begrenzungszeichen kann auch ein anderes Zeichen bestimmt werden. Dieses Zeichen wird der Funktion `getline()` als drittes Argument übergeben.

**Beispiel:**    `getline(cin, s, '.');`

Das Begrenzungszeichen wird zwar gelesen, aber nicht im String abgespeichert. Zeichen, die in diesem Beispiel nach dem ersten Punkt eingegeben werden, befinden sich noch im Eingabepuffer des Streams.

# Übungen

## Bildschirmausgabe zur 3. Aufgabe:

Artikelnummer	Stückzahl	Stückpreis
.....	.....	..... EURO

## Listing zur 5. Aufgabe:

```
// Ein Programm mit hartnäckigen Fehlern

#include <iostream>
using namespace std;

int main()
{
    char ch;
    string wort;

    cin >> "Los geht's mit der Return-Taste: " >> ch;

    cout << "Geben Sie ein Wort
           mit höchstens drei Zeichen ein: ";

    cin >> setprecision(3) >> wort;

    cout >> "Ihre Eingabe: " >> ch >> endl;

    return 0;
}
```

### 1. Aufgabe

Welche Ausgaben erzeugt das Programm auf der Seite „*Formatierte Ausgabe von Gleitpunktzahlen*“ in diesem Kapitel?

### 2. Aufgabe

Formulieren Sie die entsprechenden Anweisungen:

- a) Die Zahl 0.123456 soll linksbündig in ein Feld mit der Feldbreite 15 ausgegeben werden.
- b) Die Zahl 23.987 soll als Festpunktzahl, auf zwei Ziffern hinter dem Dezimalpunkt gerundet, rechtsbündig in ein Feld der Breite 12 ausgegeben werden.
- c) Die Zahl -123.456 soll in exponentieller Darstellung und mit vier Stellen nach dem Dezimalpunkt ausgegeben werden. Wäre eine Feldbreite 10 sinnvoll?

### 3. Aufgabe

Schreiben Sie ein C++-Programm, das eine Artikelnummer, eine Stückzahl und einen Stückpreis von der Tastatur einliest und wie nebenstehend gezeigt auf dem Bildschirm ausgibt.

### 4. Aufgabe

Erstellen Sie ein C++-Programm, das einen beliebigen Zeichencode (eine positive Ganzzahl) von der Tastatur einliest und das entsprechende Zeichen und den Zeichencode dezimal, oktal und hexadezimal auf dem Bildschirm ausgibt.



Der Typ der Variablen bestimmt, ob ein Zeichen oder eine Zahl eingelesen bzw. ausgegeben wird.

Warum wird, Ihrer Meinung nach, bei Eingabe der Zahl 336 das Zeichen P ausgegeben?

### 5. Aufgabe:

Korrigieren Sie die Fehler im nebenstehenden Programm.

# Lösungen

## Zur 1. Aufgabe:

Ausgabe des Beispielprogramms zur Formatierung von Gleitpunktzahlen:

```
Standard: 12
showpoint: 12.
fixed: 12.00
scientific: 1.20e+001
```

## Zur 2. Aufgabe:

```
#include <iostream>
#include <iomanip> // Für setw() und setprecision()
using namespace std;

int main()
{
    double x1 = 0.123456, x2 = 23.987, x3 = -123.456;

    // a)
    cout << left << setw(15) << x1 << endl;

    // b)
    cout << fixed << setprecision(2) << right << setw(12)
        << x2 << endl;

    // c)
    cout << scientific << setprecision(4) << x3 << endl;
    // Ausgabe: -1.2346e+002
    // Hier wäre also nur eine Feldbreite von 12 oder
    // größer sinnvoll!

    return 0;
}
```

## Zur 3. Aufgabe:

```
// Artikeldaten einlesen und formatiert ausgeben.
//
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    long nummer = 0;
    int anzahl = 0;
```

```

double preis = 0.0;

// Eingabe:
cout << "\nBitte Artikeldaten eingeben.\n";
cout << "Artikelnummer: ";
cin >> nummer;

cout << "Stückzahl: ";
cin >> anzahl;
cout << "Stückpreis: ";
cin >> preis;

// Ausgabe:
cout <<
"\n\tArtikelnummer\t\tStückzahl\t\tStückpreis";

cout << "\n\t"
<< setw(8) << nummer
<< setw(16) << anzahl
<< fixed << setprecision(2)
<< setw(16) << preis << " EURO" << endl;

return 0;
}

```

**Zur 4. Aufgabe:**

```

#include <iostream>
#include <iomanip> // Für den Manipulator setw()
using namespace std;

int main()
{
    unsigned char zeichen = 0;
    unsigned int code = 0;

    cout << "\nBitte dezimal einen Zeichencode eingeben: ";
    cin >> code;

    zeichen = code; // Für die Ausgabe des
                   // entsprechenden Zeichens

    cout << "\nDas zugehörige Zeichen: " << zeichen << endl;

    code = zeichen; // Zeichencode. Nur notwendig,
                   // falls Eingabe > 255 war.

    cout << "\nZeichencodes"
<< "\n dezimal: " << setw(3) << dec << code
<< "\n oktal: " << setw(3) << oct << code

```

## Lösungen (Fortsetzung)

```

        << "\n hexadezimal: " << setw(3) << hex << code
        << endl;

    return 0;
}

```

Bei der Eingabe von 336 wird 80 im niederwertigen Byte der Variablen `code` gespeichert (da  $336 = 256 + 80$ ). Bei der Zuweisung an die Variable `zeichen` erhält daher `zeichen` den Wert 80, was der Zeichencode von `P` ist.

### Zur 5. Aufgabe:

Das korrigierte Programm:

```

// Korrigierte Stellen sind kommentiert.
//
#include <iostream>
#include <iomanip>          // Für den Manipulator setw()
#include <string>          // Für die Klasse string
using namespace std;

int main()
{
    string wort;           // Zum Einlesen eines Wortes.
                           // char ch; wird nicht benötigt.

                           // cout << ... statt cin >> .
    cout << "Los geht's mit der Return-Taste: ";

    cin.get();             // Newline einlesen

    cout << "Geben Sie ein Wort "           // "
         "mit höchstens drei Zeichen ein: "; // "

    cin >> setw(3) >> wort;                 // setw(3) anstatt
                                           // setprecision(3)

    cout << "Ihre Eingabe: "               // <<
         << wort << endl;                  // statt >> ch

    return 0;
}

```

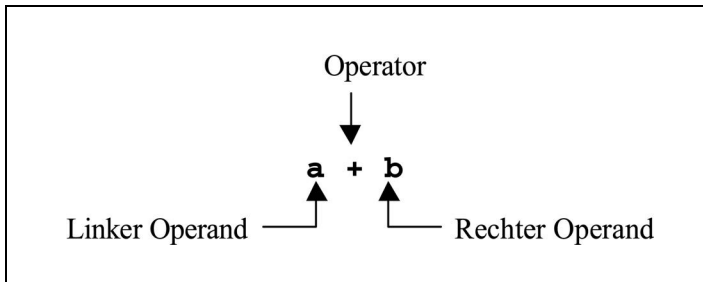
# Kapitel 5

## Operatoren für elementare Datentypen

In diesem Kapitel lernen Sie Operatoren kennen, die für Rechenoperationen und Programmverzweigungen benötigt werden. Weitere Operatoren, z. B. für Bitmanipulationen, und die Überladung von Operatoren werden in späteren Kapiteln behandelt.

# Binäre arithmetische Operatoren

## Binärer Operator und Operanden



## Die binären arithmetischen Operatoren

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulodivision

## Beispielprogramm

```
#include <iostream>
using namespace std;
int main()
{
    double x, y;
    cout << "\nGeben Sie zwei Zahlen ein: ";
    cin >> x >> y;
    cout << "Der Durchschnitt beider Zahlen ist: "
        << (x + y)/2.0 << endl;
    return 0;
}
```

## Beispiel für einen Programmablauf

```
Geben Sie zwei Zahlen ein: 4.75 12.3456
Der Durchschnitt beider Zahlen ist: 8.5478
```